MODBUS APPLICATION PROTOCOL SPECIFICATION
V1.1b3

CONTENTS

1 Introduction 2
1.1 Scope of this document 2
2 Abbreviations 2
3 Context 3
4 General description 3
4.1 Protocol description 3
4.2 Data Encoding 5
4.3 MODBUS Data model 6
4.4 MODBUS Addressing model 7
4.5 Define MODBUS Transaction 8
5 Function Code Categories 10
5.1 Public Function Code Definition 11
6 Function codes descriptions 11
6.1 01 (0x01) Read Coils 11
6.2 02 (0x02) Read Discrete Inputs 12
6.3 03 (0x03) Read Holding Registers 15
6.4 04 (0x04) Read Input Registers 16
6.5 05 (0x05) Write Single Coil 17
6.6 06 (0x06) Write Single Register 19
6.7 07 (0x07) Read Exception Status (Serial Line only) 20
6.8 08 (0x08) Diagnostics (Serial Line only) 21
6.8.1 Sub-function codes supported by the serial line devices 22
6.8.2 Example and state diagram 24
6.9 11 (0x0B) Get Comm Event Counter (Serial Line only) 25
6.10 12 (0x0C) Get Comm Event Log (Serial Line only) 26
6.11 15 (0x0F) Write Multiple Coils 29
6.12 16 (0x10) Write Multiple registers 30
6.13 17 (0x11) Report Server ID (Serial Line only) 31
6.14 20 (0x14) Read File Record 32
6.15 21 (0x15) Write File Record 34
6.16 22 (0x16) Mask Write Register 36
6.17 23 (0x17) Read/Write Multiple registers 38
6.18 24 (0x18) Read FIFO Queue 40
6.19 43 (0x2B) Encapsulated Interface Transport 41
6.20 43 / 13 (0x2B / 0x0D) CANopen General Reference Request and Response PDU 42
6.21 43 / 14 (0x2B / 0x0E) Read Device Identification 43
7 MODBUS Exception Responses 47
Annex A (Informative): MODBUS RESERVED FUNCTION CODES, SUBCODES AND MEI TYPES 50
Annex B (Informative): CANOPEN GENERAL REFERENCE COMMAND 50

IPR2025-00671
TankLogix, LLC v. SitePro, Inc.
TankLogix's Exhibit 1009

EX1009, Page 1

1 Introduction

1.1 Scope of this document

MODBUS is an application layer messaging protocol, positioned at level 7 of the OSI model, which provides client/server communication between devices connected on different types of buses or networks.

The industry's serial de facto standard since 1979, MODBUS continues to enable millions of automation devices to communicate. Today, support for the simple and elegant structure of MODBUS continues to grow. The Internet community can access MODBUS at a reserved system port 502 on the TCP/IP stack.

MODBUS is a request/reply protocol and offers services specified by **function codes**. MODBUS function codes are elements of MODBUS request/reply PDUs. The objective of this document is to describe the function codes used within the framework of MODBUS transactions.

MODBUS is an application layer messaging protocol for client/server communication between devices connected on different types of buses or networks.

It is currently implemented using:

- TCP/IP over Ethernet. See MODBUS Messaging Implementation Guide V1.0a.
- Asynchronous serial transmission over a variety of media (wire : EIA/TIA-232-E, EIA-422, EIA/TIA-485-A; fiber, radio, etc.)
- MODBUS PLUS, a high speed token passing network.

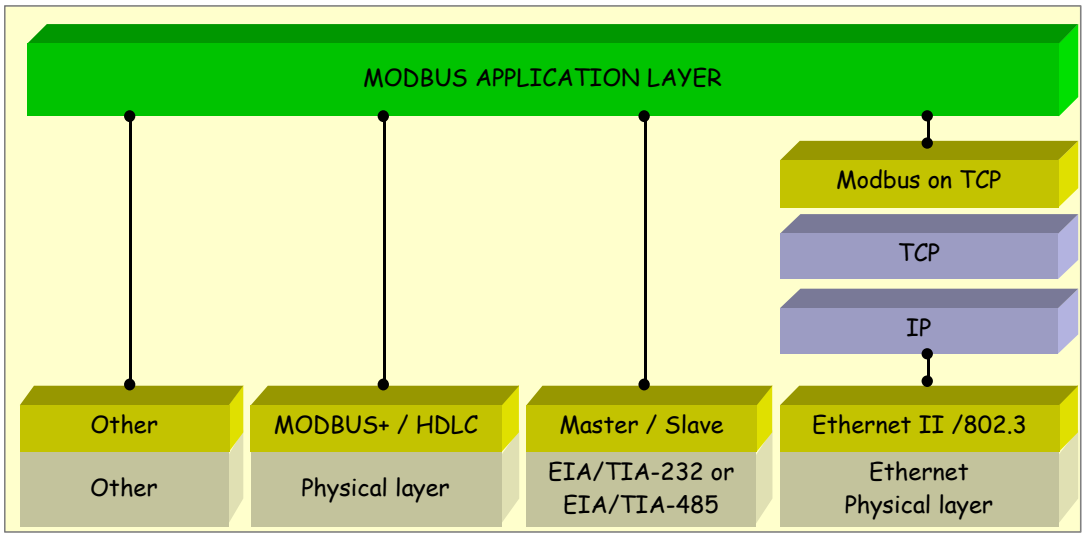

Figure 1: MODBUS communication stack

References

1. RFC 791, Internet Protocol, Sep81 DARPA

2 Abbreviations

ADU Application Data Unit
HDLC High level Data Link Control
HMI Human Machine Interface
IETF Internet Engineering Task Force
I/O Input/Output
IP Internet Protocol
MAC Media Access Control
MB MODBUS Protocol

MBAP MODBUS Application Protocol
PDU Protocol Data Unit
PLC Programmable Logic Controller
TCP Transmission Control Protocol

3 Context

The MODBUS protocol allows an easy communication within all types of network architectures.

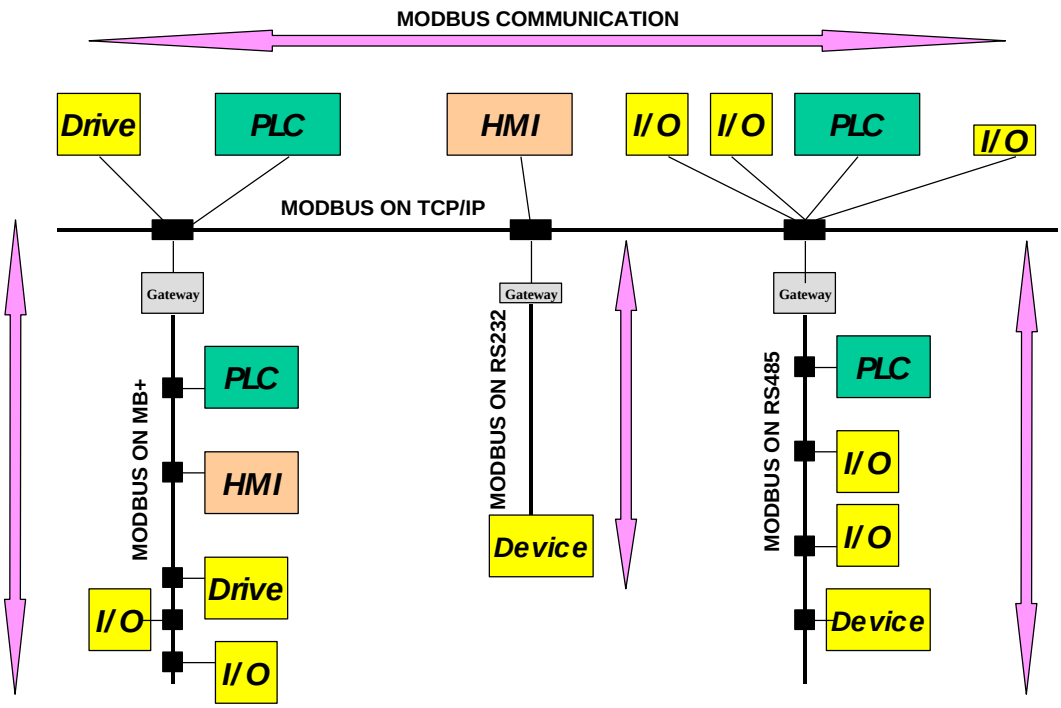

Figure 2: Example of MODBUS Network Architecture

Every type of devices (PLC, HMI, Control Panel, Driver, Motion control, I/O Device…) can use MODBUS protocol to initiate a remote operation.
The same communication can be done as well on serial line as on an Ethernet TCP/IP networks. Gateways allow a communication between several types of buses or network using the MODBUS protocol.

4 General description

4.1 Protocol description

The MODBUS protocol defines a simple protocol data unit **(PDU)** independent of the underlying communication layers. The mapping of MODBUS protocol on specific buses or network can introduce some additional fields on the application data unit **(ADU)**.

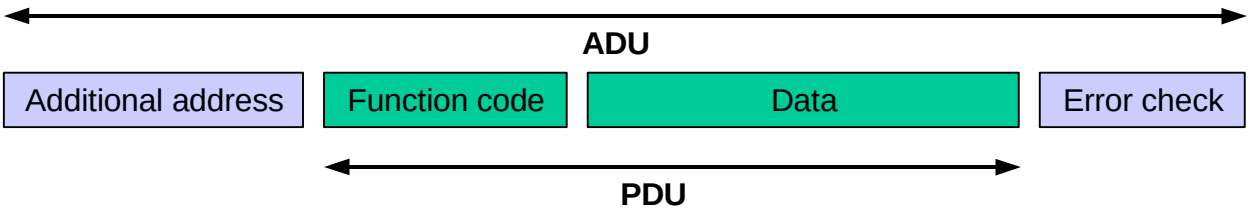

Figure 3: General MODBUS frame

The MODBUS application data unit is built by the client that initiates a MODBUS transaction. The function indicates to the server what kind of action to perform. The MODBUS application protocol establishes the format of a request initiated by a client.

The function code field of a MODBUS data unit is coded in one byte. Valid codes are in the range of 1 ... 255 decimal (the range 128 – 255 is reserved and used for exception responses). When a message is sent from a Client to a Server device the function code field tells the server what kind of action to perform. Function code "0" is not valid.

Sub-function codes are added to some function codes to define multiple actions.

The data field of messages sent from a client to server devices contains additional information that the server uses to take the action defined by the function code. This can include items like discrete and register addresses, the quantity of items to be handled, and the count of actual data bytes in the field.

The data field may be nonexistent (of zero length) in certain kinds of requests, in this case the server does not require any additional information. The function code alone specifies the action.

If no error occurs related to the MODBUS function requested in a properly received MODBUS ADU the data field of a response from a server to a client contains the data requested. If an error related to the MODBUS function requested occurs, the field contains an exception code that the server application can use to determine the next action to be taken.

For example a client can read the ON / OFF states of a group of discrete outputs or inputs or it can read/write the data contents of a group of registers.

When the server responds to the client, it uses the function code field to indicate either a normal (error-free) response or that some kind of error occurred (called an exception response). For a normal response, the server simply echoes to the request the original function code.

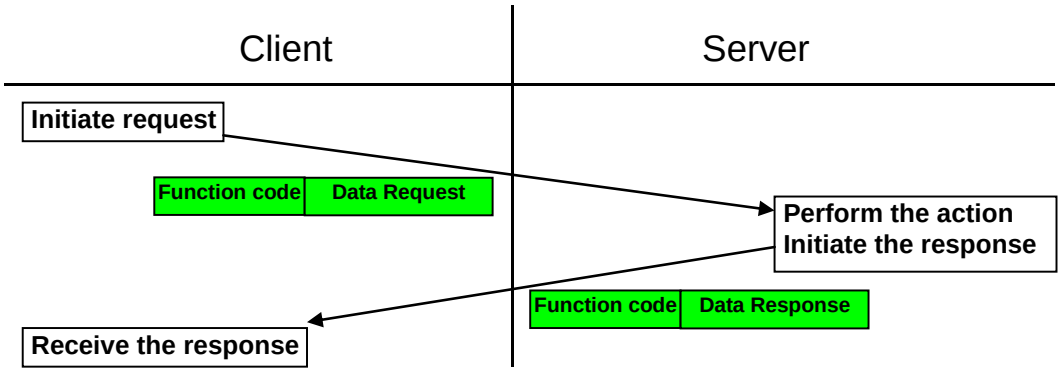

Figure 4: MODBUS transaction (error free)

For an exception response, the server returns a code that is equivalent to the original function code from the request PDU with its most significant bit set to logic 1.

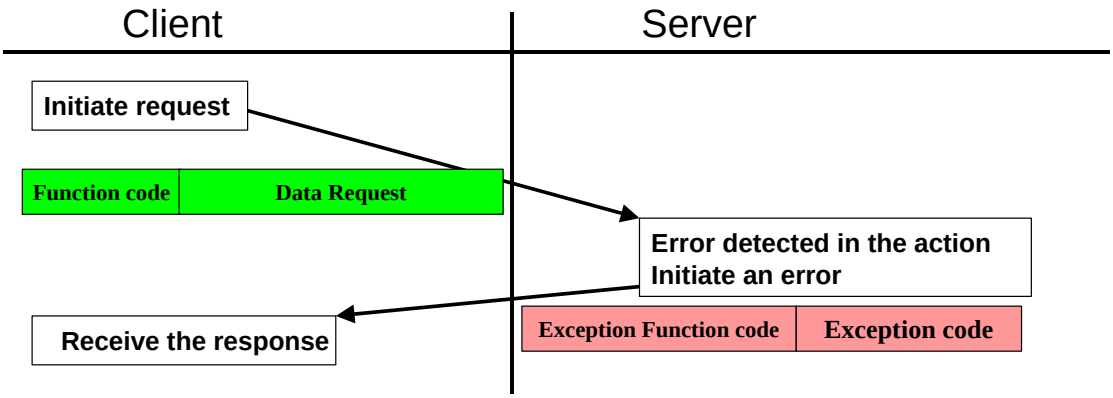

Figure 5: MODBUS transaction (exception response)

☞ **Note**: It is desirable to manage a time out in order not to indefinitely wait for an answer which will perhaps never arrive.

The size of the MODBUS PDU is limited by the size constraint inherited from the first MODBUS implementation on Serial Line network (max. RS485 ADU = 256 bytes).
Therefore:
MODBUS **PDU for serial line communication =** 256 - Server address (1 byte) - CRC (2 bytes) = **253 bytes**.

Consequently:
RS232 / RS485 **ADU** = 253 bytes + Server address (1 byte) + CRC (2 bytes) = **256 bytes**.
TCP MODBUS **ADU** = 253 bytes + MBAP (7 bytes) = **260 bytes**.

The MODBUS protocol defines three PDUs. They are :

- MODBUS Request PDU, mb_req_pdu
- MODBUS Response PDU, mb_rsp_pdu
- MODBUS Exception Response PDU, mb_excep_rsp_pdu

The mb_req_pdu is defined as:

mb_req_pdu = {function_code, request_data}, where
function_code = [1 byte] MODBUS function code,
request_data = [n bytes] This field is function code dependent and usually contains information such as variable references, variable counts, data offsets, sub-function codes etc.

The mb_rsp_pdu is defined as:
mb_rsp_pdu = {function_code, response_data}, where
function_code = [1 byte] MODBUS function code
response_data = [n bytes] This field is function code dependent and usually contains information such as variable references, variable counts, data offsets, sub-function codes, etc.

The mb_excep_rsp_pdu is defined as:
mb_excep_rsp_pdu = {exception-function_code, request_data}, where
exception-function_code = [1 byte] MODBUS function code + 0x80
exception_code = [1 byte] MODBUS Exception Code Defined in table "MODBUS Exception Codes" (see section 7).

4.2 Data Encoding

- MODBUS uses a 'big-Endian' representation for addresses and data items. This means that when a numerical quantity larger than a single byte is transmitted, the most significant byte is sent first. So for example

Register size	value			
16 - bits	0x1234	the first byte sent is	0x12	then 0x34

☞ **Note**: For more details, see [1] .

4.3 MODBUS Data model

MODBUS bases its data model on a series of tables that have distinguishing characteristics. The four primary tables are:

Primary tables	Object type	Type of	Comments
Discretes Input	Single bit	Read-Only	This type of data can be provided by an I/O system.
Coils	Single bit	Read-Write	This type of data can be alterable by an application program.
Input Registers	16-bit word	Read-Only	This type of data can be provided by an I/O system
Holding Registers	16-bit word	Read-Write	This type of data can be alterable by an application program.

The distinctions between inputs and outputs, and between bit-addressable and word-addressable data items, do not imply any application behavior. It is perfectly acceptable, and very common, to regard all four tables as overlaying one another, if this is the most natural interpretation on the target machine in question.

For each of the primary tables, the protocol allows individual selection of 65536 data items, and the operations of read or write of those items are designed to span multiple consecutive data items up to a data size limit which is dependent on the transaction function code.

It's obvious that all the data handled via MODBUS (bits, registers) must be located in device application memory. But physical address in memory should not be confused with data reference. The only requirement is to link data reference with physical address.

MODBUS logical reference numbers, which are used in MODBUS functions, are unsigned integer indices starting at zero.

- **Implementation examples of MODBUS model**

The examples below show two ways of organizing the data in device. There are different organizations possible, but not all are described in this document. Each device can have its own organization of the data according to its application

Example 1 : Device having 4 separate blocks

The example below shows data organization in a device having digital and analog, inputs and outputs. Each block is separate because data from different blocks have no correlation. Each block is thus accessible with different MODBUS functions.

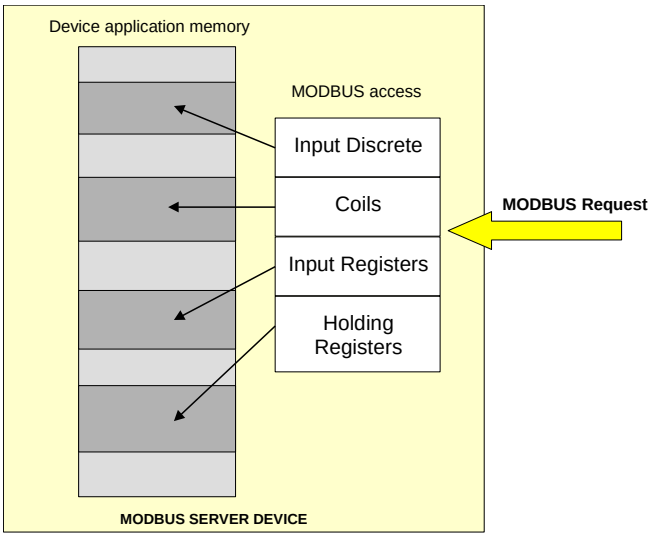

Figure 6 MODBUS Data Model with separate block

Example 2: Device having only 1 block

In this example, the device has only 1 data block. The same data can be reached via several MODBUS functions, either via a 16 bit access or via an access bit.

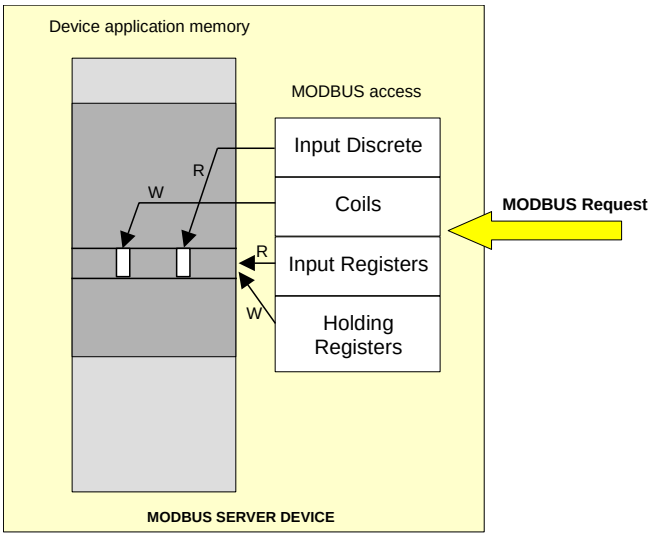

Figure 7 MODBUS Data Model with only 1 block

4.4 MODBUS Addressing model

The MODBUS application protocol defines precisely PDU addressing rules.

In a MODBUS PDU each data is addressed from 0 to 65535.

It also defines clearly a MODBUS data model composed of 4 blocks that comprises several elements numbered from 1 to n.

In the MODBUS data Model each element within a data block is numbered from 1 to n.

Afterwards the MODBUS data model has to be bound to the device application (IEC-61131 object, or other application model).

The pre-mapping between the MODBUS data model and the device application is totally vendor device specific.

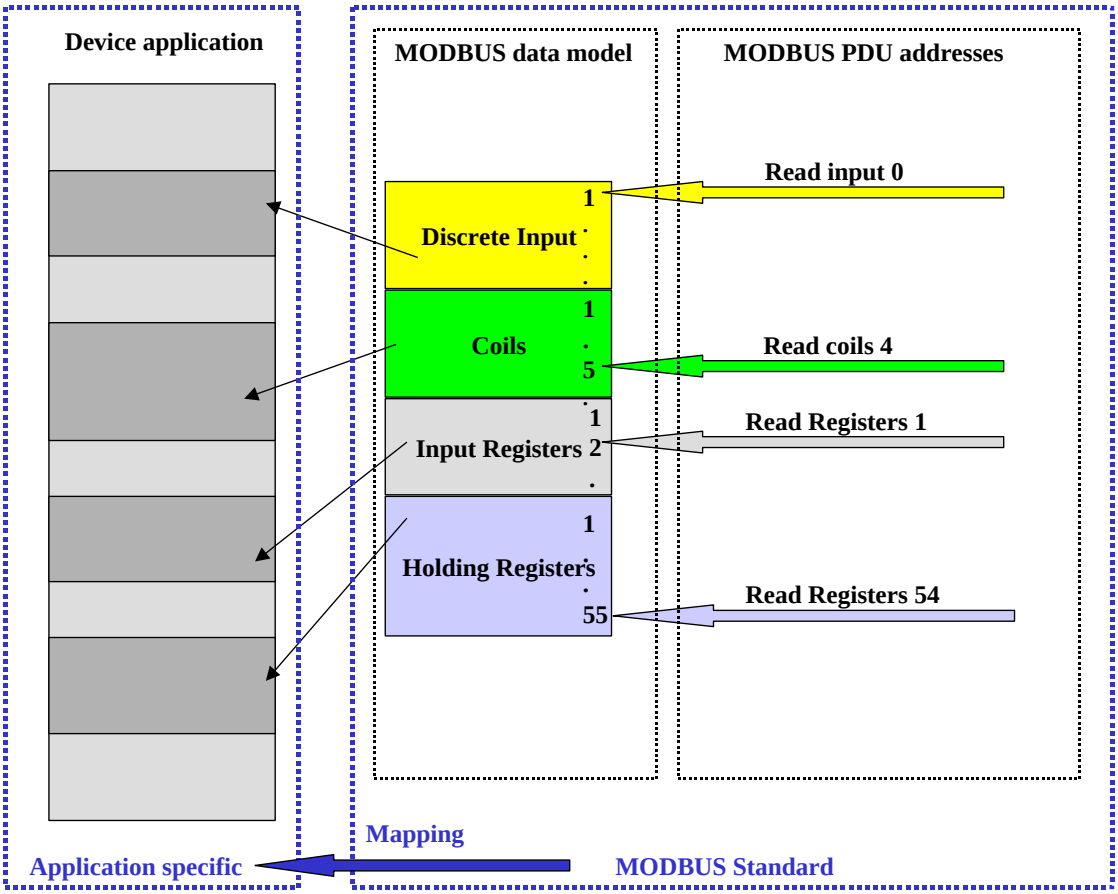

Figure 8 MODBUS Addressing model

The previous figure shows that a MODBUS data numbered X is addressed in the MODBUS PDU X-1.

4.5 Define MODBUS Transaction

The following state diagram describes the generic processing of a MODBUS transaction in server side.

V

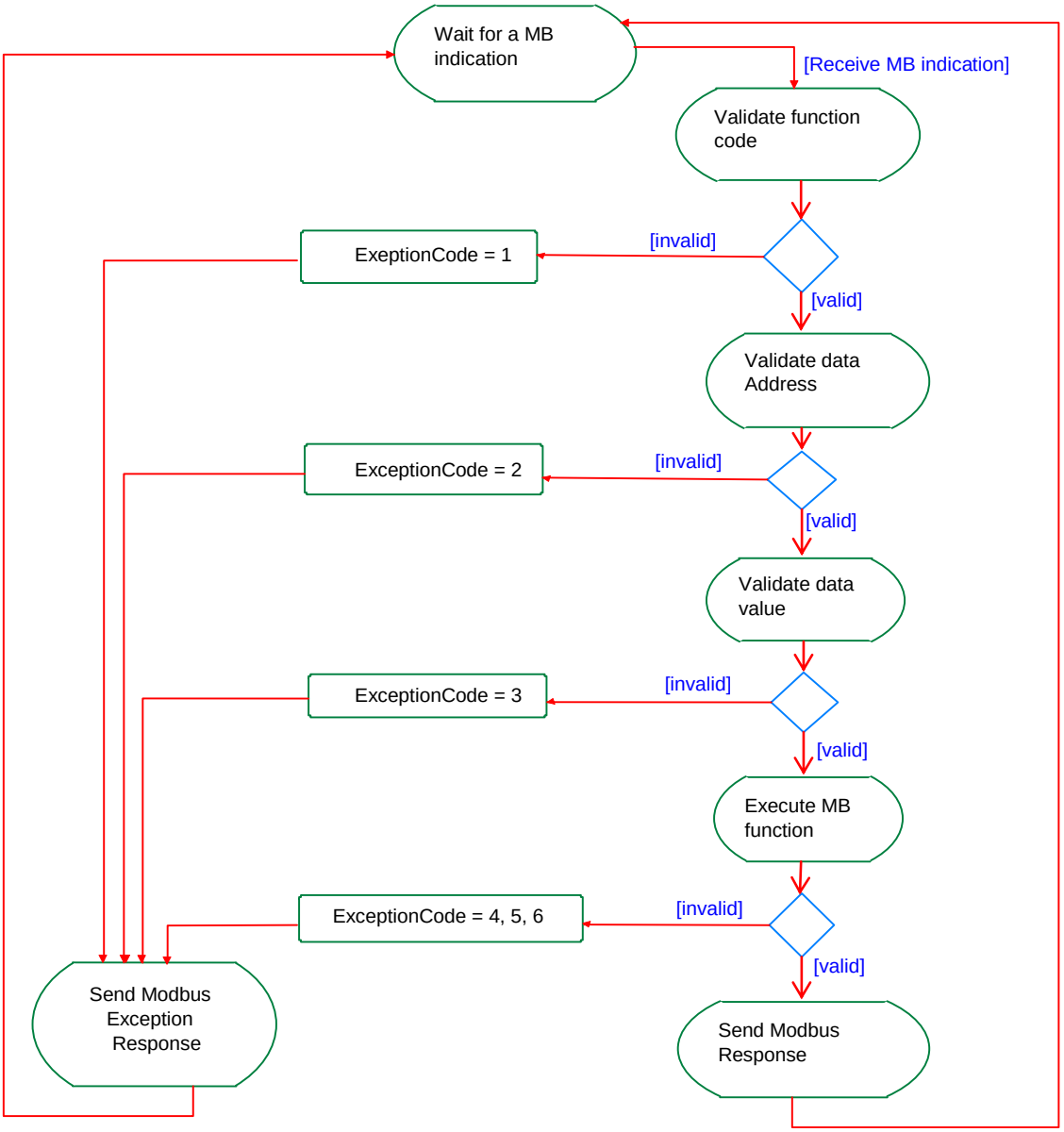

Figure 9 MODBUS Transaction state diagram

Once the request has been processed by a server, a MODBUS response using the adequate MODBUS server transaction is built.

Depending on the result of the processing two types of response are built :

- A positive MODBUS response :
  - the response function code = the request function code

- A MODBUS Exception response (see section 7):
  - the objective is to provide to the client relevant information concerning the error detected during the processing ;
  - the exception function code = the request function code + 0x80 ;
  - an exception code is provided to indicate the reason of the error.

5 Function Code Categories

There are three categories of MODBUS Functions codes. They are :

Public Function Codes

- Are well defined function codes ,
- guaranteed to be unique,
- validated by the MODBUS.org community,
- publicly documented
- have available conformance test,
- includes both defined public assigned function codes as well as unassigned function codes reserved for future use.

User-Defined Function Codes

- there are two ranges of user-defined function codes, i.e. 65 to 72 and from 100 to 110 decimal.
- user can select and implement a function code that is not supported by the specification.
- there is no guarantee that the use of the selected function code will be unique
- if the user wants to re-position the functionality as a public function code, he must initiate an RFC to introduce the change into the public category and to have a new public function code assigned.
- MODBUS Organization, Inc expressly reserves the right to develop the proposed RFC.

Reserved Function Codes

- Function Codes currently used by some companies for legacy products and that are not available for public use.
- Informative Note: The reader is asked refer to Annex A (Informative) MODBUS RESERVED FUNCTION CODES, SUBCODES AND MEI TYPES.

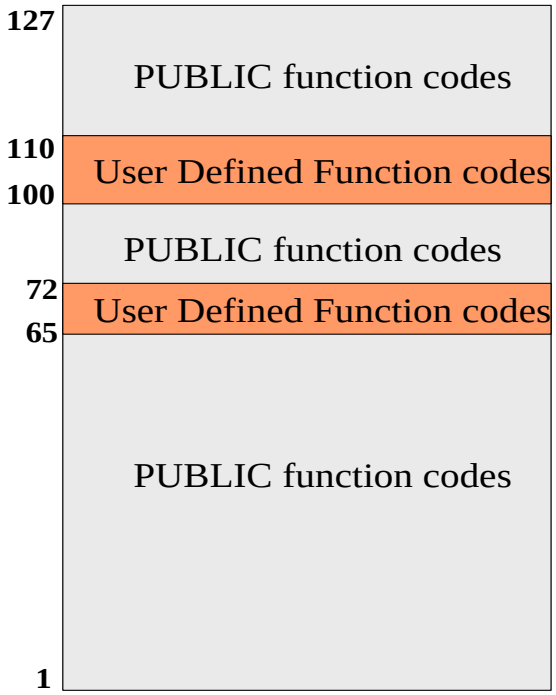

Figure 10 MODBUS Function Code Categories

5.1 Public Function Code Definition

				Function Codes			
				code	*Sub code*	*(hex)*	*Section*
Data Access	**Bit access**	Physical Discrete Inputs	Read Discrete Inputs	**02**		**02**	6.2
		Internal Bits Or Physical coils	Read Coils	**01**		**01**	6.1
			Write Single Coil	**05**		**05**	6.5
			Write Multiple Coils	**15**		**0F**	6.11
	16 bits access	Physical Input Registers	Read Input Register	**04**		**04**	6.4
		Internal Registers Or Physical Output Registers	Read Holding Registers	**03**		**03**	6.3
			Write Single Register	**06**		**06**	6.6
			Write Multiple Registers	**16**		**10**	6.12
			Read/Write Multiple Registers	**23**		**17**	6.17
			Mask Write Register	**22**		**16**	6.16
			Read FIFO queue	**24**		**18**	6.18
	File record access		Read File record	**20**		**14**	6.14
			Write File record	**21**		**15**	6.15
Diagnostics			Read Exception status	**07**		**07**	6.7
			Diagnostic	**08**	**00-18,20**	**08**	6.8
			Get Com event counter	**11**		**OB**	6.9
			Get Com Event Log	**12**		**0C**	6.10
			Report Server ID	**17**		**11**	6.13
			Read device Identification	**43**	**14**	**2B**	6.21
Other			Encapsulated Interface Transport	**43**	**13,14**	**2B**	6.19
			CANopen General Reference	**43**	**13**	**2B**	6.20

6 Function codes descriptions

6.1 01 (0x01) Read Coils

This function code is used to read from 1 to 2000 contiguous status of coils in a remote device. The Request PDU specifies the starting address, i.e. the address of the first coil specified, and the number of coils. In the PDU Coils are addressed starting at zero. Therefore coils numbered 1-16 are addressed as 0-15.

The coils in the response message are packed as one coil per bit of the data field. Status is indicated as 1= ON and 0= OFF. The LSB of the first data byte contains the output addressed in the query. The other coils follow toward the high order end of this byte, and from low order to high order in subsequent bytes.

If the returned output quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeros (toward the high order end of the byte). The Byte Count field specifies the quantity of complete bytes of 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** Byte	n = N or N+1

***N** = Quantity of Outputs / 8, if the remainder is different of 0 ⇒ N = N+1

Error

Function code	1 Byte	**Function code + 0x80**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of a request to read discrete outputs 20–38:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**01**	Function	**01**
Starting Address Hi	**00**	Byte Count	**03**
Starting Address Lo	**13**	Outputs status 27-20	**CD**
Quantity of Outputs Hi	**00**	Outputs status 35-28	**6B**
Quantity of Outputs Lo	**13**	Outputs status 38-36	**05**

The status of outputs 27–20 is shown as the byte value CD hex, or binary 1100 1101. Output 27 is the MSB of this byte, and output 20 is the LSB.

By convention, bits within a byte are shown with the MSB to the left, and the LSB to the right. Thus the outputs in the first byte are '27 through 20', from left to right. The next byte has outputs '35 through 28', left to right. As the bits are transmitted serially, they flow from LSB to MSB: 20 . . . 27, 28 . . . 35, and so on.

In the last data byte, the status of outputs 38-36 is shown as the byte value 05 hex, or binary 0000 0101. Output 38 is in the sixth bit position from the left, and output 36 is the LSB of this byte. The five remaining high order bits are zero filled.

☞ **Note**: The five remaining bits (toward the high order end) are zero filled.

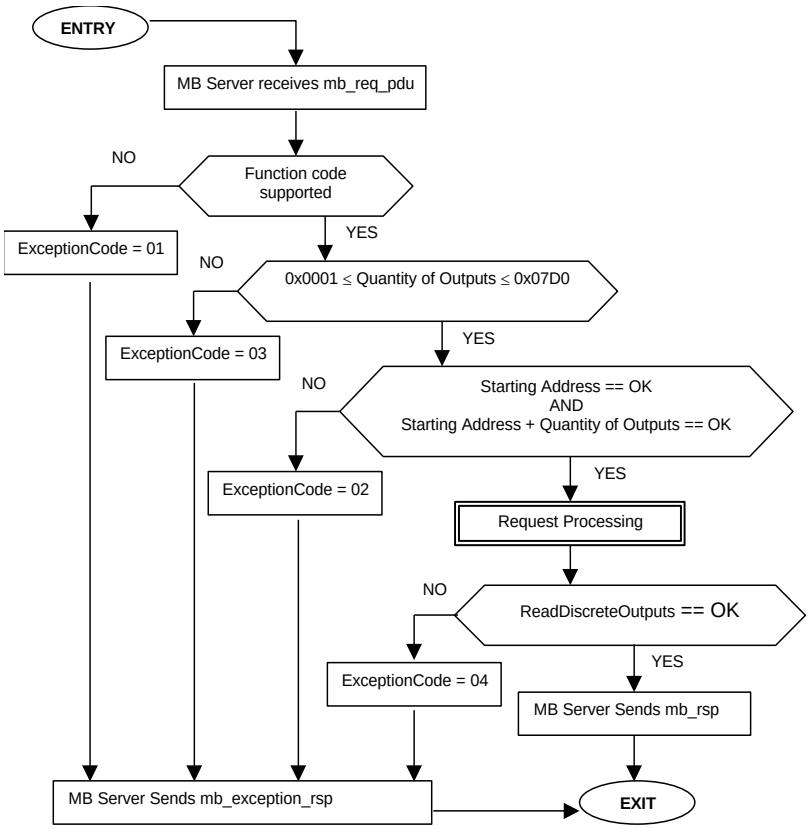

Figure 11: Read Coils state diagram

6.2 02 (0x02) Read Discrete Inputs

This function code is used to read from 1 to 2000 contiguous status of discrete inputs in a remote device. The Request PDU specifies the starting address, i.e. the address of the first

input specified, and the number of inputs. In the PDU Discrete Inputs are addressed starting at zero. Therefore Discrete inputs numbered 1-16 are addressed as 0-15.

The discrete inputs in the response message are packed as one input per bit of the data field. Status is indicated as 1= ON; 0= OFF. The LSB of the first data byte contains the input addressed in the query. The other inputs follow toward the high order end of this byte, and from low order to high order in subsequent bytes.

If the returned input quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeros (toward the high order end of the byte). The Byte Count field specifies the quantity of complete bytes of data.

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*** x 1 Byte	

***N** = Quantity of Inputs / 8 if the remainder is different of 0 ⇒ N = N+1

Error

Error code	1 Byte	**0x82**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of a request to read discrete inputs 197 – 218:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**02**	Function	**02**
Starting Address Hi	**00**	Byte Count	**03**
Starting Address Lo	**C4**	Inputs Status 204-197	**AC**
Quantity of Inputs Hi	**00**	Inputs Status 212-205	**DB**
Quantity of Inputs Lo	**16**	Inputs Status 218-213	**35**

The status of discrete inputs 204–197 is shown as the byte value AC hex, or binary 1010 1100. Input 204 is the MSB of this byte, and input 197 is the LSB.

The status of discrete inputs 218–213 is shown as the byte value 35 hex, or binary 0011 0101. Input 218 is in the third bit position from the left, and input 213 is the LSB.

☞ **Note**: The two remaining bits (toward the high order end) are zero filled.

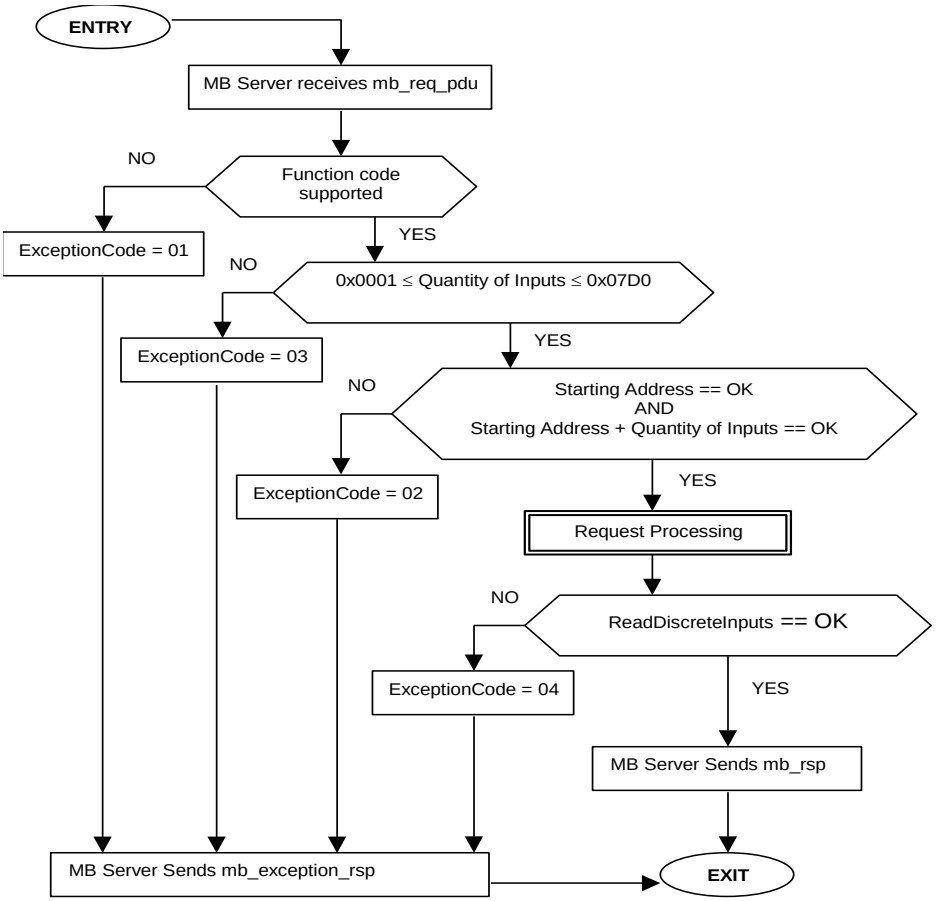

Figure 12: Read Discrete Inputs state diagram

6.3 03 (0x03) Read Holding Registers

This function code is used to read the contents of a contiguous block of holding registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore registers numbered 1-16 are addressed as 0-15.

The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register, the first byte contains the high order bits and the second contains the low order bits.

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	

***N** = Quantity of Registers

Error

Error code	1 Byte	**0x83**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of a request to read registers 108 – 110:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**03**	Function	**03**
Starting Address Hi	**00**	Byte Count	**06**
Starting Address Lo	**6B**	Register value Hi (108)	**02**
No. of Registers Hi	**00**	Register value Lo (108)	**2B**
No. of Registers Lo	**03**	Register value Hi (109)	**00**
		Register value Lo (109)	**00**
		Register value Hi (110)	**00**
		Register value Lo (110)	**64**

The contents of register 108 are shown as the two byte values of 02 2B hex, or 555 decimal. The contents of registers 109–110 are 00 00 and 00 64 hex, or 0 and 100 decimal, respectively.

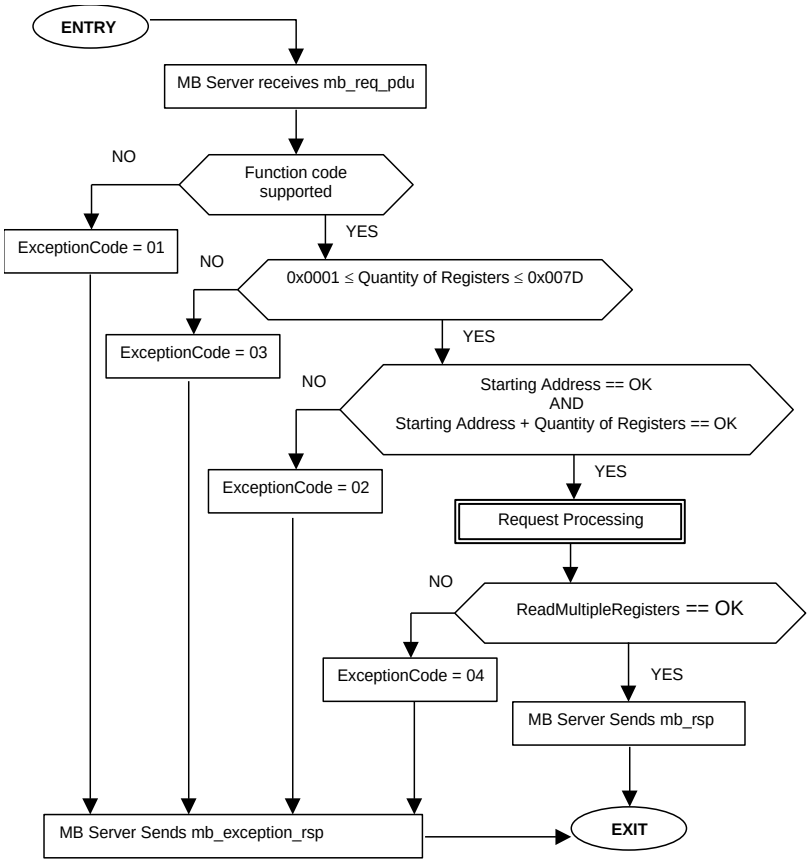

Figure 13: Read Holding Registers state diagram

6.4 04 (0x04) Read Input Registers

This function code is used to read from 1 to 125 contiguous input registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore input registers numbered 1-16 are addressed as 0-15.

The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register, the first byte contains the high order bits and the second contains the low order bits.

Request

Function code	1 Byte	**0x04**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Input Registers	2 Bytes	0x0001 to 0x007D

Response

Function code	1 Byte	**0x04**
Byte count	1 Byte	2 x **N***
Input Registers	**N*** x 2 Bytes	

***N** = Quantity of Input Registers

Error

Error code	1 Byte	**0x84**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of a request to read input register 9:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**04**	Function	**04**
Starting Address Hi	**00**	Byte Count	**02**
Starting Address Lo	**08**	Input Reg. 9 Hi	**00**
Quantity of Input Reg. Hi	**00**	Input Reg. 9 Lo	**0A**

Quantity of Input Reg. Lo	01	

The contents of input register 9 are shown as the two byte values of 00 0A hex, or 10 decimal.

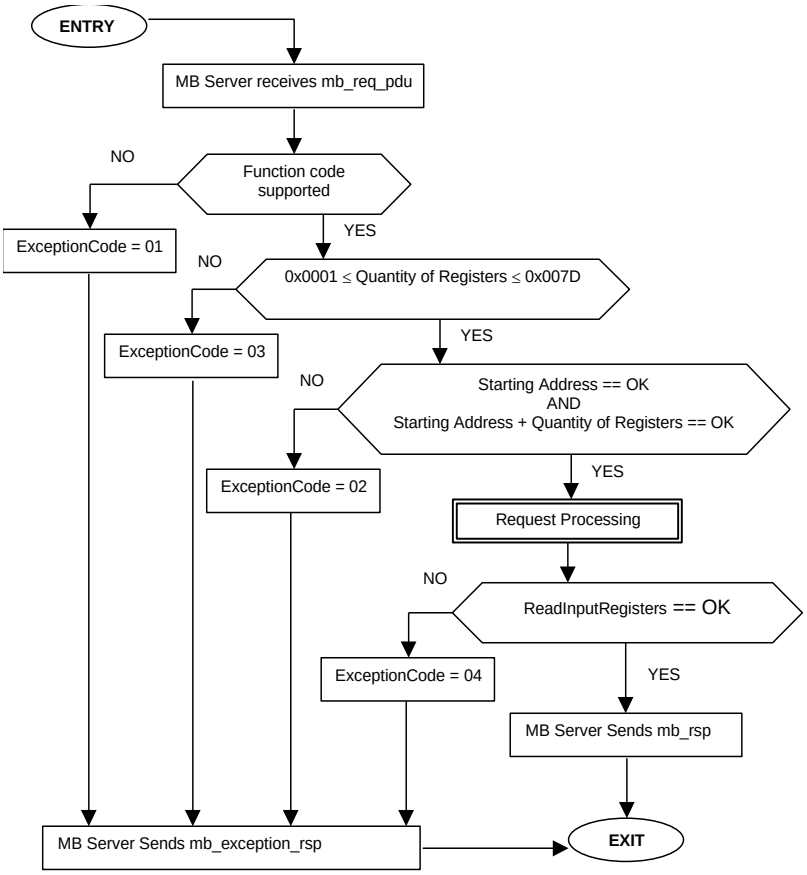

Figure 14: Read Input Registers state diagram

6.5 05 (0x05) Write Single Coil

This function code is used to write a single output to either ON or OFF in a remote device.

The requested ON/OFF state is specified by a constant in the request data field. A value of FF 00 hex requests the output to be ON. A value of 00 00 requests it to be OFF. All other values are illegal and will not affect the output.

The Request PDU specifies the address of the coil to be forced. Coils are addressed starting at zero. Therefore coil numbered 1 is addressed as 0. The requested ON/OFF state is specified by a constant in the Coil Value field. A value of 0XFF00 requests the coil to be ON. A value of 0X0000 requests the coil to be off. All other values are illegal and will not affect the coil.

The normal response is an echo of the request, returned after the coil state has been written.

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

Error code	1 Byte	**0x85**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of a request to write Coil 173 ON:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**05**	Function	**05**
Output Address Hi	**00**	Output Address Hi	**00**
Output Address Lo	**AC**	Output Address Lo	**AC**
Output Value Hi	**FF**	Output Value Hi	**FF**
Output Value Lo	**00**	Output Value Lo	**00**

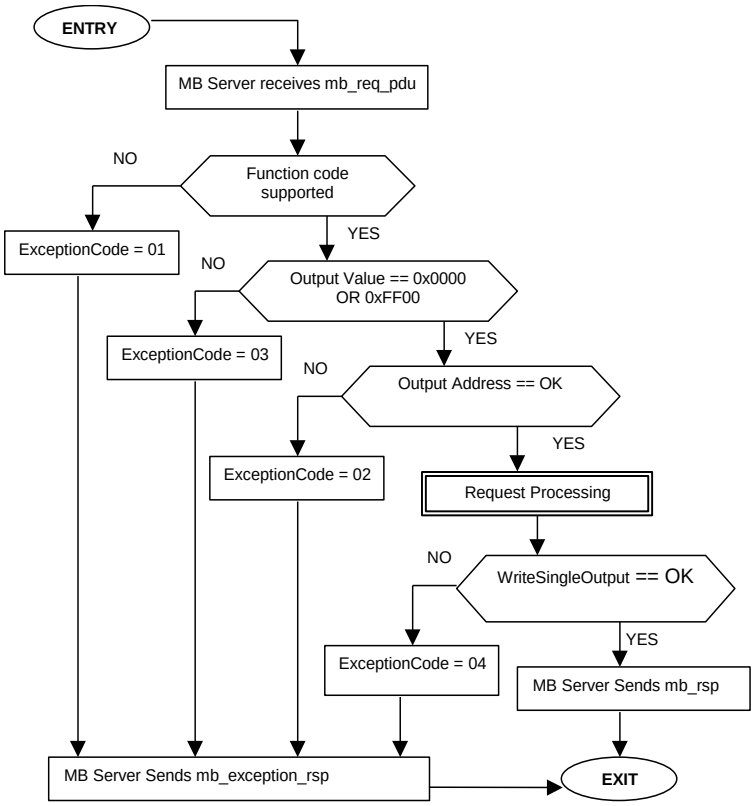

Figure 15: Write Single Output state diagram

6.6 06 (0x06) Write Single Register

This function code is used to write a single holding register in a remote device.

The Request PDU specifies the address of the register to be written. Registers are addressed starting at zero. Therefore register numbered 1 is addressed as 0.

The normal response is an echo of the request, returned after the register contents have been written.

Request

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

Response

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

Error

Error code	1 Byte	**0x86**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of a request to write register 2 to 00 03 hex:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**06**	Function	**06**
Register Address Hi	**00**	Register Address Hi	**00**
Register Address Lo	**01**	Register Address Lo	**01**
Register Value Hi	**00**	Register Value Hi	**00**
Register Value Lo	**03**	Register Value Lo	**03**

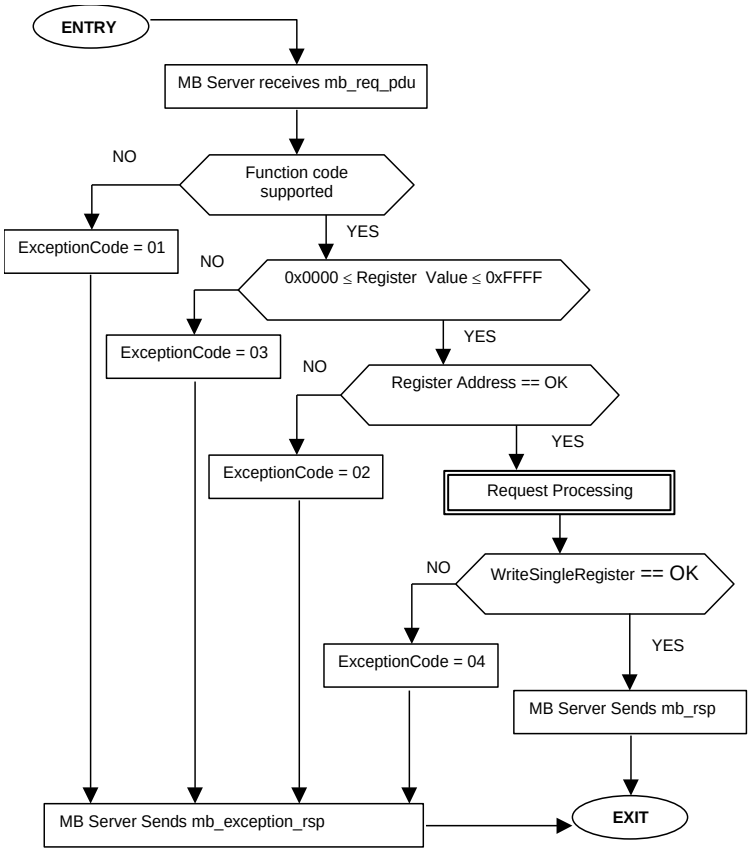

Figure 16: Write Single Register state diagram

6.7 07 (0x07) Read Exception Status (Serial Line only)

This function code is used to read the contents of eight Exception Status outputs in a remote device.

The function provides a simple method for accessing this information, because the Exception Output references are known (no output reference is needed in the function).

The normal response contains the status of the eight Exception Status outputs. The outputs are packed into one data byte, with one bit per output. The status of the lowest output reference is contained in the least significant bit of the byte.

The contents of the eight Exception Status outputs are device specific.

Request

Function code	1 Byte	**0x07**

Response

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

Error

Error code	1 Byte	**0x87**
Exception code	1 Byte	01 or 04

Here is an example of a request to read the exception status:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**07**	Function	**07**
		Output Data	**6D**

In this example, the output data is 6D hex (0110 1101 binary). Left to right, the outputs are OFF–ON–ON–OFF–ON–ON–OFF–ON. The status is shown from the highest to the lowest addressed output.

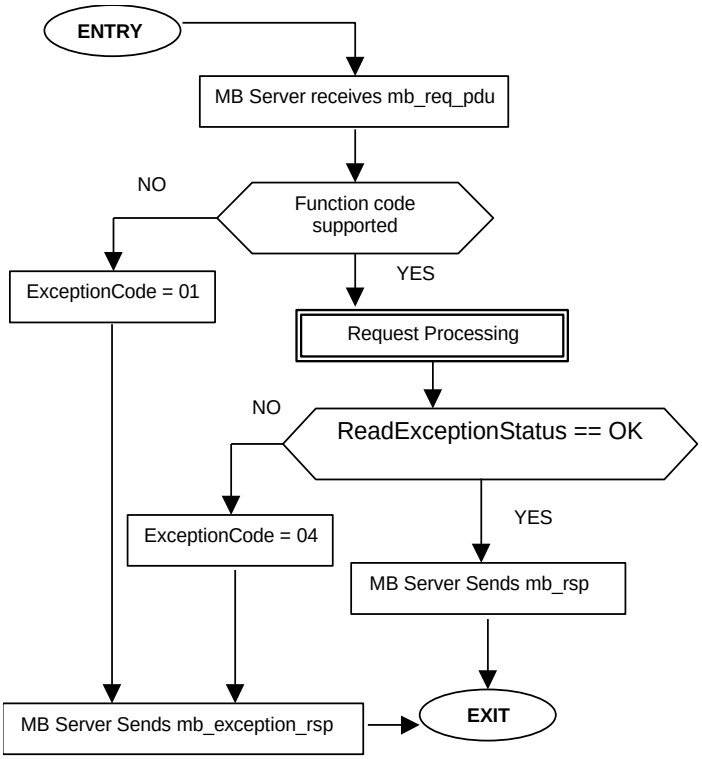

Figure 17: Read Exception Status state diagram

6.8 08 (0x08) Diagnostics (Serial Line only)

MODBUS function code 08 provides a series of tests for checking the communication system between a client device and a server, or for checking various internal error conditions within a server.

The function uses a two–byte sub-function code field in the query to define the type of test to be performed. The server echoes both the function code and sub-function code in a normal response. Some of the diagnostics cause data to be returned from the remote device in the data field of a normal response.

In general, issuing a diagnostic function to a remote device does not affect the running of the user program in the remote device. User logic, like discrete and registers, is not accessed by the diagnostics. Certain functions can optionally reset error counters in the remote device.

A server device can, however, be forced into 'Listen Only Mode' in which it will monitor the messages on the communications system but not respond to them. This can affect the outcome of your application program if it depends upon any further exchange of data with the remote device. Generally, the mode is forced to remove a malfunctioning remote device from the communications system.

The following diagnostic functions are dedicated to serial line devices.

The normal response to the Return Query Data request is to loopback the same data. The function code and sub-function codes are also echoed.

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

Error code	1 Byte	**0x88**
Exception code	1 Byte	01 or 03 or 04

6.8.1 Sub-function codes supported by the serial line devices

Here the list of sub-function codes supported by the serial line devices. Each sub-function code is then listed with an example of the data field contents that would apply for that diagnostic.

Sub-function code		Name
Hex	Dec	
00	00	Return Query Data
01	01	Restart Communications Option
02	02	Return Diagnostic Register
03	03	Change ASCII Input Delimiter
04	04	Force Listen Only Mode
	05.. 09	RESERVED
0A	10	Clear Counters and Diagnostic Register
0B	11	Return Bus Message Count
0C	12	Return Bus Communication Error Count
0D	13	Return Bus Exception Error Count
0E	14	Return Server Message Count
0F	15	Return Server No Response Count
10	16	Return Server NAK Count
11	17	Return Server Busy Count
12	18	Return Bus Character Overrun Count
13	19	RESERVED
14	20	Clear Overrun Counter and Flag
N.A.	21 … 65535	RESERVED

00 Return Query Data

The data passed in the request data field is to be returned (looped back) in the response. The entire response message should be identical to the request.

Sub-function	Data Field (Request)	Data Field (Response)
00 00	Any	Echo Request Data

01 Restart Communications Option

The remote device serial line port must be initialized and restarted, and all of its communications event counters are cleared. If the port is currently in Listen Only Mode, no response is returned. This function is the only one that brings the port out of Listen Only Mode. If the port is not currently in Listen Only Mode, a normal response is returned. This occurs before the restart is executed.

When the remote device receives the request, it attempts a restart and executes its power–up confidence tests. Successful completion of the tests will bring the port online.

A request data field contents of FF 00 hex causes the port's Communications Event Log to be cleared also. Contents of 00 00 leave the log as it was prior to the restart.

Sub-function	Data Field (Request)	Data Field (Response)
00 01	00 00	Echo Request Data
00 01	FF 00	Echo Request Data

02 Return Diagnostic Register

The contents of the remote device's 16–bit diagnostic register are returned in the response.

Sub-function	Data Field (Request)	Data Field (Response)
00 02	00 00	Diagnostic Register Contents

03 Change ASCII Input Delimiter

The character 'CHAR' passed in the request data field becomes the end of message delimiter for future messages (replacing the default LF character). This function is useful in cases of a Line Feed is not required at the end of ASCII messages.

Sub-function	Data Field (Request)	Data Field (Response)
00 03	CHAR 00	Echo Request Data

04 Force Listen Only Mode

Forces the addressed remote device to its Listen Only Mode for MODBUS communications. This isolates it from the other devices on the network, allowing them to continue communicating without interruption from the addressed remote device. No response is returned.

When the remote device enters its Listen Only Mode, all active communication controls are turned off. The Ready watchdog timer is allowed to expire, locking the controls off. While the device is in this mode, any MODBUS messages addressed to it or broadcast are monitored, but no actions will be taken and no responses will be sent.

The only function that will be processed after the mode is entered will be the Restart Communications Option function (function code 8, sub-function 1).

Sub-function	Data Field (Request)	Data Field (Response)
00 04	00 00	No Response Returned

10 (0A Hex) Clear Counters and Diagnostic Register

The goal is to clear all counters and the diagnostic register. Counters are also cleared upon power–up.

Sub-function	Data Field (Request)	Data Field (Response)
00 0A	00 00	Echo Request Data

11 (0B Hex) Return Bus Message Count

The response data field returns the quantity of messages that the remote device has detected on the communications system since its last restart, clear counters operation, or power–up.

Sub-function	Data Field (Request)	Data Field (Response)
00 0B	00 00	Total Message Count

12 (0C Hex) Return Bus Communication Error Count

The response data field returns the quantity of CRC errors encountered by the remote device since its last restart, clear counters operation, or power–up.

Sub-function	Data Field (Request)	Data Field (Response)
00 0C	00 00	CRC Error Count

13 (0D Hex) Return Bus Exception Error Count

The response data field returns the quantity of MODBUS exception responses returned by the remote device since its last restart, clear counters operation, or power–up.

Exception responses are described and listed in section 7 .

Sub-function	Data Field (Request)	Data Field (Response)
00 0D	00 00	Exception Error Count

14 (0E Hex) Return Server Message Count

The response data field returns the quantity of messages addressed to the remote device, or broadcast, that the remote device has processed since its last restart, clear counters operation, or power–up.

Sub-function	Data Field (Request)	Data Field (Response)
00 0E	00 00	Server Message Count

15 (0F Hex) Return Server No Response Count

The response data field returns the quantity of messages addressed to the remote device for which it has returned no response (neither a normal response nor an exception response), since its last restart, clear counters operation, or power–up.

Sub-function	Data Field (Request)	Data Field (Response)
00 0F	00 00	Server No Response Count

16 (10 Hex) Return Server NAK Count

The response data field returns the quantity of messages addressed to the remote device for which it returned a Negative Acknowledge (NAK) exception response, since its last restart, clear counters operation, or power–up. Exception responses are described and listed in section 7 .

Sub-function	Data Field (Request)	Data Field (Response)
00 10	00 00	Server NAK Count

17 (11 Hex) Return Server Busy Count

The response data field returns the quantity of messages addressed to the remote device for which it returned a Server Device Busy exception response, since its last restart, clear counters operation, or power–up.

Sub-function	Data Field (Request)	Data Field (Response)
00 11	00 00	Server Device Busy Count

18 (12 Hex) Return Bus Character Overrun Count

The response data field returns the quantity of messages addressed to the remote device that it could not handle due to a character overrun condition, since its last restart, clear counters operation, or power–up. A character overrun is caused by data characters arriving at the port faster than they can be stored, or by the loss of a character due to a hardware malfunction.

Sub-function	Data Field (Request)	Data Field (Response)
00 12	00 00	Server Character Overrun Count

20 (14 Hex) Clear Overrun Counter and Flag

Clears the overrun error counter and reset the error flag.

Sub-function	Data Field (Request)	Data Field (Response)
00 14	00 00	Echo Request Data

6.8.2 Example and state diagram

Here is an example of a request to remote device to Return Query Data. This uses a sub-function code of zero (00 00 hex in the two–byte field). The data to be returned is sent in the two–byte data field (A5 37 hex).

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**08**	Function	**08**
Sub-function Hi	**00**	Sub-function Hi	**00**
Sub-function Lo	**00**	Sub-function Lo	**00**
Data Hi	**A5**	Data Hi	**A5**
Data Lo	**37**	Data Lo	**37**

The data fields in responses to other kinds of queries could contain error counts or other data requested by the sub-function code.

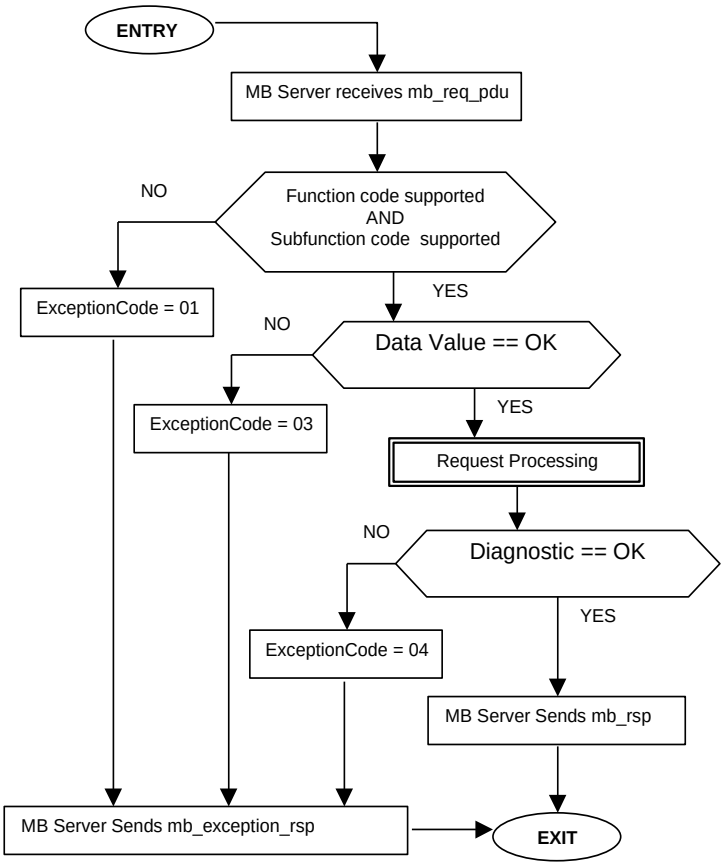

Figure 18: Diagnostic state diagram

6.9 11 (0x0B) Get Comm Event Counter (Serial Line only)

This function code is used to get a status word and an event count from the remote device's communication event counter.

By fetching the current count before and after a series of messages, a client can determine whether the messages were handled normally by the remote device.

The device's event counter is incremented once for each successful message completion. It is not incremented for exception responses, poll commands, or fetch event counter commands.

The event counter can be reset by means of the Diagnostics function (code 08), with a sub-function of Restart Communications Option (code 00 01) or Clear Counters and Diagnostic Register (code 00 0A).

The normal response contains a two–byte status word, and a two–byte event count. The status word will be all ones (FF FF hex) if a previously–issued program command is still being processed by the remote device (a busy condition exists). Otherwise, the status word will be all zeros.

Request

Function code	1 Byte	**0x0B**

Response

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

Error

Error code	1 Byte	**0x8B**
Exception code	1 Byte	01 or 04

Here is an example of a request to get the communications event counter in remote device:

Request	Response

Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**0B**	Function	**0B**
		Status Hi	**FF**
		Status Lo	**FF**
		Event Count Hi	**01**
		Event Count Lo	**08**

In this example, the status word is FF FF hex, indicating that a program function is still in progress in the remote device. The event count shows that 264 (01 08 hex) events have been counted by the device.

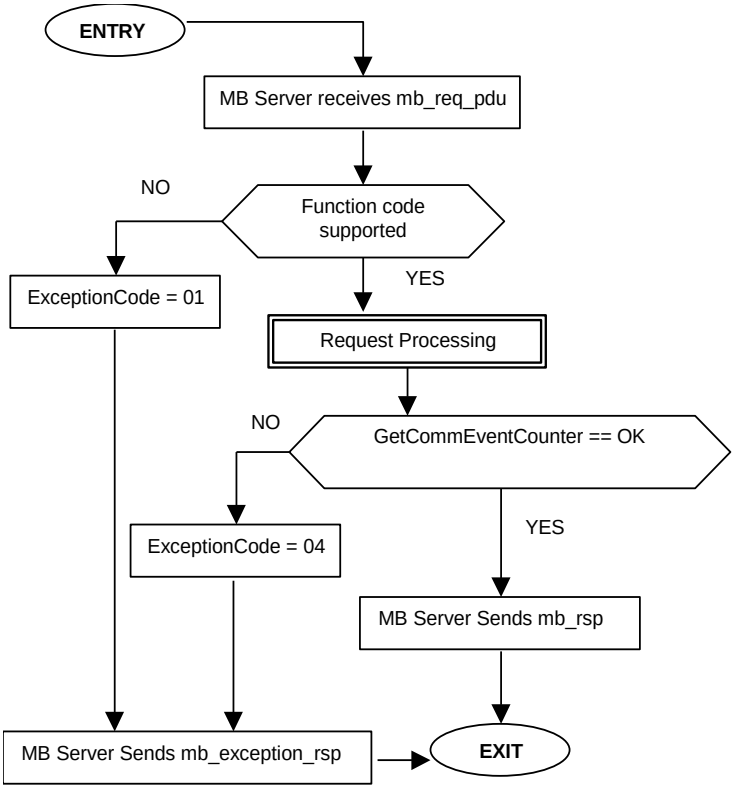

Figure 19: Get Comm Event Counter state diagram

6.10 12 (0x0C) Get Comm Event Log (Serial Line only)

This function code is used to get a status word, event count, message count, and a field of event bytes from the remote device.

The status word and event counts are identical to that returned by the Get Communications Event Counter function (11, 0B hex).

The message counter contains the quantity of messages processed by the remote device since its last restart, clear counters operation, or power–up. This count is identical to that returned by the Diagnostic function (code 08), sub-function Return Bus Message Count (code 11, 0B hex).

The event bytes field contains 0-64 bytes, with each byte corresponding to the status of one MODBUS send or receive operation for the remote device. The remote device enters the events into the field in chronological order. Byte 0 is the most recent event. Each new byte flushes the oldest byte from the field.

The normal response contains a two–byte status word field, a two–byte event count field, a two–byte message count field, and a field containing 0-64 bytes of events. A byte count field defines the total length of the data in these four fields.

Request

Function code	1 Byte	**0x0C**

Response

Function code	1 Byte	**0x0C**
Byte Count	1 Byte	**N***
Status	2 Bytes	0x0000 to 0xFFFF
Event Count	2 Bytes	0x0000 to 0xFFFF
Message Count	2 Bytes	0x0000 to 0xFFFF
Events	(**N**-6) x 1 Byte	

***N** = Quantity of Events + 3 x 2 Bytes, (Length of Status, Event Count and Message Count)

Error

Error code	1 Byte	**0x8C**
Exception code	1 Byte	01 or 04

Here is an example of a request to get the communications event log in remote device:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**0C**	Function	**0C**
		Byte Count	**08**
		Status Hi	**00**
		Status Lo	**00**
		Event Count Hi	**01**
		Event Count Lo	**08**
		Message Count Hi	**01**
		Message Count Lo	**21**
		Event 0	**20**
		Event 1	**00**

In this example, the status word is 00 00 hex, indicating that the remote device is not processing a program function. The event count shows that 264 (01 08 hex) events have been counted by the remote device. The message count shows that 289 (01 21 hex) messages have been processed.

The most recent communications event is shown in the Event 0 byte. Its content (20 hex) show that the remote device has most recently entered the Listen Only Mode.

The previous event is shown in the Event 1 byte. Its contents (00 hex) show that the remote device received a Communications Restart.

The layout of the response's event bytes is described below.

What the Event Bytes Contain

An event byte returned by the Get Communications Event Log function can be any one of four types. The type is defined by bit 7 (the high–order bit) in each byte. It may be further defined by bit 6. This is explained below.

- **Remote device MODBUS Receive Event**

  The remote device stores this type of event byte when a query message is received. It is stored before the remote device processes the message. This event is defined by bit 7 set to logic '1'. The other bits will be set to a logic '1' if the corresponding condition is TRUE. The bit layout is:

Bit	**Contents**
0	Not Used
1	Communication Error
2	Not Used
3	Not Used
4	Character Overrun

5	Currently in Listen Only Mode
6	Broadcast Received
7	1

- **Remote device MODBUS Send Event**

The remote device stores this type of event byte when it finishes processing a request message. It is stored if the remote device returned a normal or exception response, or no response. This event is defined by bit 7 set to a logic '0', with bit 6 set to a '1'. The other bits will be set to a logic '1' if the corresponding condition is TRUE. The bit layout is:

Bit	Contents
0	Read Exception Sent (Exception Codes 1-3)
1	Server Abort Exception Sent (Exception Code 4)
2	Server Busy Exception Sent (Exception Codes 5-6)
3	Server Program NAK Exception Sent (Exception Code 7)
4	Write Timeout Error Occurred
5	Currently in Listen Only Mode
6	1
7	0

- **Remote device Entered Listen Only Mode**

The remote device stores this type of event byte when it enters the Listen Only Mode. The event is defined by a content of 04 hex.

- **Remote device Initiated Communication Restart**

The remote device stores this type of event byte when its communications port is restarted. The remote device can be restarted by the Diagnostics function (code 08), with sub-function Restart Communications Option (code 00 01).

That function also places the remote device into a 'Continue on Error' or 'Stop on Error' mode. If the remote device is placed into 'Continue on Error' mode, the event byte is added to the existing event log. If the remote device is placed into 'Stop on Error' mode, the byte is added to the log and the rest of the log is cleared to zeros.

The event is defined by a content of zero.

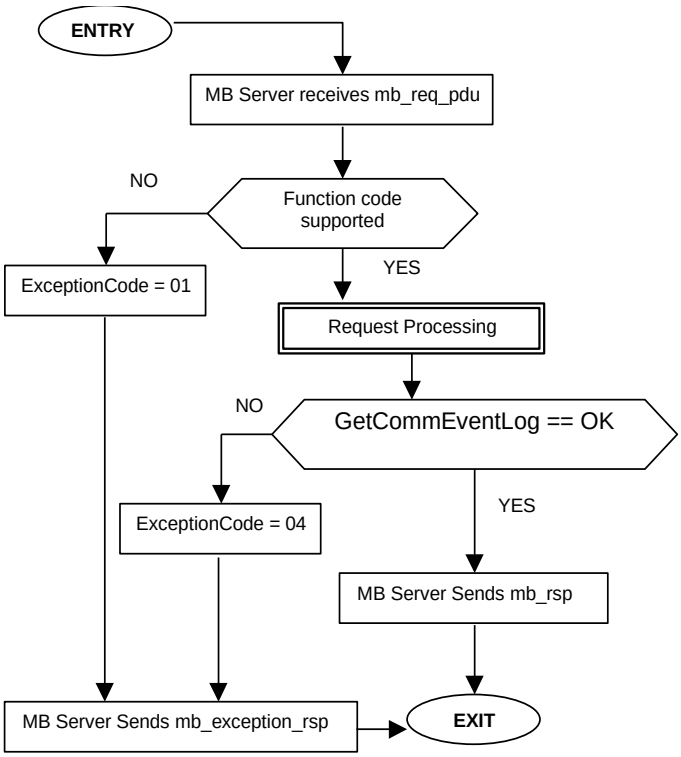

Figure 20: Get Comm Event Log state diagram

6.11 15 (0x0F) Write Multiple Coils

This function code is used to force each coil in a sequence of coils to either ON or OFF in a remote device. The Request PDU specifies the coil references to be forced. Coils are addressed starting at zero. Therefore coil numbered 1 is addressed as 0.

The requested ON/OFF states are specified by contents of the request data field. A logical '1' in a bit position of the field requests the corresponding output to be ON. A logical '0' requests it to be OFF.

The normal response returns the function code, starting address, and quantity of coils forced.

Request PDU

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***
Outputs Value	**N*** x 1 Byte	

***N** = Quantity of Outputs / 8, if the remainder is different of 0 ⇒ N = N+1

Response PDU

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

Error

Error code	1 Byte	**0x8F**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of a request to write a series of 10 coils starting at coil 20:

The request data contents are two bytes: CD 01 hex (1100 1101 0000 0001 binary). The binary bits correspond to the outputs in the following way:

Bit:	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	1
Output:	27	26	25	24	23	22	21	20	–	–	–	–	–	–	29	28

The first byte transmitted (CD hex) addresses outputs 27-20, with the least significant bit addressing the lowest output (20) in this set.

The next byte transmitted (01 hex) addresses outputs 29-28, with the least significant bit addressing the lowest output (28) in this set. Unused bits in the last data byte should be zero–filled.

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**0F**	Function	**0F**
Starting Address Hi	**00**	Starting Address Hi	**00**
Starting Address Lo	**13**	Starting Address Lo	**13**
Quantity of Outputs Hi	**00**	Quantity of Outputs Hi	**00**
Quantity of Outputs Lo	**0A**	Quantity of Outputs Lo	**0A**
Byte Count	**02**		
Outputs Value Hi	**CD**		
Outputs Value Lo	**01**		

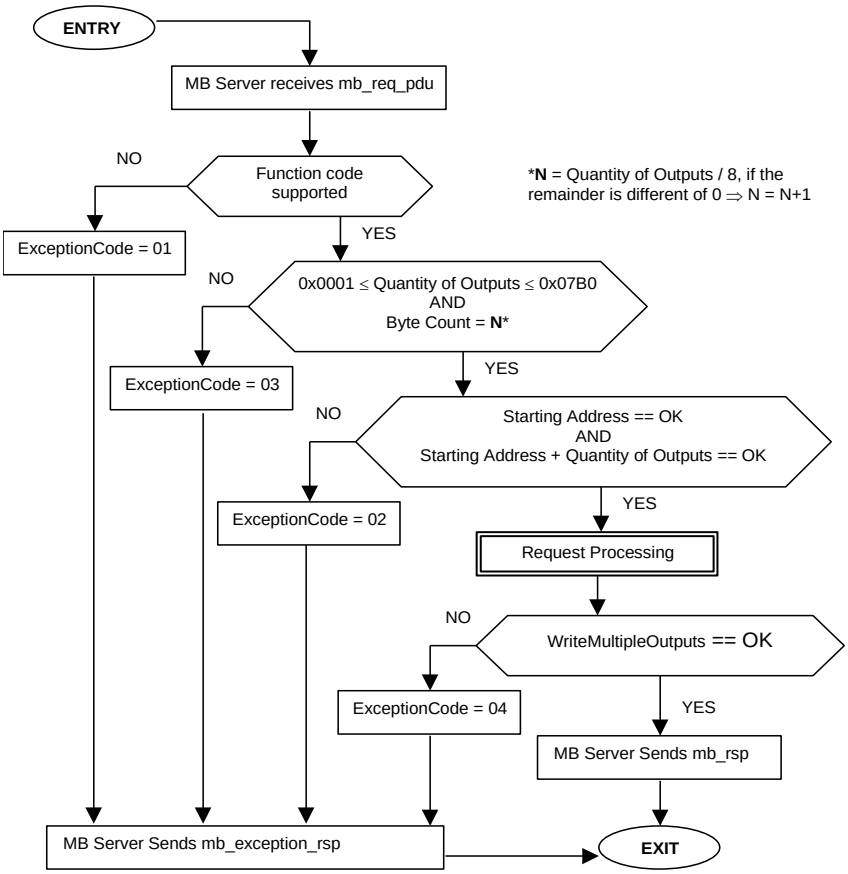

Figure 21: Write Multiple Outputs state diagram

6.12 16 (0x10) Write Multiple registers

This function code is used to write a block of contiguous registers (1 to 123 registers) in a remote device.

The requested written values are specified in the request data field. Data is packed as two bytes per register.

The normal response returns the function code, starting address, and quantity of registers written.

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***
Registers Value	**N*** x 2 Bytes	value

***N** = Quantity of Registers

Response

Function code	1 Byte	**0x10**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 123 (0x7B)

Error

Error code	1 Byte	**0x90**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of a request to write two registers starting at 2 to 00 0A and 01 02 hex:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**10**	Function	**10**
Starting Address Hi	**00**	Starting Address Hi	**00**
Starting Address Lo	**01**	Starting Address Lo	**01**

Quantity of Registers Hi	**00**	Quantity of Registers Hi	**00**
Quantity of Registers Lo	**02**	Quantity of Registers Lo	**02**
Byte Count	**04**		
Registers Value Hi	**00**		
Registers Value Lo	**0A**		
Registers Value Hi	**01**		
Registers Value Lo	**02**		

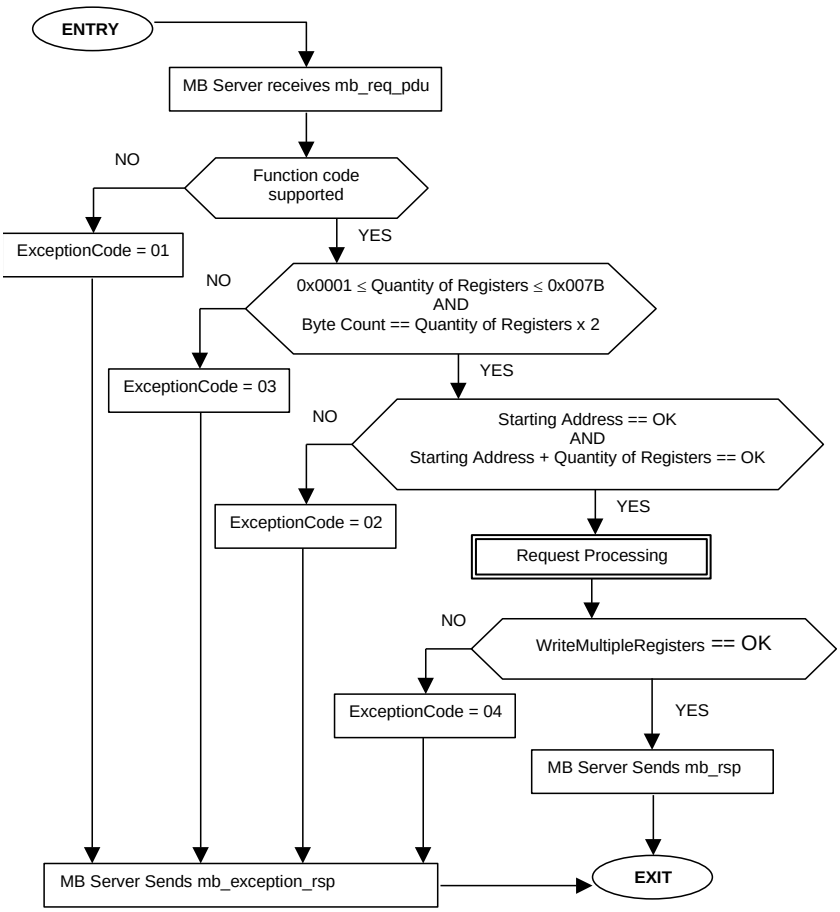

Figure 22: Write Multiple Registers state diagram

6.13 17 (0x11) Report Server ID (Serial Line only)

This function code is used to read the description of the type, the current status, and other information specific to a remote device.

The format of a normal response is shown in the following example. The data contents are specific to each type of device.

Request

Function code	1 Byte	**0x11**

Response

Function code	1 Byte	**0x11**
Byte Count	1 Byte	
Server ID	*device specific*	
Run Indicator Status	1 Byte	0x00 = OFF, 0xFF = ON
Additional Data		

Error

Error code	1 Byte	**0x91**
Exception code	1 Byte	01 or 04

Here is an example of a request to report the ID and status:

Request		Response	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**11**	Function	**11**
		Byte Count	Device Specific
		Server ID	Device Specific
		Run Indicator Status	0x00 or 0xFF
		Additional Data	Device Specific

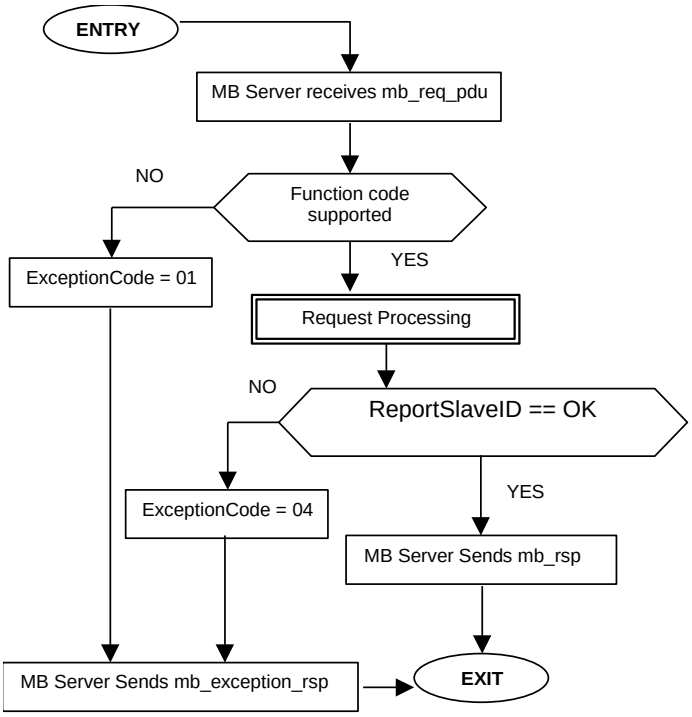

Figure 23: Report server ID state diagram

6.14 20 (0x14) Read File Record

This function code is used to perform a file record read. All Request Data Lengths are provided in terms of number of bytes and all Record Lengths are provided in terms of registers.

A file is an organization of records. Each file contains 10000 records, addressed 0000 to 9999 decimal or 0X0000 to 0X270F. For example, record 12 is addressed as 12.

The function can read multiple groups of references. The groups can be separating (non-contiguous), but the references within each group must be sequential.

Each group is defined in a separate 'sub-request' field that contains 7 bytes:

The reference type: 1 byte (must be specified as 6)

The File number: 2 bytes

The starting record number within the file: 2 bytes

The length of the record to be read: 2 bytes.

The quantity of registers to be read, combined with all other fields in the expected response, must not exceed the allowable length of the MODBUS PDU : 253 bytes.

The normal response is a series of 'sub-responses', one for each 'sub-request'. The byte count field is the total combined count of bytes in all 'sub-responses'. In addition, each 'sub-response' contains a field that shows its own byte count.

Request

Function code	1 Byte	**0x14**
Byte Count	1 Byte	0x07 to 0xF5 bytes
Sub-Req. x, Reference Type	1 Byte	06
Sub-Req. x, File Number	2 Bytes	0x0001 to 0xFFFF

Sub-Req. x, Record Number	2 Bytes	0x0000 to 0x270F
Sub-Req. x, Record Length	2 Bytes	**N**
Sub-Req. x+1, ...		

Response

Function code	1 Byte	**0x14**
Resp. data Length	1 Byte	0x07 to 0xF5
Sub-Req. x, File Resp. length	1 Byte	0x07 to 0xF5
Sub-Req. x, Reference Type	1 Byte	6
Sub-Req. x, Record Data	**N** x 2 Bytes	
Sub-Req. x+1, ...		

Error

Error code	1 Byte	**0x94**
Exception code	1 Byte	01 or 02 or 03 or 04 or 08

While it is allowed for the File Number to be in the range 1 to 0xFFFF, it should be noted that interoperability with legacy equipment may be compromised if the File Number is greater than 10 (0x0A).

Here is an example of a request to read two groups of references from remote device:

- Group 1 consists of two registers from file 4, starting at register 1 (address 0001).
- Group 2 consists of two registers from file 3, starting at register 9 (address 0009).

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**14**	Function	**14**
Byte Count	**0E**	Resp. Data length	**0C**
Sub-Req. 1, Ref. Type	**06**	Sub-Req. 1, File resp. length	**05**
Sub-Req. 1, File Number Hi	**00**	Sub-Req. 1, Ref. Type	**06**
Sub-Req. 1, File Number Lo	**04**	Sub-Req. 1, Register.Data Hi	**0D**
Sub-Req. 1, Record number Hi	**00**	Sub-Req. 1, Register.DataLo	**FE**
Sub-Req. 1, Record number Lo	**01**	Sub-Req. 1, Register.Data Hi	**00**
Sub-Req. 1, Record Length Hi	**00**	Sub-Req. 1, Register.DataLo	**20**
Sub-Req. 1, Record Length Lo	**02**	Sub-Req. 2, File resp. length	**05**
Sub-Req. 2, Ref. Type	**06**	Sub-Req. 2, Ref. Type	**06**
Sub-Req. 2, File Number Hi	**00**	Sub-Req. 2, Register.Data H	**33**
Sub-Req. 2, File Number Lo	**03**	Sub-Req. 2, Register.DataLo	**CD**
Sub-Req. 2, Record number Hi	**00**	Sub-Req. 2, Register.Data Hi	**00**
Sub-Req. 2, Record number Lo	**09**	Sub-Req. 2, Register.DataLo	**40**
Sub-Req. 2, Record Length Hi	**00**		
Sub-Req. 2, Record Length Lo	**02**		

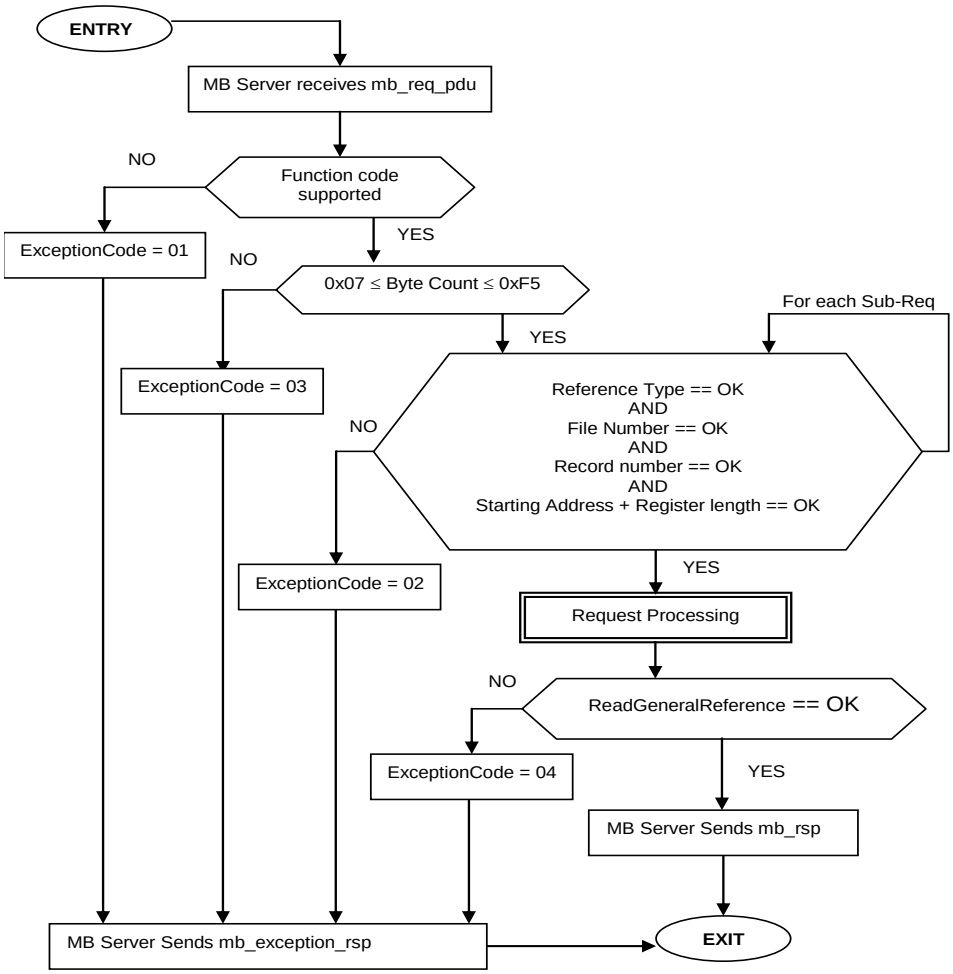

Figure 24: Read File Record state diagram

6.15 21 (0x15) Write File Record

This function code is used to perform a file record write. All Request Data Lengths are provided in terms of number of bytes and all Record Lengths are provided in terms of the number of 16-bit words.

A file is an organization of records. Each file contains 10000 records, addressed 0000 to 9999 decimal or 0X0000 to 0X270F. For example, record 12 is addressed as 12.

The function can write multiple groups of references. The groups can be separate, i.e. non–contiguous, but the references within each group must be sequential.

Each group is defined in a separate 'sub-request' field that contains 7 bytes plus the data:

The reference type: 1 byte (must be specified as 6)

The file number: 2 bytes

The starting record number within the file: 2 bytes

The length of the record to be written: 2 bytes

The data to be written: 2 bytes per register.

The quantity of registers to be written, combined with all other fields in the request, must not exceed the allowable length of the MODBUS PDU : 253bytes.

The normal response is an echo of the request.

Request

Function code	1 Byte	**0x15**
Request data length	1 Byte	0x09 to 0xFB
Sub-Req. x, Reference Type	1 Byte	06
Sub-Req. x, File Number	2 Bytes	0x0001 to 0xFFFF
Sub-Req. x, Record Number	2 Bytes	0x0000 to 0x270F

Sub-Req. x, Record length	2 Bytes	**N**
Sub-Req. x, Record data	**N** x 2 Bytes	
Sub-Req. x+1, ...		

Response

Function code	1 Byte	**0x15**
Response Data length	1 Byte	0x09 to 0xFB
Sub-Req. x, Reference Type	1 Byte	06
Sub-Req. x, File Number	2 Bytes	0x0001 to 0xFFFF
Sub-Req. x, Record number	2 Bytes	0x0000 to 0x270F
Sub-Req. x, Record length	2 Bytes	**N**
Sub-Req. x, Record Data	**N** x 2 Bytes	
Sub-Req. x+1, ...		

Error

Error code	1 Byte	**0x95**
Exception code	1 Byte	01 or 02 or 03 or 04 or 08

While it is allowed for the File Number to be in the range 1 to 0xFFFF, it should be noted that interoperability with legacy equipment may be compromised if the File Number is greater than 10 (0x0A).

Here is an example of a request to write one group of references into remote device:

- The group consists of three registers in file 4, starting at register 7 (address 0007).

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**15**	Function	**15**
Request Data length	**0D**	Request Data length	**0D**
Sub-Req. 1, Ref. Type	**06**	Sub-Req. 1, Ref. Type	**06**
Sub-Req. 1, File Number Hi	**00**	Sub-Req. 1, File Number Hi	**00**
Sub-Req. 1, File Number Lo	**04**	Sub-Req. 1, File Number Lo	**04**
Sub-Req. 1, Record number Hi	**00**	Sub-Req. 1, Record number Hi	**00**
Sub-Req. 1, Record number Lo	**07**	Sub-Req. 1, Record number Lo	**07**
Sub-Req. 1, Record length Hi	**00**	Sub-Req. 1, Record length Hi	**00**
Sub-Req. 1, Record length Lo	**03**	Sub-Req. 1, Record length Lo	**03**
Sub-Req. 1, Register Data Hi	**06**	Sub-Req. 1, Register Data Hi	**06**
Sub-Req. 1, Register Data Lo	**AF**	Sub-Req. 1, Register Data Lo	**AF**
Sub-Req. 1, Register Data Hi	**04**	Sub-Req. 1, Register Data Hi	**04**
Sub-Req. 1, Register Data Lo	**BE**	Sub-Req. 1, Register Data Lo	**BE**
Sub-Req. 1, Register Data Hi	**10**	Sub-Req. 1, Register Data Hi	**10**
Sub-Req. 1, Register Data Lo	**0D**	Sub-Req. 1, Register Data Lo	**0D**

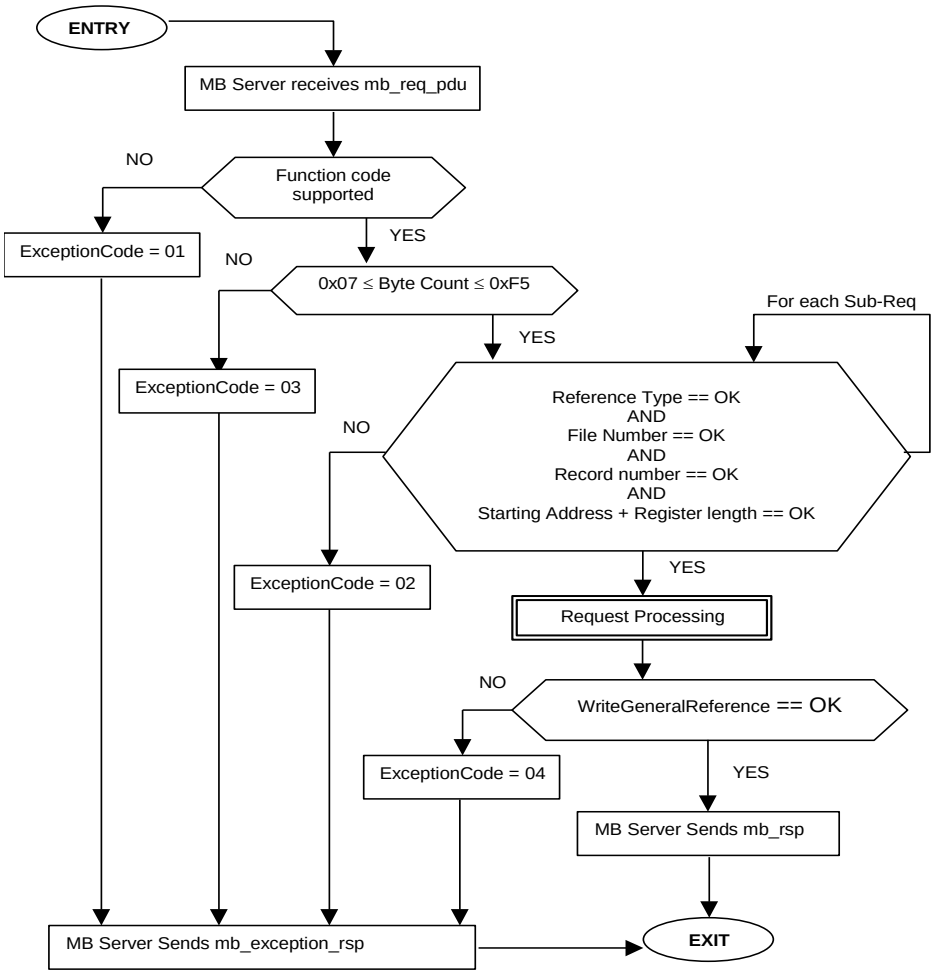

Figure 25: Write File Record state diagram

6.16 22 (0x16) Mask Write Register

This function code is used to modify the contents of a specified holding register using a combination of an AND mask, an OR mask, and the register's current contents. The function can be used to set or clear individual bits in the register.

The request specifies the holding register to be written, the data to be used as the AND mask, and the data to be used as the OR mask. Registers are addressed starting at zero. Therefore registers 1-16 are addressed as 0-15.

The function's algorithm is:

Result = (Current Contents AND And_Mask) OR (Or_Mask AND (NOT And_Mask))

For example:

	Hex	Binary
Current Contents=	12	0001 0010
And_Mask =	F2	1111 0010
Or_Mask =	25	0010 0101
(NOT And_Mask)=	0D	0000 1101
Result =	17	0001 0111

☞ **Note**:

- If the Or_Mask value is zero, the result is simply the logical ANDing of the current contents and And_Mask. If the And_Mask value is zero, the result is equal to the Or_Mask value.
- The contents of the register can be read with the Read Holding Registers function (function code 03). They could, however, be changed subsequently as the controller scans its user logic program.

The normal response is an echo of the request. The response is returned after the register has been written.

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

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

Error code	1 Byte	**0x96**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of a Mask Write to register 5 in remote device, using the above mask values.

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**16**	Function	**16**
Reference address Hi	**00**	Reference address Hi	**00**
Reference address Lo	**04**	Reference address Lo	**04**
And_Mask Hi	**00**	And_Mask Hi	**00**
And_Mask Lo	**F2**	And_Mask Lo	**F2**
Or_Mask Hi	**00**	Or_Mask Hi	**00**
Or_Mask Lo	**25**	Or_Mask Lo	**25**

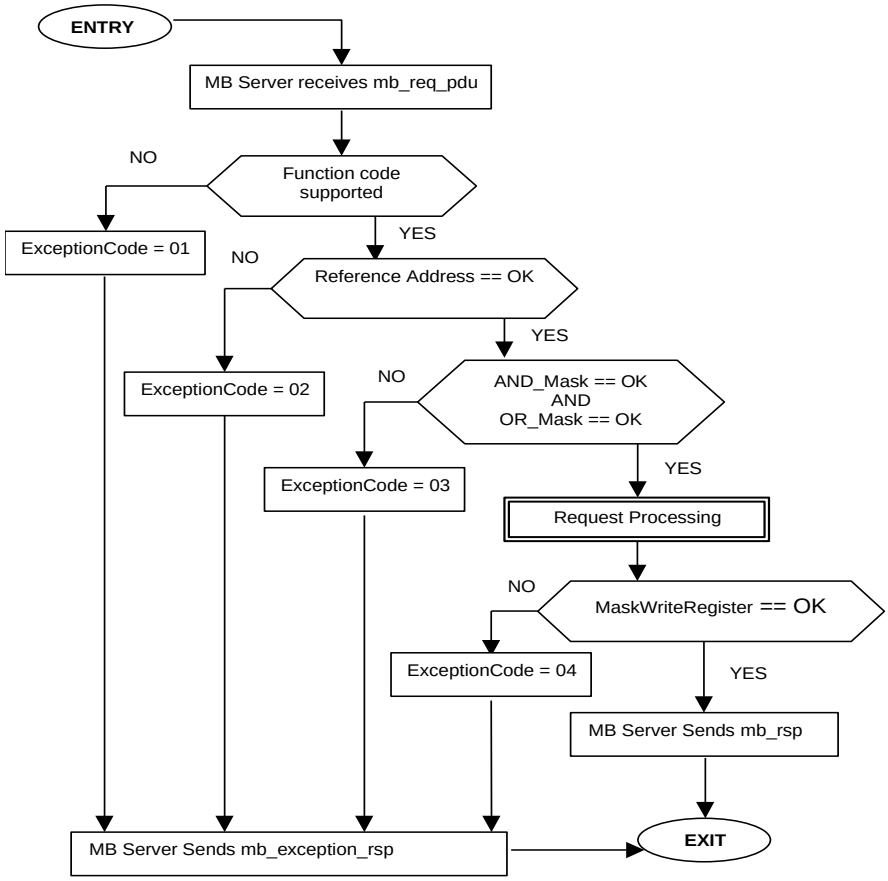

Figure 26: Mask Write Holding Register state diagram

6.17 23 (0x17) Read/Write Multiple registers

This function code performs a combination of one read operation and one write operation in a single MODBUS transaction. The write operation is performed before the read.

Holding registers are addressed starting at zero. Therefore holding registers 1-16 are addressed in the PDU as 0-15.

The request specifies the starting address and number of holding registers to be read as well as the starting address, number of holding registers, and the data to be written. The byte count specifies the number of bytes to follow in the write data field.

The normal response contains the data from the group of registers that were read. The byte count field specifies the quantity of bytes to follow in the read data field.

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	0x0000 to 0xFFFF
Quantity to Write	2 Bytes	0x0001 to 0X0079
Write Byte Count	1 Byte	2 x **N***
Write Registers Value	**N***x 2 Bytes	

***N** = Quantity to Write

Response

Function code	1 Byte	**0x17**
Byte Count	1 Byte	2 x **N'***
Read Registers value	**N'*** x 2 Bytes	

***N'** = Quantity to Read

Error

Error code	1 Byte	**0x97**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of a request to read six registers starting at register 4, and to write three registers starting at register 15:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**17**	Function	**17**
Read Starting Address Hi	**00**	Byte Count	**0C**
Read Starting Address Lo	**03**	Read Registers value Hi	**00**
Quantity to Read Hi	**00**	Read Registers value Lo	**FE**
Quantity to Read Lo	**06**	Read Registers value Hi	**0A**
Write Starting Address Hi	**00**	Read Registers value Lo	**CD**
Write Starting address Lo	**0E**	Read Registers value Hi	**00**
Quantity to Write Hi	**00**	Read Registers value Lo	**01**
Quantity to Write Lo	**03**	Read Registers value Hi	**00**
Write Byte Count	**06**	Read Registers value Lo	**03**
Write Registers Value Hi	**00**	Read Registers value Hi	**00**
Write Registers Value Lo	**FF**	Read Registers value Lo	**0D**
Write Registers Value Hi	**00**	Read Registers value Hi	**00**
Write Registers Value Lo	**FF**	Read Registers value Lo	**FF**
Write Registers Value Hi	**00**		
Write Registers Value Lo	**FF**		

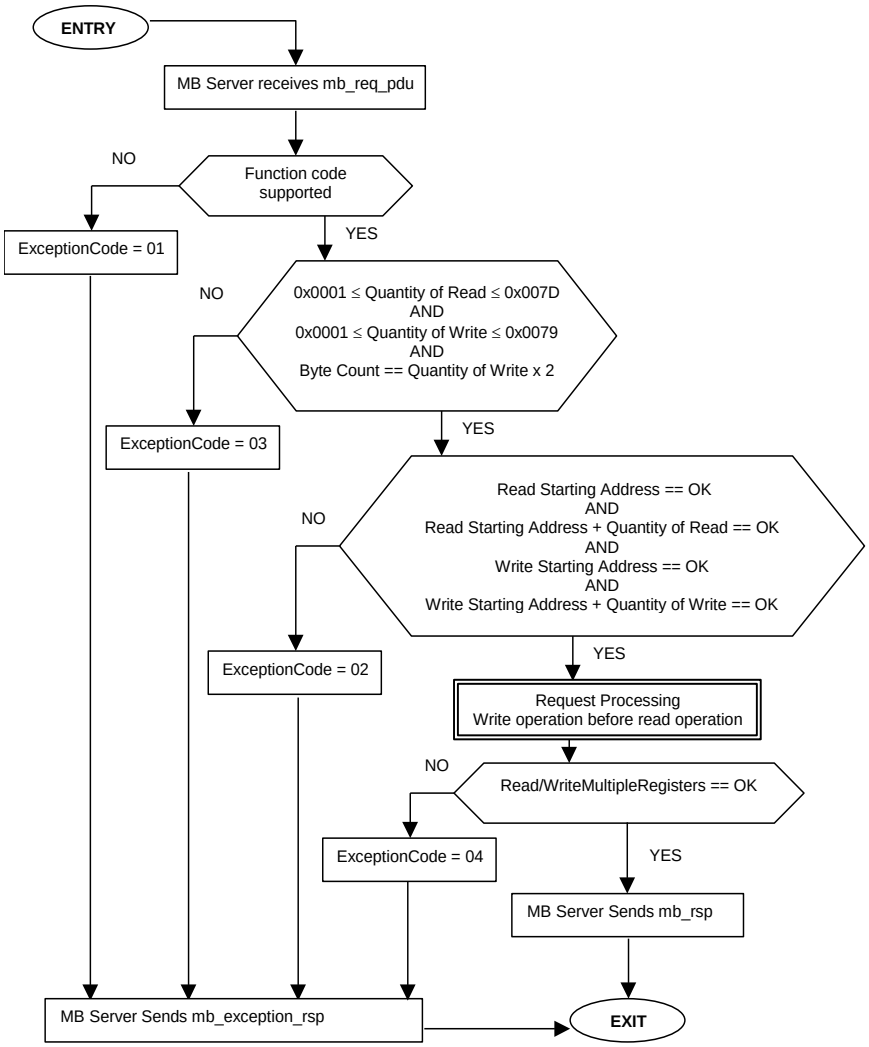

Figure 27: Read/Write Multiple Registers state diagram

6.18 24 (0x18) Read FIFO Queue

This function code allows to read the contents of a First-In-First-Out (FIFO) queue of register in a remote device. The function returns a count of the registers in the queue, followed by the queued data. Up to 32 registers can be read: the count, plus up to 31 queued data registers. The queue count register is returned first, followed by the queued data registers.

The function reads the queue contents, but does not clear them.

In a normal response, the byte count shows the quantity of bytes to follow, including the queue count bytes and value register bytes (but not including the error check field).

The queue count is the quantity of data registers in the queue (not including the count register).

If the queue count exceeds 31, an exception response is returned with an error code of 03 (Illegal Data Value).

Request

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

Response

Function code	1 Byte	**0x18**
Byte Count	2 Bytes	
FIFO Count	2 Bytes	≤ 31
FIFO Value Register	**N*** x 2 Bytes	

***N** = FIFO Count

Error

Error code	1 Byte	**0x98**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of Read FIFO Queue request to remote device. The request is to read the queue starting at the pointer register 1246 (0x04DE):

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**18**	Function	**18**
FIFO Pointer Address Hi	**04**	Byte Count Hi	**00**
FIFO Pointer Address Lo	**DE**	Byte Count Lo	**06**
		FIFO Count Hi	**00**
		FIFO Count Lo	**02**
		FIFO Value Register Hi	**01**
		FIFO Value Register Lo	**B8**
		FIFO Value Register Hi	**12**
		FIFO Value Register Lo	**84**

In this example, the FIFO pointer register (1246 in the request) is returned with a queue count of 2. The two data registers follow the queue count. These are:

1247 (contents 440 decimal -- 0x01B8); and 1248 (contents 4740 -- 0x1284).

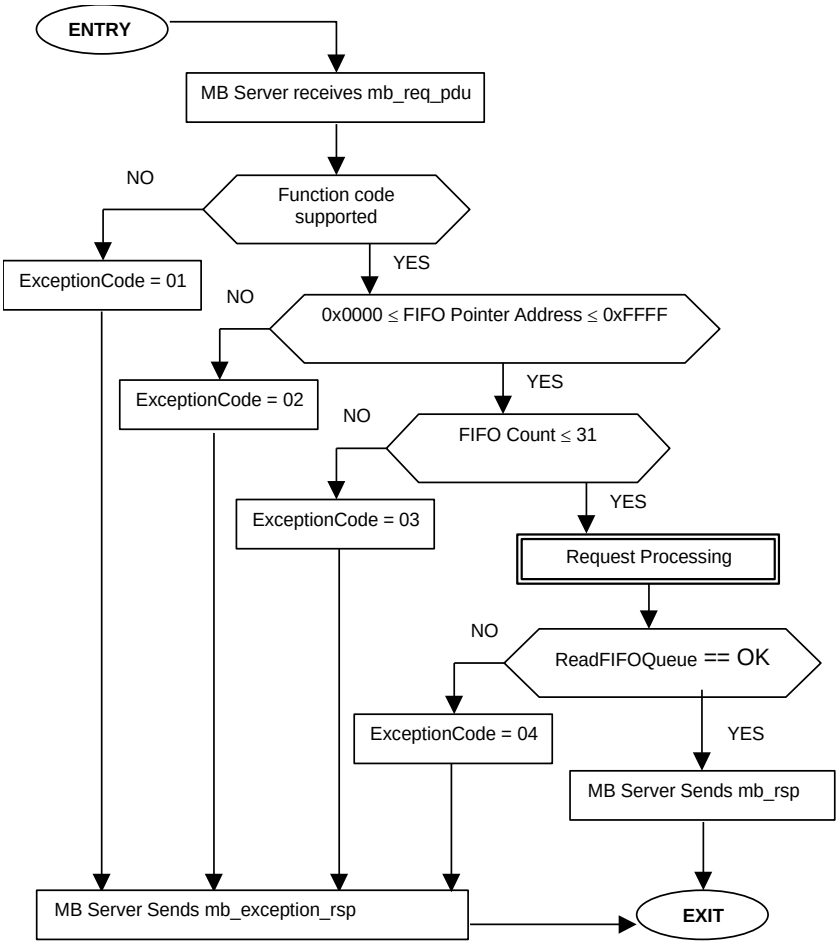

Figure 28: Read FIFO Queue state diagram

6.19 43 (0x2B) Encapsulated Interface Transport

Informative Note: The user is asked to refer to Annex A (Informative) MODBUS RESERVED FUNCTION CODES, SUBCODES AND MEI TYPES.

Function Code 43 and its MEI Type 14 for Device Identification is one of two Encapsulated Interface Transport currently available in this Specification. The following function codes and MEI Types shall not be part of this published Specification and these function codes and MEI Types are specifically reserved: 43/0-12 and 43/15-255.

The MODBUS Encapsulated Interface (MEI)Transport is a mechanism for tunneling service requests and method invocations, as well as their returns, inside MODBUS PDUs.

The primary feature of the MEI Transport is the encapsulation of method invocations or service requests that are part of a defined interface as well as method invocation returns or service responses.

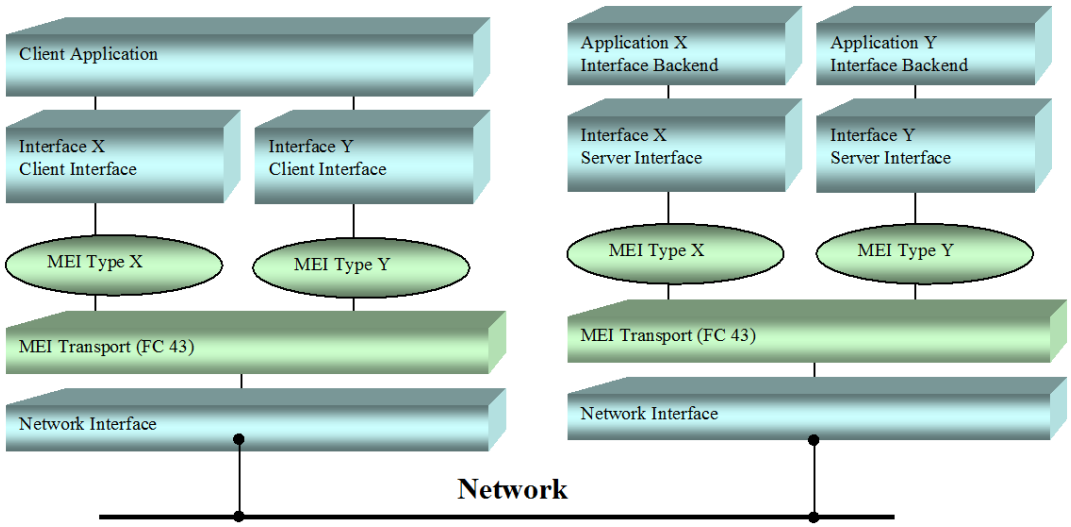

Figure 29: MODBUS encapsulated Interface Transport

The **Network Interface** can be any communication stack used to send MODBUS PDUs, such as TCP/IP, or serial line.

A **MEI Type** is a MODBUS Assigned Number and therefore will be unique, the value between 0 to 255 are Reserved according to Annex A (Informative) except for MEI Type 13 and MEI Type 14.

The MEI Type is used by MEI Transport implementations to dispatch a method invocation to the indicated interface.

Since the MEI Transport service is interface agnostic, any specific behavior or policy required by the interface must be provided by the interface, e.g. MEI transaction processing, MEI interface error handling, etc.

Request

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

* MEI = MODBUS Encapsulated Interface

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 :** **Fc 0x2B + 0x80**
Exception code	1 Byte	01 or 02 or 03 or 04

As an example see Read device identification request.

6.20 43 / 13 (0x2B / 0x0D) CANopen General Reference Request and Response PDU

The CANopen General reference Command is an encapsulation of the services that will be used to access (read from or write to) the entries of a CAN-Open Device Object Dictionary as well as controlling and monitoring the CANopen system, and devices.

The MEI Type 13 (0x0D) is a MODBUS Assigned Number licensed to CiA for the CANopen General Reference.

The system is intended to work within the limitations of existing MODBUS networks. Therefore, the information needed to query or modify the object dictionaries in the system is

mapped into the format of a MODBUS message. The PDU will have the 253 Byte limitation in both the Request and the Response message.

Informative: Please refer to Annex B for a reference to a specification that provides information on MEI Type 13.

6.21 43 / 14 (0x2B / 0x0E) Read Device Identification

This function code allows reading the identification and additional information relative to the physical and functional description of a remote device, only.

The Read Device Identification interface is modeled as an address space composed of a set of addressable data elements. The data elements are called objects and an object Id identifies them.

The interface consists of 3 categories of objects :

- Basic Device Identification. All objects of this category are mandatory : VendorName, Product code, and revision number.
- Regular Device Identification. In addition to Basic data objects, the device provides additional and optional identification and description data objects. All of the objects of this category are defined in the standard but their implementation is optional .
- Extended Device Identification. In addition to regular data objects, the device provides additional and optional identification and description private data about the physical device itself. All of these data are device dependent.

Object Id	Object Name / Description	Type	M/O	category
0x00	VendorName	ASCII String	**Mandatory**	**Basic**
0x01	ProductCode	ASCII String	**Mandatory**	
0x02	MajorMinorRevision	ASCII String	**Mandatory**	
0x03	VendorUrl	ASCII String	Optional	**Regular**
0x04	ProductName	ASCII String	Optional	
0x05	ModelName	ASCII String	Optional	
0x06	UserApplicationName	ASCII String	Optional	
0x07 … 0x7F	*Reserved*		Optional	
0x80 … 0xFF	*Private objects may be **optionally** defined. The range [0x80 – 0xFF] is Product dependant.*	device dependant	Optional	**Extended**

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

* MEI = MODBUS Encapsulated Interface

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	00 / FF
Next Object Id	1 Byte	Object ID number
Number of objects	1 Byte	
List Of		
Object ID	1 Byte	
Object length	1 Byte	
Object Value	Object length	Depending on the object ID

Error

Function code	1 Byte	**0xAB :** **Fc 0x2B + 0x80**

Exception code	1 Byte	01 or 02 or 03 or 04

Request parameters description :

A MODBUS Encapsulated Interface assigned number 14 identifies the Read identification request.

The parameter " Read Device ID code " allows to define four access types :

01: request to get the basic device identification (stream access)
02: request to get the regular device identification (stream access)
03: request to get the extended device identification (stream access)
04: request to get one specific identification object (individual access)

An exception code 03 is sent back in the response if the Read device ID code is illegal.

In case of a response that does not fit into a single response, several transactions (request/response) must be done. The Object Id byte gives the identification of the first object to obtain. For the first transaction, the client must set the Object Id to 0 to obtain the beginning of the device identification data. For the following transactions, the client must set the Object Id to the value returned by the server in its previous response.

Remark : An object is indivisible, therefore any object must have a size consistent with the size of transaction response.

If the Object Id does not match any known object, the server responds as if object 0 were pointed out (restart at the beginning).

In case of an individual access: ReadDevId code 04**,** the Object Id in the request gives the identification of the object to obtain, and if the Object Id doesn't match to any known object, the server returns an exception response with exception code = 02 (Illegal data address).

If the server device is asked for a description level (readDevice Code)higher that its conformity level , It must respond in accordance with its actual conformity level.

Response parameter description :

Function code :	Function code 43 (decimal) 0x2B (hex)
MEI Type	14 (0x0E) MEI Type assigned number for Device Identification Interface
ReadDevId code :	Same as request ReadDevId code : 01, 02, 03 or 04
Conformity Level	Identification conformity level of the device and type of supported access 0x01: basic identification (stream access only) 0x02: regular identification (stream access only) 0x03: extended identification (stream access only) 0x81: basic identification (stream access and individual access) 0x82: regular identification (stream access and individual access) 0x83: extended identification(stream access and individual access)
More Follows	***In case of ReadDevId codes 01, 02 or 03 (stream access),*** If the identification data doesn't fit into a single response, several request/response transactions may be required. 0x00 : no more Object are available 0xFF : other identification Object are available and further MODBUS transactions are required ***In case of ReadDevId code 04 (individual access),*** this field must be set to 00.
Next Object Id	If "MoreFollows = FF", identification of the next Object to be asked for. If "MoreFollows = 00", must be set to 00 (useless)
Number Of Objects	Number of identification Object returned in the response (for an individual access, Number Of Objects = 1)
Object0.Id	Identification of the first Object returned in the PDU (stream access) or the requested Object (individual access)

Object0.Length	Length of the first Object in byte
Object0.Value	Value of the first Object (Object0.Length bytes)
…	
ObjectN.Id	Identification of the last Object (within the response)
ObjectN.Length	Length of the last Object in byte
ObjectN.Value	Value of the last Object (ObjectN.Length bytes)

Example of a Read Device Identification request for "Basic device identification" : In this example all information are sent in one response PDU.

Request		**Response**	
Field Name	*Value*	*Field Name*	*Value*
Function	**2B**	Function	**2B**
MEI Type	**0E**	MEI Type	**0E**
Read Dev Id code	**01**	Read Dev Id Code	**01**
Object Id	**00**	Conformity Level	**01**
		More Follows	**00**
		NextObjectId	**00**
		Number Of Objects	**03**
		Object Id	**00**
		Object Length	**16**
		Object Value	**" Company identification"**
		Object Id	**01**
		Object Length	**0D**
		Object Value	**" Product code XX"**
		Object Id	**02**
		Object Length	**05**
		Object Value	**"V2.11"**

In case of a device that required several transactions to send the response the following transactions is intiated.

First transaction :

Request		**Response**	
Field Name	*Value*	*Field Name*	*Value*
Function	**2B**	Function	**2B**
MEI Type	**0E**	MEI Type	**0E**
Read Dev Id code	**01**	Read Dev Id Code	**01**
Object Id	**00**	Conformity Level	**01**
		More Follows	**FF**
		NextObjectId	**02**
		Number Of Objects	**03**
		Object Id	**00**
		Object Length	**16**
		Object Value	**" Company identification"**
		Object Id	**01**
		Object Length	**1C**
		Object Value	**" Product code XXXXXXXXXXXXXXXX"**

Second transaction :

Request		**Response**	
Field Name	*Value*	*Field Name*	*Value*
Function	**2B**	Function	**2B**
MEI Type	**0E**	MEI Type	**0E**
Read Dev Id code	**01**	Read Dev Id Code	**01**
Object Id	**02**	Conformity Level	**01**
		More Follows	**00**
		NextObjectId	**00**
		Number Of Objects	**03**
		Object Id	**02**
		Object Length	**05**
		Object Value	**"V2.11"**

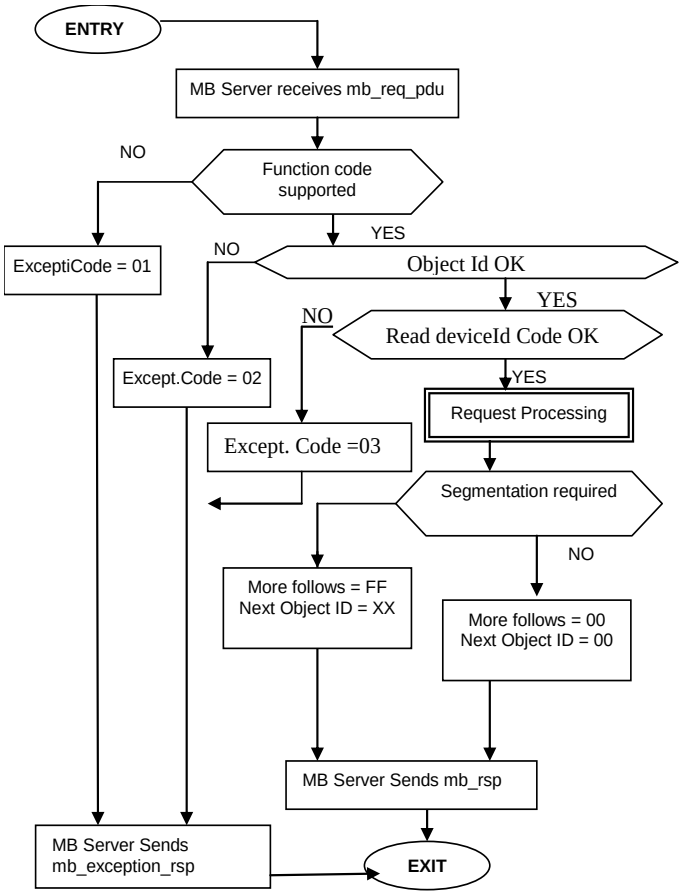

Figure 30: Read Device Identification state diagram

7 MODBUS Exception Responses

When a client device sends a request to a server device it expects a normal response. One of four possible events can occur from the client's query:

- If the server device receives the request without a communication error, and can handle the query normally, it returns a normal response.
- If the server does not receive the request due to a communication error, no response is returned. The client program will eventually process a timeout condition for the request.
- If the server receives the request, but detects a communication error (parity, LRC, CRC, ...), no response is returned. The client program will eventually process a timeout condition for the request.
- If the server receives the request without a communication error, but cannot handle it (for example, if the request is to read a non–existent output or register), the server will return an exception response informing the client of the nature of the error.

The exception response message has two fields that differentiate it from a normal response:

Function Code Field: In a normal response, the server echoes the function code of the original request in the function code field of the response. All function codes have a most–significant bit (MSB) of 0 (their values are all below 80 hexadecimal). In an exception response, the server sets the MSB of the function code to 1. This makes the function code value in an exception response exactly 80 hexadecimal higher than the value would be for a normal response.

With the function code's MSB set, the client's application program can recognize the exception response and can examine the data field for the exception code.

Data Field: In a normal response, the server may return data or statistics in the data field (any information that was requested in the request). In an exception response, the server returns an exception code in the data field. This defines the server condition that caused the exception.

Example of a client request and server exception response

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**01**	Function	**81**
Starting Address Hi	**04**	Exception Code	**02**
Starting Address Lo	**A1**		
Quantity of Outputs Hi	**00**		
Quantity of Outputs Lo	**01**		

In this example, the client addresses a request to server device. The function code (01) is for a Read Output Status operation. It requests the status of the output at address 1185 (04A1 hex). Note that only that one output is to be read, as specified by the number of outputs field (0001).

If the output address is non–existent in the server device, the server will return the exception response with the exception code shown (02). This specifies an illegal data address for the server.

A listing of exception codes begins on the next page.

MODBUS Exception Codes		
Code	**Name**	**Meaning**
01	ILLEGAL FUNCTION	The function code received in the query is not an allowable action for the server. 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 server 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.
02	ILLEGAL DATA ADDRESS	The data address received in the query is not an allowable address for the server. More specifically, the combination of reference number and transfer length is invalid. For a controller with 100 registers, the PDU addresses the first register as 0, and the last one as 99. If a request is submitted with a starting register address of 96 and a quantity of registers of 4, then this request will successfully operate (address-wise at least) on registers 96, 97, 98, 99. If a request is submitted with a starting register address of 96 and a quantity of registers of 5, then this request will fail with Exception Code 0x02 “Illegal Data Address” since it attempts to operate on registers 96, 97, 98, 99 and 100, and there is no register with address 100.
03	ILLEGAL DATA VALUE	A value contained in the query data field is not an allowable value for server. This indicates a fault in the structure of the 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.
04	SERVER DEVICE FAILURE	An unrecoverable error occurred while the server was attempting to perform the requested action.
05	ACKNOWLEDGE	Specialized use in conjunction with programming commands. The server 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 client. The client can next issue a Poll Program Complete message to determine if processing is completed.
06	SERVER DEVICE BUSY	Specialized use in conjunction with programming commands. The server is engaged in processing a long–duration program command. The client should retransmit the message later when the server is free.
08	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 server attempted to read record file, but detected a parity error in the memory. The client can retry the request, but service may be required

		on the server device.
0A	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 that the gateway is misconfigured or overloaded.
0B	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.

Annex A (Informative): MODBUS RESERVED FUNCTION CODES, SUBCODES AND MEI TYPES

The following function codes and subcodes shall not be part of this published Specification and these function codes and subcodes are specifically reserved. The format is function code/subcode or just function code where all the subcodes (0-255) are reserved: 8/19; 8/21-65535, 9, 10, 13, 14, 41, 42, 90, 91, 125, 126 and 127.

Function Code 43 and its MEI Type 14 for Device Identification and MEI Type 13 for CANopen General Reference Request and Response PDU are the currently available Encapsulated Interface Transports in this Specification.

The following function codes and MEI Types shall not be part of this published Specification and these function codes and MEI Types are specifically reserved: 43/0-12 and 43/15-255. In this Specification, a User Defined Function code having the same or similar result as the Encapsulated Interface Transport is not supported.

MODBUS is a registered trademark of Schneider Automation Inc.

Annex B (Informative): CANOPEN GENERAL REFERENCE COMMAND

Please refer to the MODBUS website or the CiA (CAN in Automation) website for a copy and terms of use that cover Function Code 43 MEI Type 13.