

---

**Content**

1	Introduction .....	2
1.1	Scope of this document .....	2
1.2	References .....	2
2	Abbreviations .....	3
3	Context .....	3
4	General description .....	5
4.1	Protocol description .....	5
4.2	Data Encoding .....	7
4.3	MODBUS data model .....	7
4.4	Define MODBUS Transaction .....	9
5	Function Code Categories .....	11
5.1	Public Function Code Definition .....	12
6	Function codes descriptions .....	12
6.1	01 (0x01) Read Coils .....	12
6.2	02 (0x02) Read Discrete Inputs .....	14
6.3	03 (0x03) Read Holding Registers .....	17
6.4	04 (0x04) Read Input Registers .....	18
6.5	05 (0x05) Write Single Coil .....	20
6.6	06 (0x06) Write Single Register .....	22
6.7	15 (0x0F) Write Multiple Coils .....	25
6.8	16 (0x10) Write Multiple registers .....	28
6.9	20 (0x14) Read File Record .....	29
6.10	22 (0x16) Mask Write Register .....	34
6.11	23 (0x17) Read/Write Multiple registers .....	35
6.12	43 (0x2B) Read Device Identification .....	38
7	MODBUS Exception Responses .....	43

# 1 Introduction

## 1.1 Scope of this document

MODBUS is an application layer messaging protocol, positioned at level 7 of the OSI model, that 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.

## 1.2 References

1. RFC 791, Internet Protocol, Sep81 DARPA
2. MODBUS Protocol Reference Guide Rev J, MODICON, June 1996, doc # PI\_MBUS\_300

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.

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.

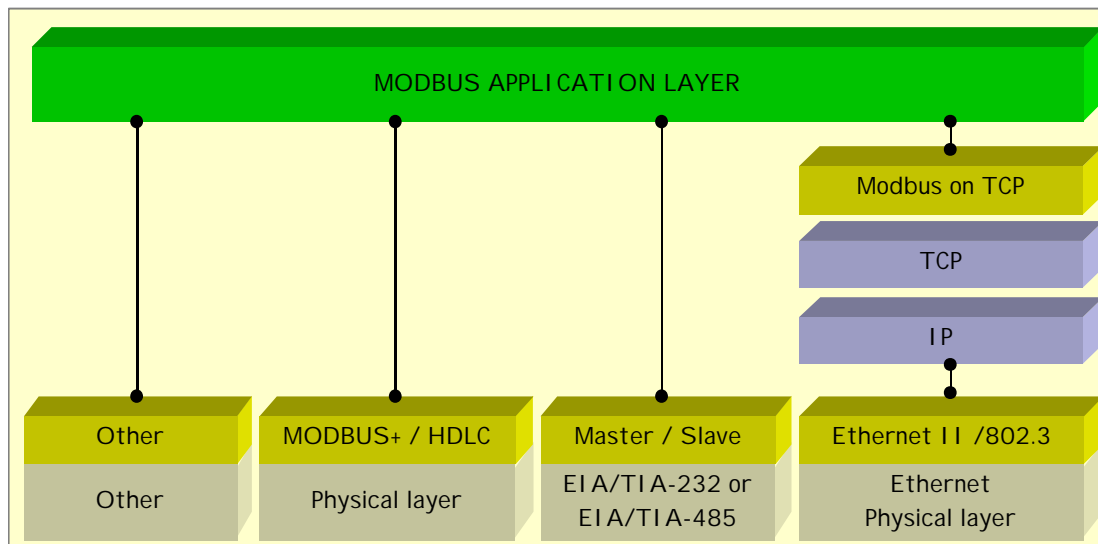


Figure 1: MODBUS communication stack

## 2 Abbreviations

<b>ADU</b>	Application Data Unit
<b>HDLC</b>	High level Data Link Control
<b>HMI</b>	Human Machine Interface
<b>IETF</b>	Internet Engineering Task Force
<b>I/O</b>	Input/Output
<b>IP</b>	Internet Protocol
<b>MAC</b>	Medium Access Control
<b>MB</b>	MODBUS Protocol
<b>MBAP</b>	MODBUS Application Protocol
<b>PDU</b>	Protocol Data Unit
<b>PLC</b>	Programmable Logic Controller
<b>TCP</b>	Transport Control Protocol

## 3 Context

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

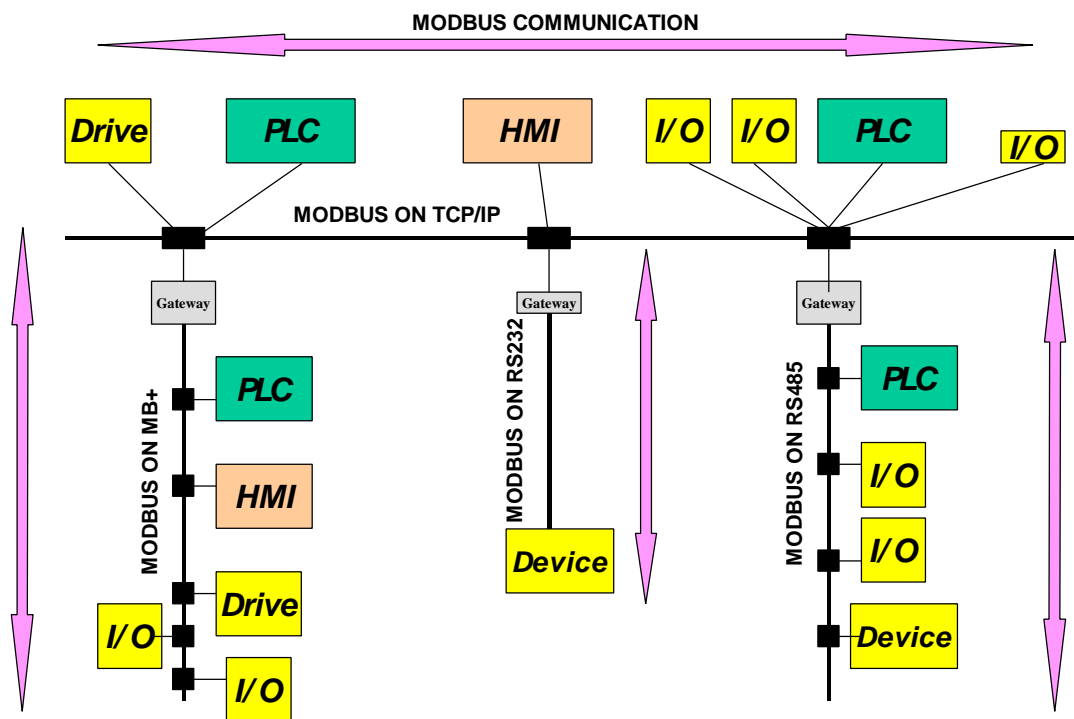


Figure 2: Example of MODBUS Network Architecture

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.

Some gateway allows a communication between several types of buses or network using the MODBUS protocol.

4 General description

4.1 Protocol description

The MODBUS protocol defined 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**).

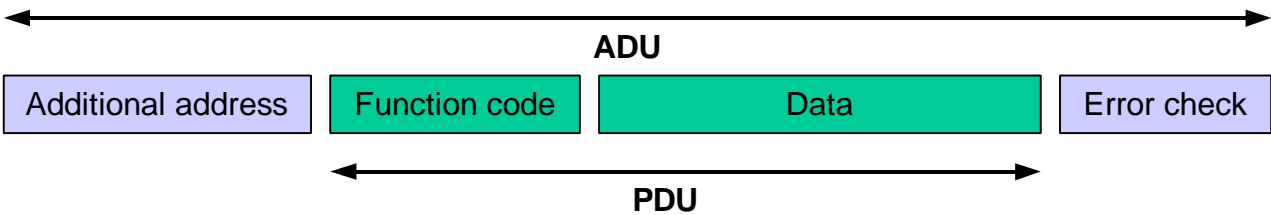


Figure 3: General MODBUS frame

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 (128 – 255 reserved 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.

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 request, 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 the original function code.

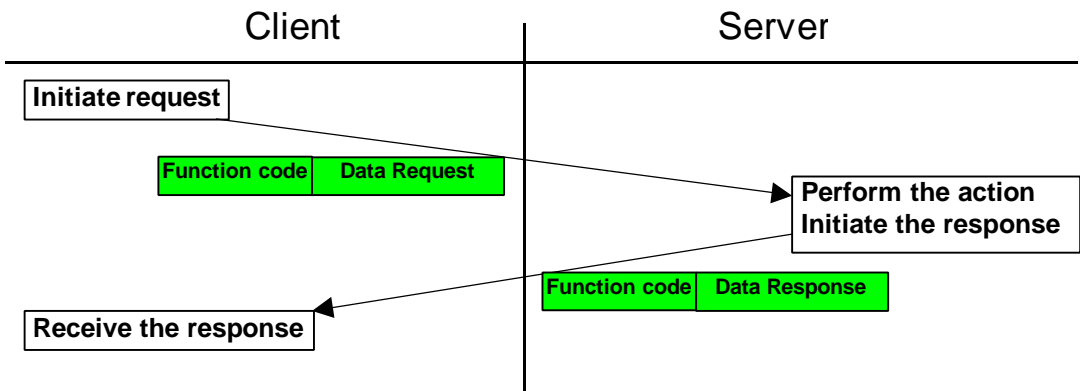


Figure 4: MODBUS transaction (error free)

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

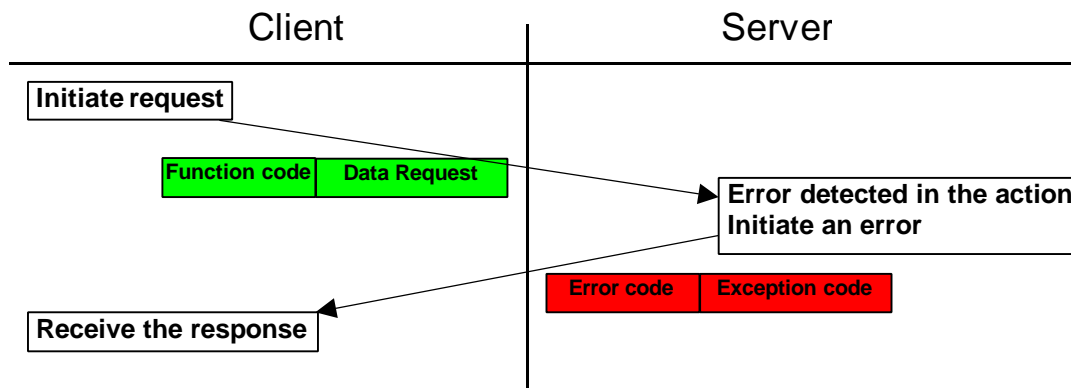


Figure 5: MODBUS transaction (exception response)



**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** = 249 bytes + MBAP (7 bytes) = **256 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 :

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 :

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 :

mb\_excep\_rsp\_pdu = { function\_code, request\_data),     where  
function\_code - [1 byte] MODBUS function code + 0x80  
exception\_code - [1 byte] MODBUS Exception Code Defined in table  
below.

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

<u>Register size</u>	<u>value</u>	
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 access	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 number, 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, all are not 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 from each other, because data from different block have no correlation. Each block is thus accessible with different MODBUS functions.

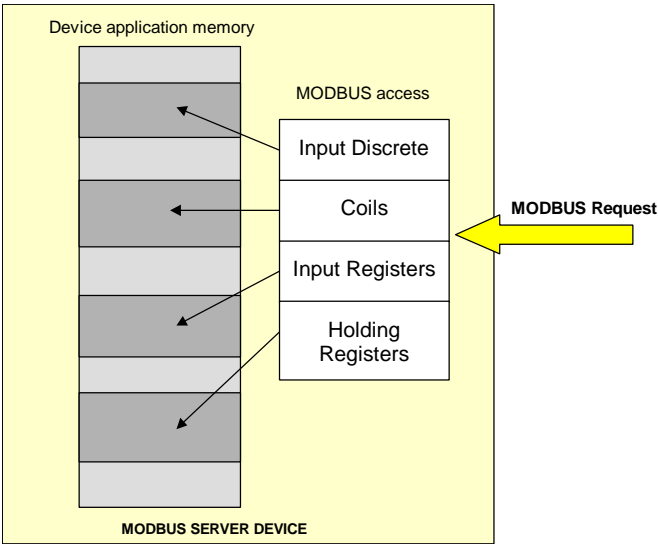


Figure 6 MODBUS Data Model with separate block

Example 2: Device having only 1 block

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

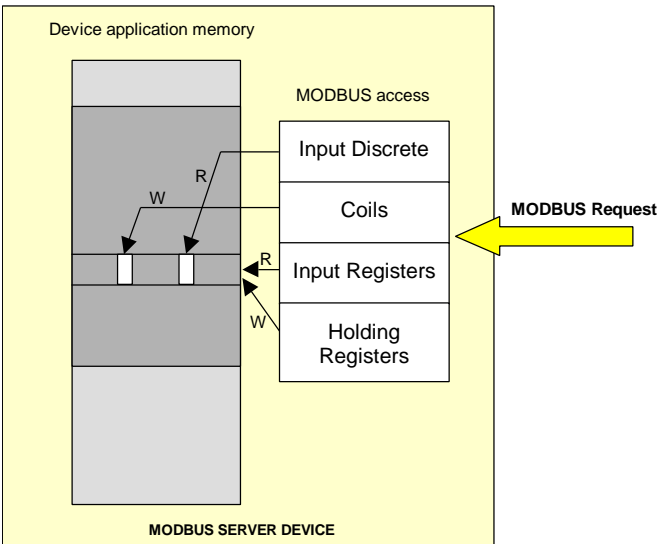


Figure 7 MODBUS Data Model with only 1 block



4.4 Define MODBUS Transaction

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

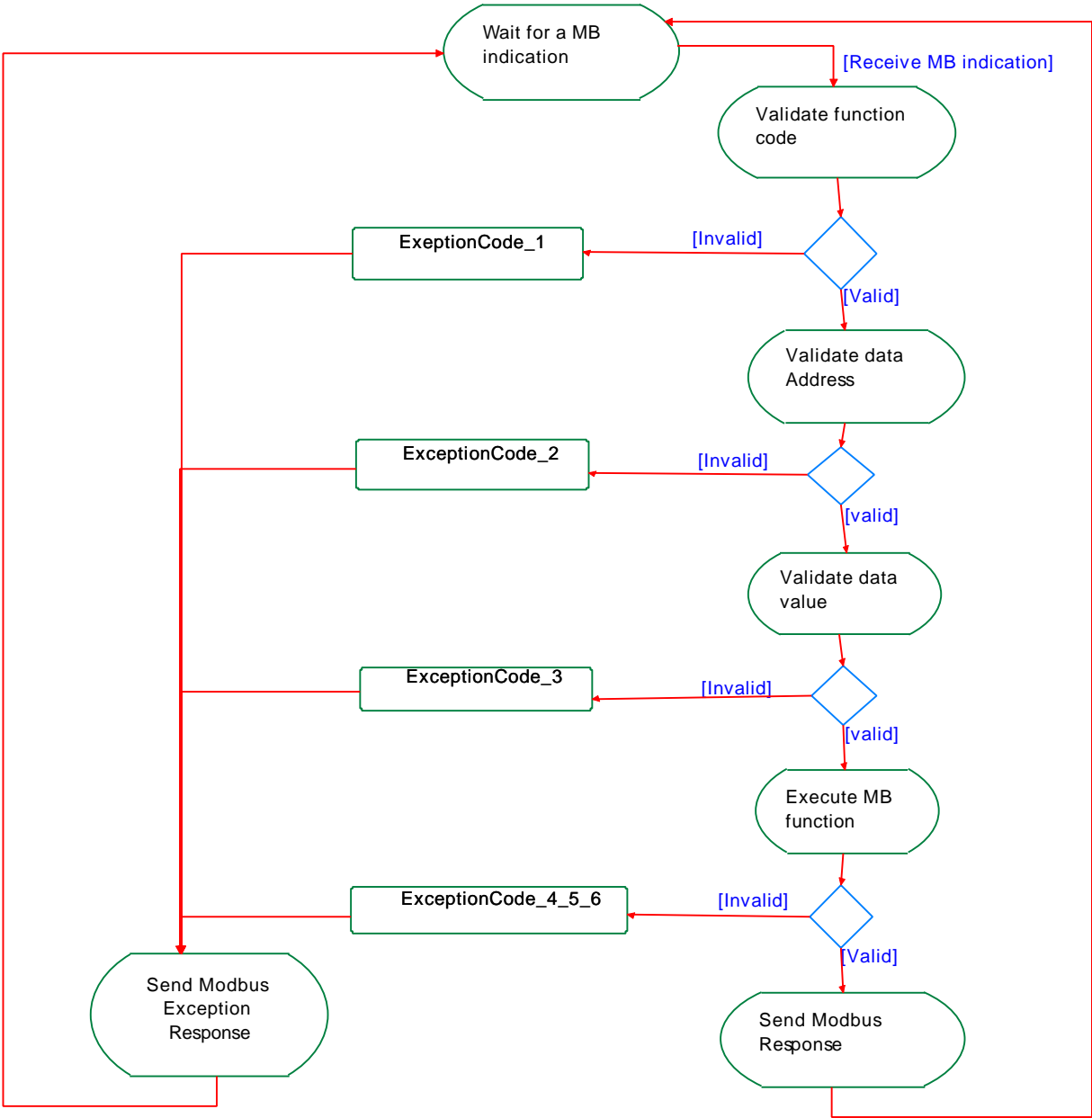


Figure 8 MODBUS Transaction state diagram

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 can be built :

- A positive MODBUS response :

- the response function code = the request function code
- A MODBUS Exception response ( see chapter 6.14):
  - the objective is to provide to the client relevant information concerning the error detected during the processing ;
  - the response 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,
- publically documented
- have available conformance test,
- are documented in the MB IETF RFC,
- includes both defined public assigned function codes as well as unassigned function codes reserved for future use.

User-Defined Function Codes

- there is a defined two ranges of user-defined function codes, ie 65 to 72 and from 100 to 110 decimal.
- user can select and implement a function code without any approval from modbus.org
- 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.

Reserved Function Codes

- Function Codes currently used by some companies for legacy products and that are not available for public use.

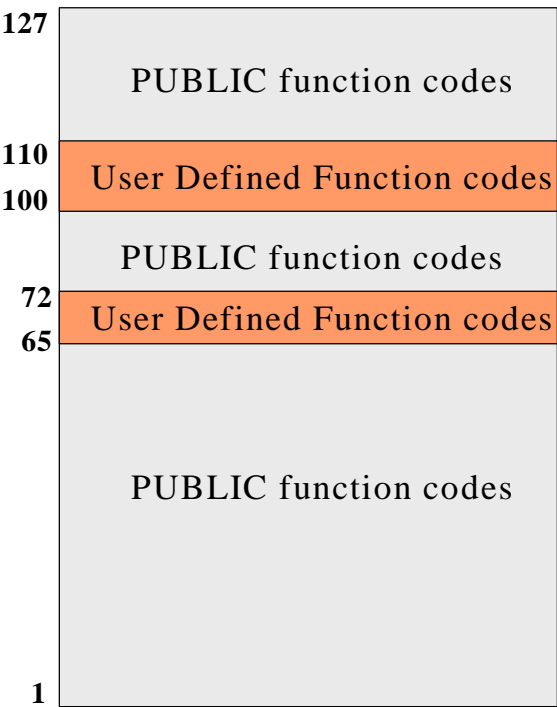


Figure 9 MODBUS Function Code Categories

## 5.1 Public Function Code Definition

				Function Codes		(hex)	Page
				code	Sub code		
Data Access	Bit access	Physical Discrete Inputs	Read Input Discrete	02		02	<a href="#">11</a>
		Internal Bits Or Physical coils	Read Coils	01		01	<a href="#">10</a>
			Write Single Coil	05		05	<a href="#">16</a>
			Write Multiple Coils	15		0F	<a href="#">37</a>
	16 bits access	Physical Input Registers	Read Input Register	04		04	<a href="#">14</a>
		Internal Registers Or Physical Output Registers	Read Multiple Registers	03		03	<a href="#">13</a>
			Write Single Register	06		06	<a href="#">17</a>
			Write Multiple Registers	16		10	<a href="#">39</a>
			Read/Write Multiple Registers	23		17	<a href="#">47</a>
			Mask Write Register	22		16	<a href="#">46</a>
	File record access		Read File record	20	6	14	<a href="#">42</a>
			Write File record	21	6	15	<a href="#">44</a>
Encapsulated Interface			Read Device Identification	43	14	2B	

## 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, ie the address of the first coil specified, and the number of coils. Coils are addressed starting at zero. Therefore coils 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 PDU

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

#### Response PDU

Function code	1 Byte	0x01
---------------	--------	------

Byte count	1 Byte	<b>N*</b>
Coil Status	<b>n</b> Byte	n = N or N+1

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

### Error

Function code	1 Byte	<b>Function code + 0x80</b>
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	<b>01</b>	Function	<b>01</b>
Starting Address Hi	<b>00</b>	Byte Count	<b>03</b>
Starting Address Lo	<b>13</b>	Outputs status 27-20	<b>CD</b>
Quantity of Outputs Hi	<b>00</b>	Outputs status 35-28	<b>6B</b>
Quantity of Outputs Lo	<b>13</b>	Outputs status 38-36	<b>05</b>

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.

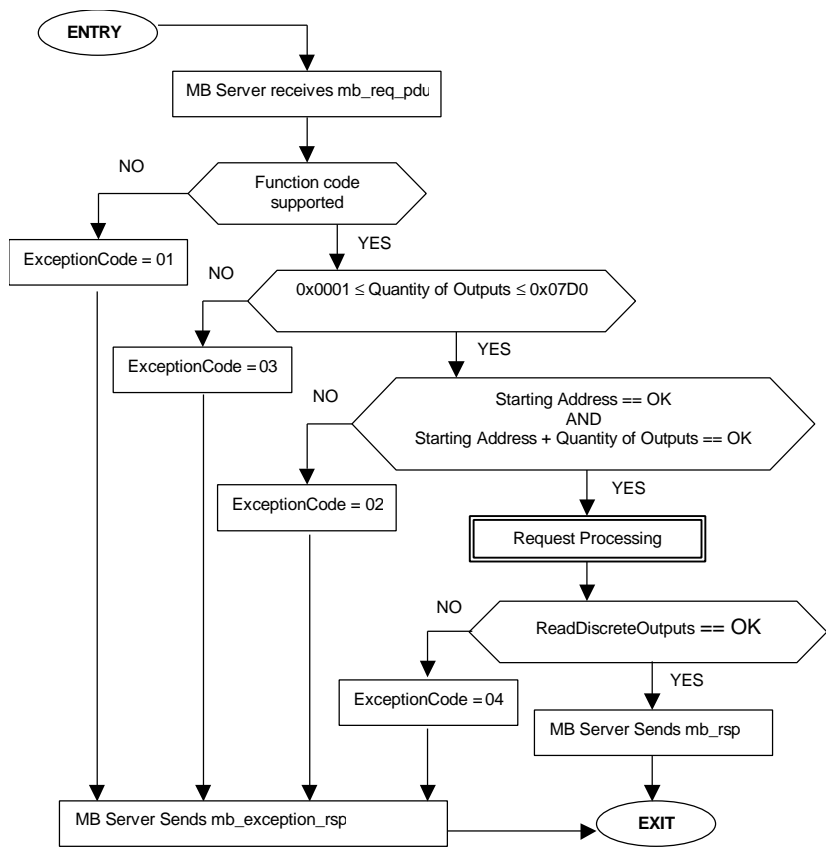


Figure 10: Read Coils state diagram

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, ie the address of the first input specified, and the number of inputs. Inputs are addressed starting at zero. Therefore inputs 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 PDU

Function code	1 Byte	0x02
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Inputs	2 Bytes	1 to 2000 (0x7D0)

## Response PDU

Function code	1 Byte	<b>0x02</b>
Byte count	1 Byte	<b>N*</b>
Input Status	<b>N*</b> x 1 Byte	

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

## Error

Error code	1 Byte	<b>0x82</b>
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	<b>02</b>	Function	<b>02</b>
Starting Address Hi	<b>00</b>	Byte Count	<b>03</b>
Starting Address Lo	<b>C4</b>	Inputs Status 204-197	<b>AC</b>
Quantity of Inputs Hi	<b>00</b>	Inputs Status 212-205	<b>DB</b>
Quantity of Inputs Lo	<b>16</b>	Inputs Status 218-213	<b>35</b>

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.

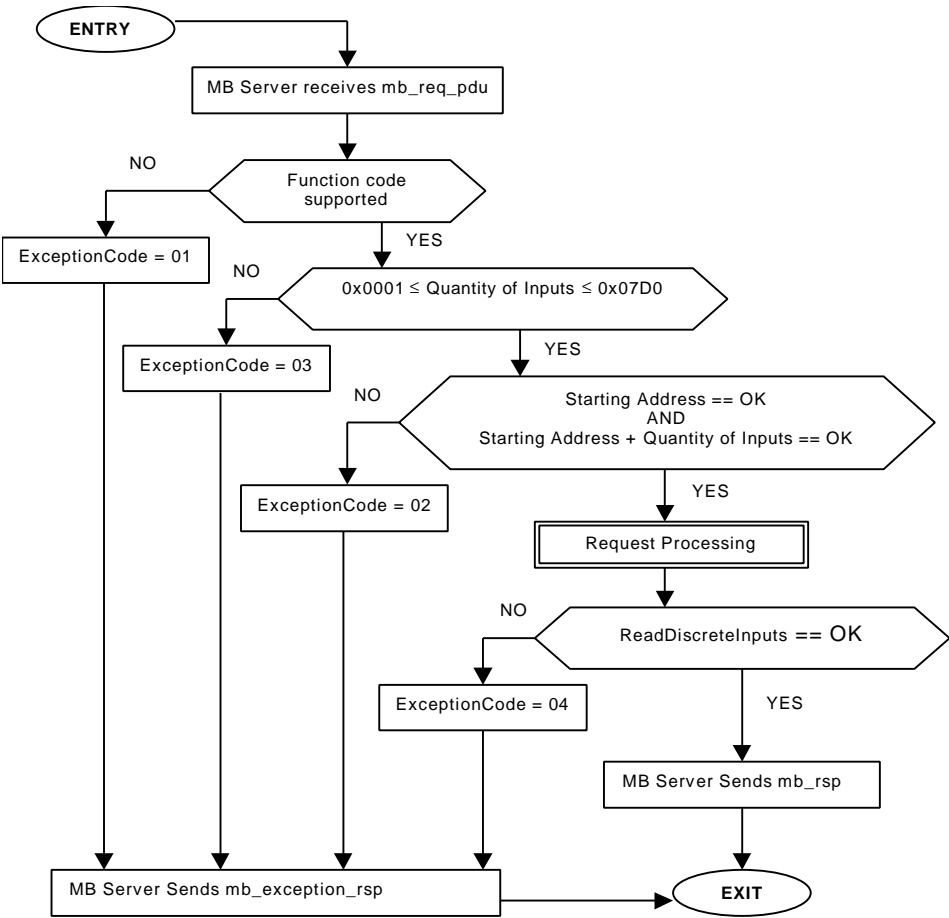


Figure 11: Read Discrete Inputs state diagram



### 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. Registers are addressed starting at zero. Therefore registers 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	<b>0x03</b>
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 125 (0x7D)

#### Response

Function code	1 Byte	<b>0x03</b>
Byte count	1 Byte	2 x <b>N</b> *
Register value	<b>N</b> * x 2 Bytes	

\***N** = Quantity of Registers

#### Error

Error code	1 Byte	<b>0x83</b>
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	<b>03</b>	Function	<b>03</b>
Starting Address Hi	<b>00</b>	Byte Count	<b>06</b>
Starting Address Lo	<b>6B</b>	Register value Hi (108)	<b>02</b>
No. of Registers Hi	<b>00</b>	Register value Lo (108)	<b>2B</b>
No. of Registers Lo	<b>03</b>	Register value Hi (109)	<b>00</b>
		Register value Lo (109)	<b>00</b>
		Register value Hi (110)	<b>00</b>
		Register value Lo (110)	<b>64</b>

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.

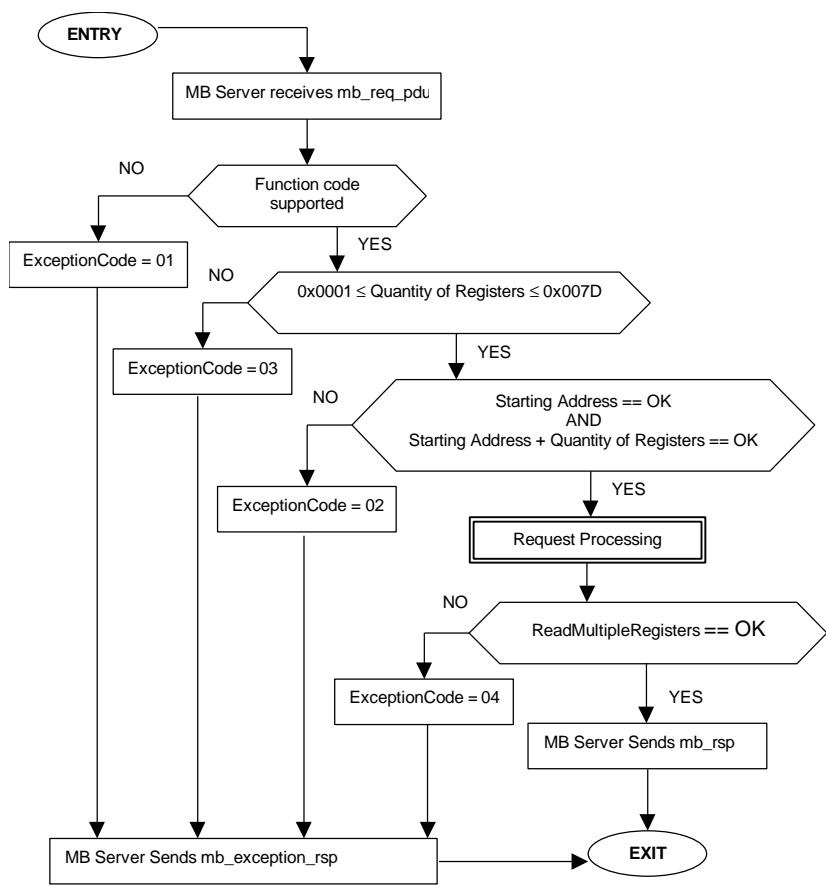


Figure 12: Read Holding Registers state diagram

6.4 04 (0x04) Read Input Registers

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

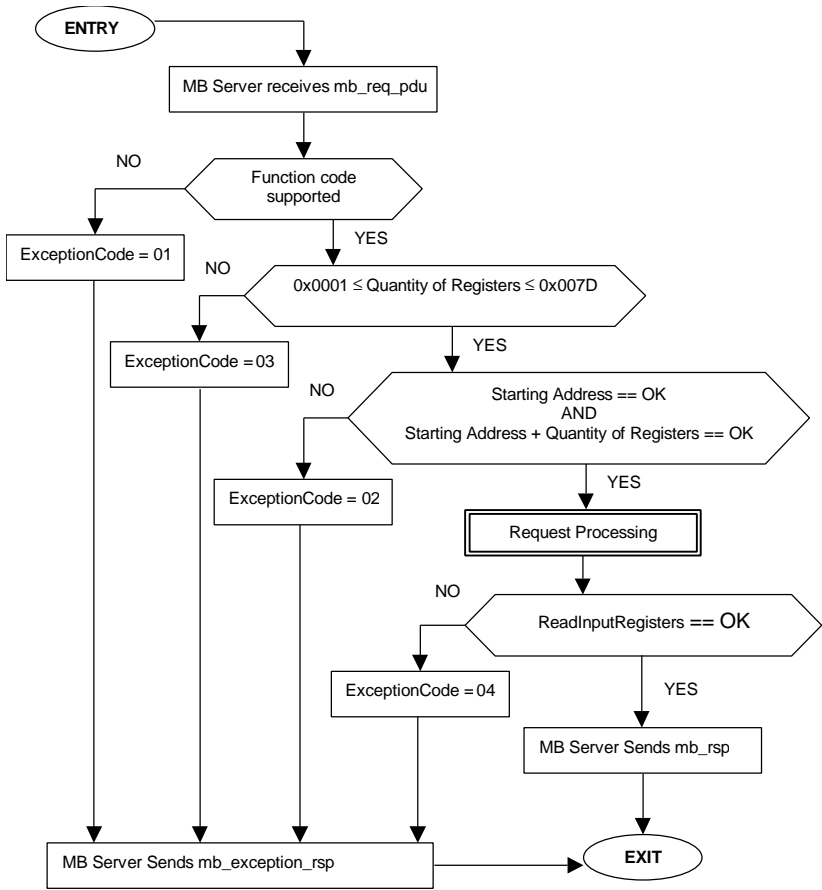


Figure 13: Read Input Registers state diagram

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 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	<b>0x05</b>
Output Address	2 Bytes	0x0000 to 0xFFFF
Output Value	2 Bytes	0x0000 or 0xFF00

#### Response

Function code	1 Byte	<b>0x05</b>
Output Address	2 Bytes	0x0000 to 0xFFFF
Output Value	2 Bytes	0x0000 or 0xFF00

#### Error

Error code	1 Byte	<b>0x85</b>
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	<b>05</b>	Function	<b>05</b>
Output Address Hi	<b>00</b>	Output Address Hi	<b>00</b>
Output Address Lo	<b>AC</b>	Output Address Lo	<b>AC</b>
Output Value Hi	<b>FF</b>	Output Value Hi	<b>FF</b>
Output Value Lo	<b>00</b>	Output Value Lo	<b>00</b>

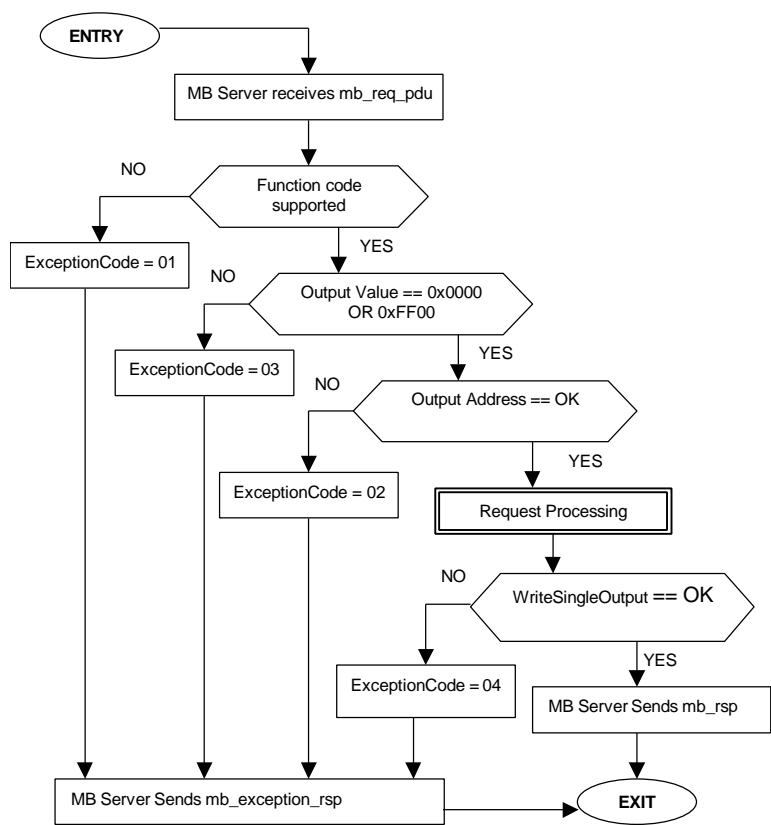


Figure 14: Write Single Output state diagram

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 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 or 0xFFFF

Response

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

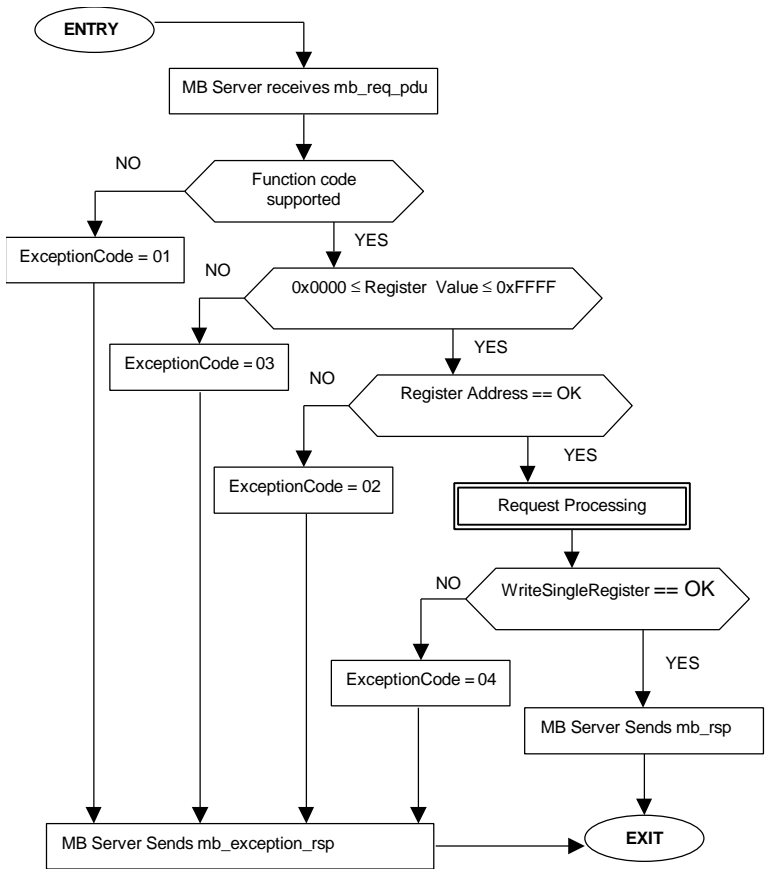


Figure 15: Write Single Register state diagram



## 6.7 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 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	<b>0x0F</b>
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Outputs	2 Bytes	0x0001 to 0x07B0
Byte Count	1 Byte	<b>N*</b>
Outputs Value	<b>N*</b> x 1 Byte	

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

### Response PDU

Function code	1 Byte	<b>0x0F</b>
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Outputs	2 Bytes	0x0001 to 0x07B0

### Error

Error code	1 Byte	<b>0x8F</b>
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		

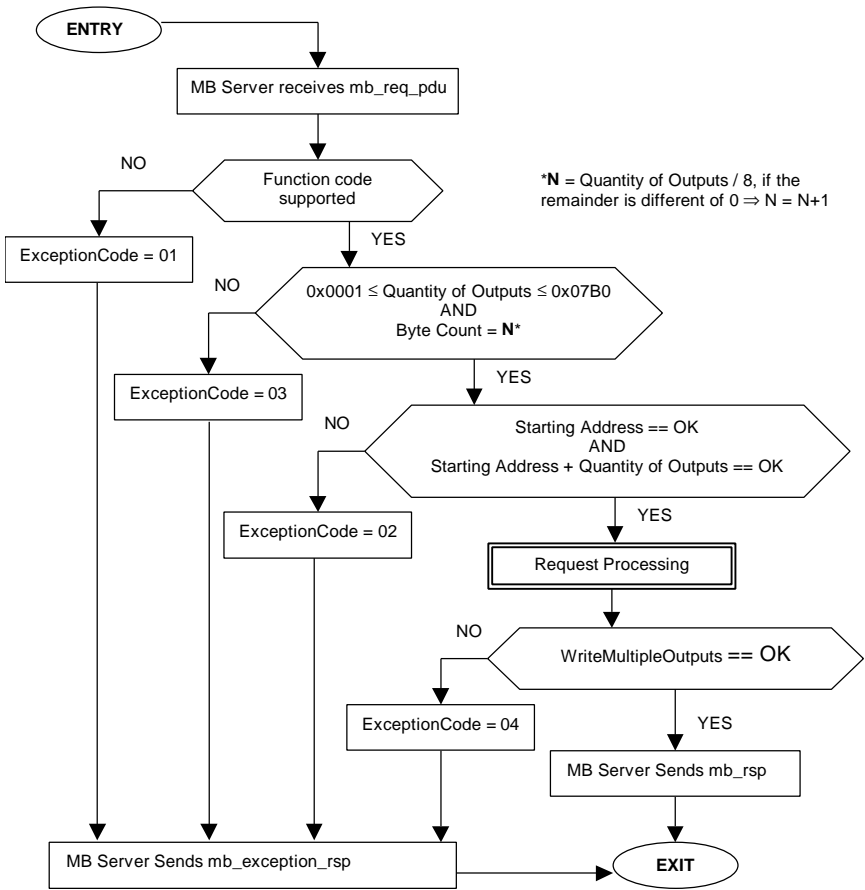


Figure 16: Write Multiple Outputs state diagram

## 6.8 16 (0x10) Write Multiple registers

This function code is used to write a block of contiguous registers (1 to approx. 120 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 PDU

Function code	1 Byte	<b>0x10</b>
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	0x0001 to 0x0078
Byte Count	1 Byte	2 x <b>N</b> *
Registers Value	<b>N</b> * x 2 Bytes	value

\***N** = Quantity of Registers

### Response PDU

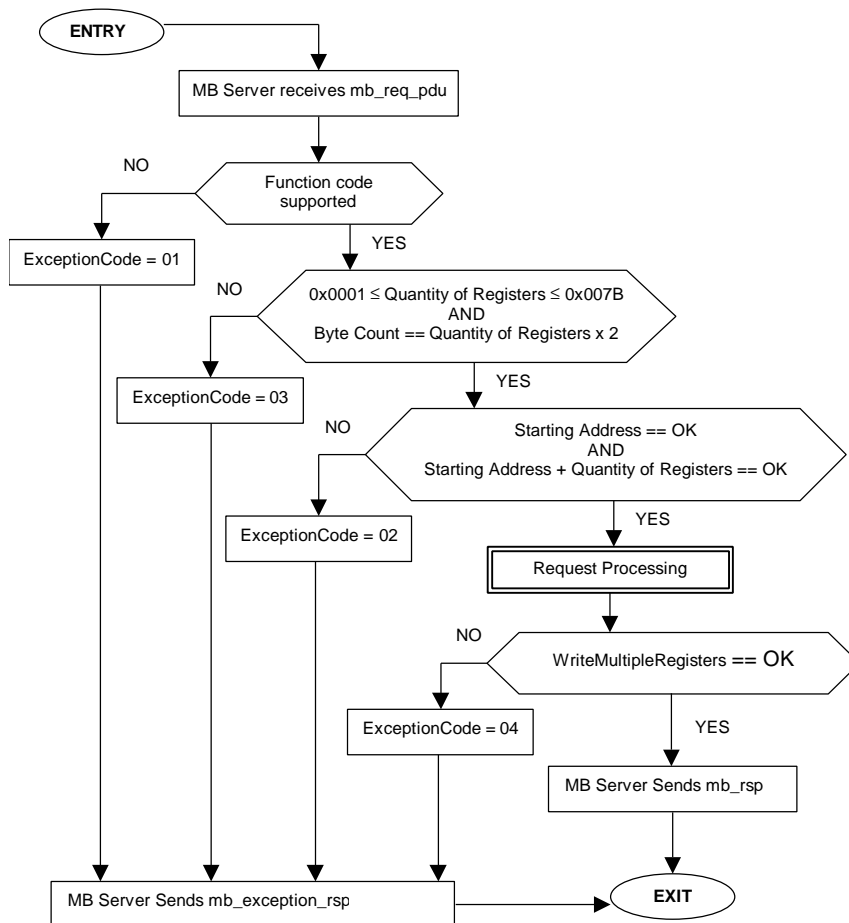
Function code	1 Byte	<b>0x10</b>
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 123 (0x7B)

### Error

Error code	1 Byte	<b>0x90</b>
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	<b>10</b>	Function	<b>10</b>
Starting Address Hi	<b>00</b>	Starting Address Hi	<b>00</b>
Starting Address Lo	<b>01</b>	Starting Address Lo	<b>01</b>
Quantity of Registers Hi	<b>00</b>	Quantity of Registers Hi	<b>00</b>
Quantity of Registers Lo	<b>02</b>	Quantity of Registers Lo	<b>02</b>
Byte Count	<b>04</b>		
Registers Value Hi	<b>00</b>		
Registers Value Lo	<b>0A</b>		
Registers Value Hi	<b>01</b>		
Registers Value Lo	<b>02</b>		



**Figure 17: Write Multiple Registers state diagram**

## 6.9 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 MODBUS messages: 256 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 PDU

Function code	1 Byte	<b>0x14</b>
Byte Count	1 Byte	0x07 to 0xF5 bytes
Sub-Req. x, Reference Type	1 Byte	06
Sub-Req. x, File Number	2 Bytes	0x0000 to 0xFFFF
Sub-Req. x, Record Number	2 Bytes	0x0000 to 0x270F
Sub-Req. x, Register Length	2 Bytes	<b>N</b>
Sub-Req. x+1, ...		

#### Response PDU

Function code	1 Byte	<b>0x14</b>
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	<b>N x 2 Bytes</b>	
Sub-Req. x+1, ...		

#### Error

Error code	1 Byte	<b>0x94</b>
Exception code	1 Byte	01 or 02 or 03 or 04 or 08

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	<b>14</b>	Function	<b>14</b>
Byte Count	<b>0C</b>	Resp. Data length	<b>0E</b>
Sub-Req. 1, Ref. Type	<b>06</b>	Sub-Req. 1, File resp. length	<b>05</b>
Sub-Req. 1, File Number Hi	<b>00</b>	Sub-Req. 1, Ref. Type	<b>06</b>
Sub-Req. 1, File Number Lo	<b>04</b>	Sub-Req. 1, Record. Data Hi	<b>0D</b>
Sub-Req. 1, Record number Hi	<b>00</b>	Sub-Req. 1, Record. Data Lo	<b>FE</b>
Sub-Req. 1, Record number Lo	<b>01</b>	Sub-Req. 1, Record. Data Hi	<b>00</b>
Sub-Req. 1, Record Length Hi	<b>00</b>	Sub-Req. 1, Record. Data Lo	<b>20</b>
Sub-Req. 1, Record Length Lo	<b>02</b>	Sub-Req. 2, File resp. length	<b>05</b>
Sub-Req. 2, Ref. Type	<b>06</b>	Sub-Req. 2, Ref. Type	<b>06</b>
Sub-Req. 2, File Number Hi	<b>00</b>	Sub-Req. 2, Record. Data Hi	<b>33</b>
Sub-Req. 2, File Number Lo	<b>03</b>	Sub-Req. 2, Record. Data Lo	<b>CD</b>
Sub-Req. 2, Record number Hi	<b>00</b>	Sub-Req. 2, Record. Data Hi	<b>00</b>
Sub-Req. 2, Record number Lo	<b>09</b>	Sub-Req. 2, Record. Data Lo	<b>40</b>

Sub-Req. 2, Record Length Hi	00	
Sub-Req. 2, Record Length Lo	02	

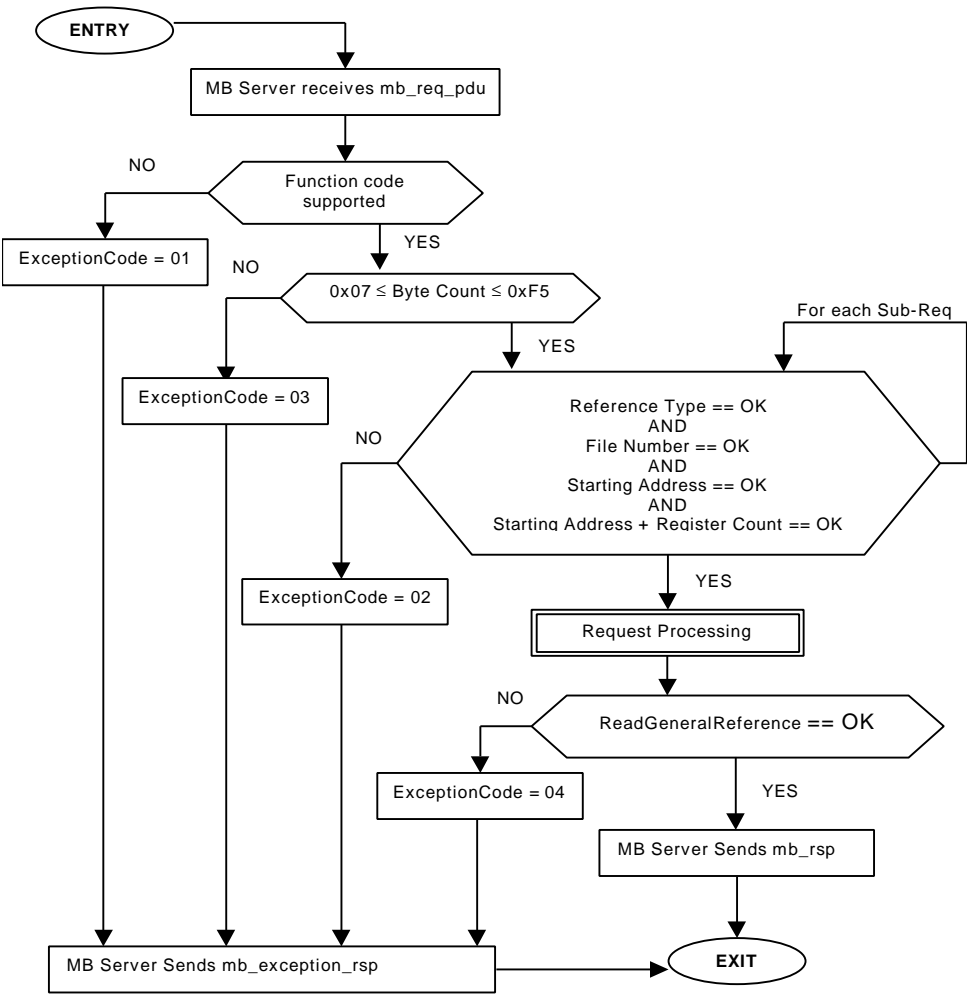


Figure 18: Read File Record state diagram

6.9.1 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, ie 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 query, must not exceed the allowable length of **MODBUS** messages: 256 bytes.

The normal response is an echo of the request.

#### Request PDU

Function code	1 Byte	<b>0x15</b>
Request data length	1 Byte	0x07 to 0xF5
Sub-Req. x, Reference Type	1 Byte	06
Sub-Req. x, File Number	2 Bytes	0x0000 to 0xFFFF
Sub-Req. x, Record Number	2 Bytes	0x0000 to 0x270F
Sub-Req. x, Record length	2 Bytes	<b>N</b>
Sub-Req. x, Record data	<b>N</b> x 2 Bytes	
Sub-Req. x+1, ...		

#### Response PDU

Function code	1 Byte	<b>0x15</b>
Response Data length	1 Byte	
Sub-Req. x, Reference Type	1 Byte	06
Sub-Req. x, File Number	2 Bytes	0x0000 to 0xFFFF
Sub-Req. x, Record number	2 Bytes	0x0000 to 0xFFFF
Sub-Req. x, Record length	2 Bytes	0x0000 to 0xFFFF <b>N</b>
Sub-Req. x, Record Data	<b>N</b> x 2 Bytes	
Sub-Req. x+1, ...		

#### Error

Error code	1 Byte	<b>0x95</b>
Exception code	1 Byte	01 or 02 or 03 or 04 or 08

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	<b>15</b>	Function	<b>15</b>
Request Data length	<b>0D</b>	Request Data length	<b>0D</b>
Sub-Req. 1, Ref. Type	<b>06</b>	Sub-Req. 1, Ref. Type	<b>06</b>
Sub-Req. 1, File Number Hi	<b>00</b>	Sub-Req. 1, File Number Hi	<b>00</b>
Sub-Req. 1, File Number Lo	<b>04</b>	Sub-Req. 1, File Number Lo	<b>04</b>
Sub-Req. 1, Record number Hi	<b>00</b>	Sub-Req. 1, Record number Hi	<b>00</b>
Sub-Req. 1, Record number Lo	<b>07</b>	Sub-Req. 1, Record number Lo	<b>07</b>



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, Record Data Hi	06	Sub-Req. 1, Record Data Hi	06
Sub-Req. 1, Record Data Lo	AF	Sub-Req. 1, Record Data Lo	AF
Sub-Req. 1, Record Data Hi	04	Sub-Req. 1, Record Data Hi	04
Sub-Req. 1, Record Data Lo	BE	Sub-Req. 1, Record Data Lo	BE
Sub-Req. 1, Record Data Hi	10	Sub-Req. 1, Record Data Hi	10
Sub-Req. 1, Reg. Data Lo	0D	Sub-Req. 1, Reg. Data Lo	0D

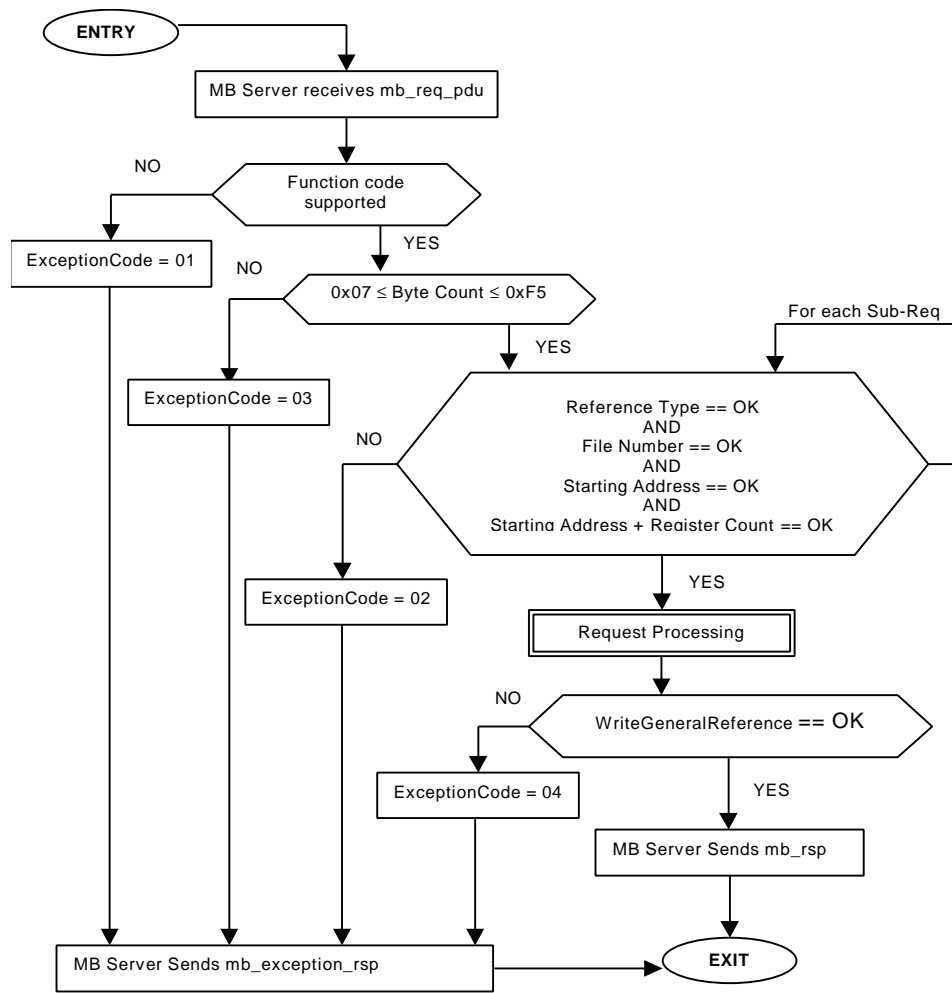


Figure 19: Write File Record state diagram

## 6.10 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 And\_Mask)

For example:

	Hex	Binary
Current Contents =	12	0001 0010
And_Mask =	F2	1111 0010
Or_Mask =	25	0010 0101
And_Mask =	0D	0000 1101
Result =	17	0001 0111



### Note:

That 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 PDU

Function code	1 Byte	<b>0x16</b>
Reference Address	2 Bytes	0x0000 to 0xFFFF
And_Mask	2 Bytes	0x0000 to 0xFFFF
Or_Mask	2 Bytes	0x0000 to 0xFFFF

### Response PDU

Function code	1 Byte	<b>0x16</b>
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	<b>0x96</b>
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	<b>16</b>	Function	<b>16</b>
Reference address Hi	<b>00</b>	Reference address Hi	<b>00</b>

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

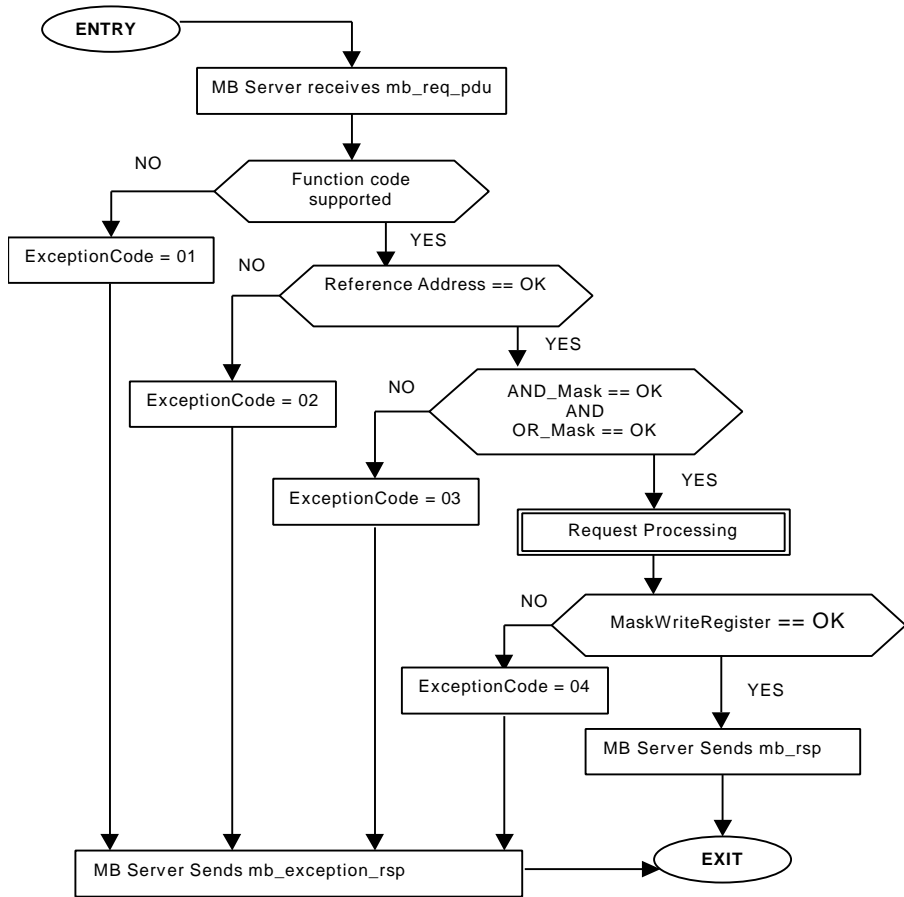


Figure 20: Mask Write Holding Register state diagram

6.11 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. Holding registers are addressed starting at zero. Therefore holding registers 1-16 are addressed 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 PDU

Function code	1 Byte	0x17
---------------	--------	------

Read Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity to Read	2 Bytes	0x0001 to approx. 0x0076
Write Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity to Write	2 Bytes	0x0001 to approx. 0x0076
Write Byte Count	1 Byte	2 x <b>N</b> *
Write Registers Value	<b>N</b> * x 2 Bytes	

\***N** = Quantity to Write

### Response PDU

Function code	1 Byte	<b>0x17</b>
Byte Count	1 Byte	2 x <b>N</b> *
Read Registers value	<b>N</b> * x 2 Bytes	

\***N** = Quantity to Read

### Error

Error code	1 Byte	<b>0x97</b>
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	<b>17</b>	Function	<b>17</b>
Read Starting Address Hi	<b>00</b>	Byte Count	<b>0C</b>
Read Starting Address Lo	<b>03</b>	Read Registers value Hi	<b>00</b>
Quantity to Read Hi	<b>00</b>	Read Registers value Lo	<b>FE</b>
Quantity to Read Lo	<b>06</b>	Read Registers value Hi	<b>0A</b>
Write Starting Address Hi	<b>00</b>	Read Registers value Lo	<b>CD</b>
Write Starting address Lo	<b>0E</b>	Read Registers value Hi	<b>00</b>
Quantity to Write Hi	<b>00</b>	Read Registers value Lo	<b>01</b>
Quantity to Write Lo	<b>03</b>	Read Registers value Hi	<b>00</b>
Write Byte Count	<b>06</b>	Read Registers value Lo	<b>03</b>
Write Registers Value Hi	<b>00</b>	Read Registers value Hi	<b>00</b>
Write Registers Value Lo	<b>FF</b>	Read Registers value Lo	<b>0D</b>
Write Registers Value Hi	<b>00</b>	Read Registers value Hi	<b>00</b>
Write Registers Value Lo	<b>FF</b>	Read Registers value Lo	<b>FF</b>
Write Registers Value Hi	<b>00</b>		
Write Registers Value Lo	<b>FF</b>		

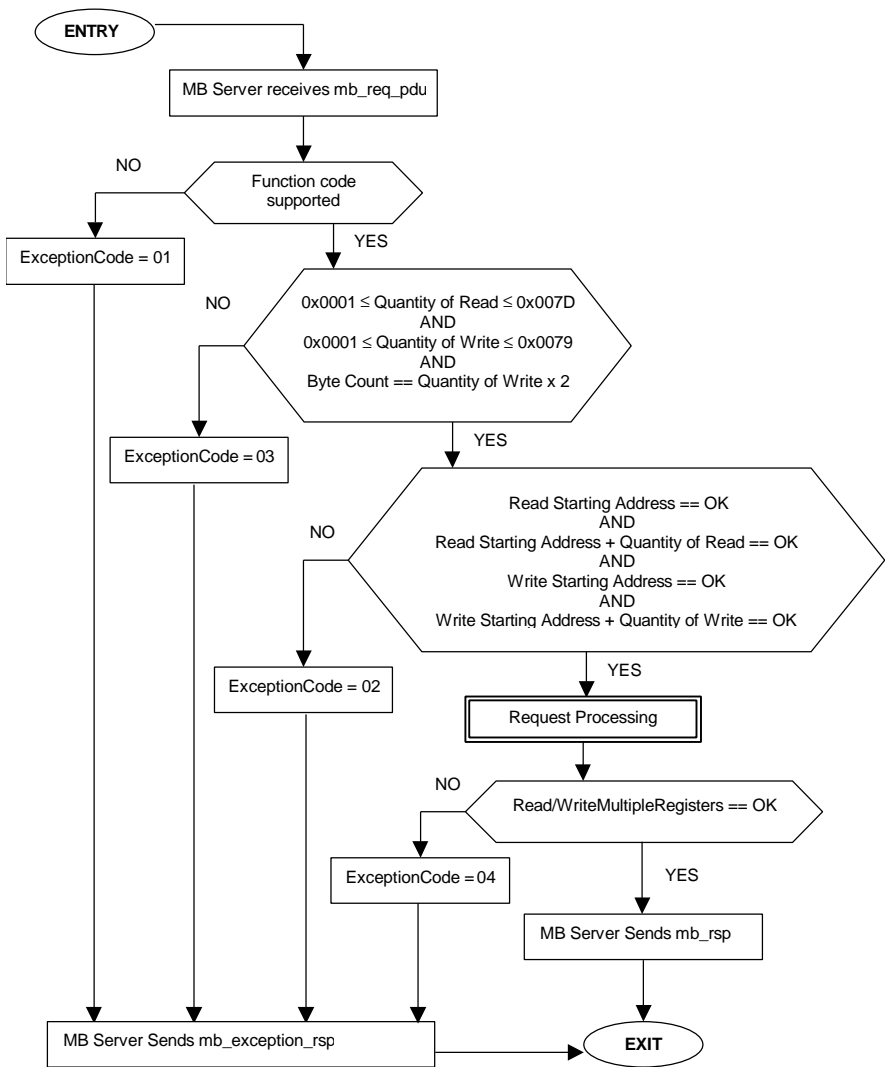


Figure 21: Read/Write Multiple Registers state diagram

## 6.12 43 (0x2B) Read Device Identification

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

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. All of these data are device dependent.

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

### Request PDU

Function code	1 Byte	<b>0x2B</b>
MEI Type	1 Byte	0x0E
Read Device ID code	1 Byte	01 / 02 / 03 / 04
Object Id	1 Byte	0x00 to 0xFF

### Response PDU

Function code	1 Byte	<b>0x2B</b>
MEI Type	1 byte	0x0E
Read Device ID code	1 Byte	01 / 02 / 03 / 04
Conformity level	1 Byte	
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	1 Byte	

**Error**

Function code	1 Byte	<b>0xAB :</b> <b>Fc 0x2B + 0x80</b>
MEI Type	1 Byte	14
Exception code	1 Byte	01, 02, 03, 04

**Request parameters description :**

A Modbus Encapsulated Interface assigned number 14 identifies the Read identification request. Four access types are defined :

- 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)

In the case where the identification data does not fit into a single response, several request/response transactions may be required. 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 start 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.

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.

If the Object Id doesn't match to any known object, the server returns an exception response with exception code = 02 (Illegal data address).

**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 01 : basic identification (stream access only) 02 : regular identification (stream access only) 03 : extended identification (stream access only) 81 : basic identification (stream access and individual access) 82 : regular identification (stream access and individual access) 83 : extended identification (stream access and individual access)
More Follows	<b><i>In case of ReadDevId codes 01, 02 or 03 (stream access),</i></b> If the identification data doesn't fit into a single response, several request/response transactions may be required. 00 : no more Object are available FF : other identification Object are available and further Modbus transactions are required <b><i>In case of ReadDevId code 04 (individual access),</i></b> 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	0A
		Object Value	" Product code "
		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 initiated.

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	1A
		Object Value	" Product code XXXXXXXXXXXXXXXXXX"

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"

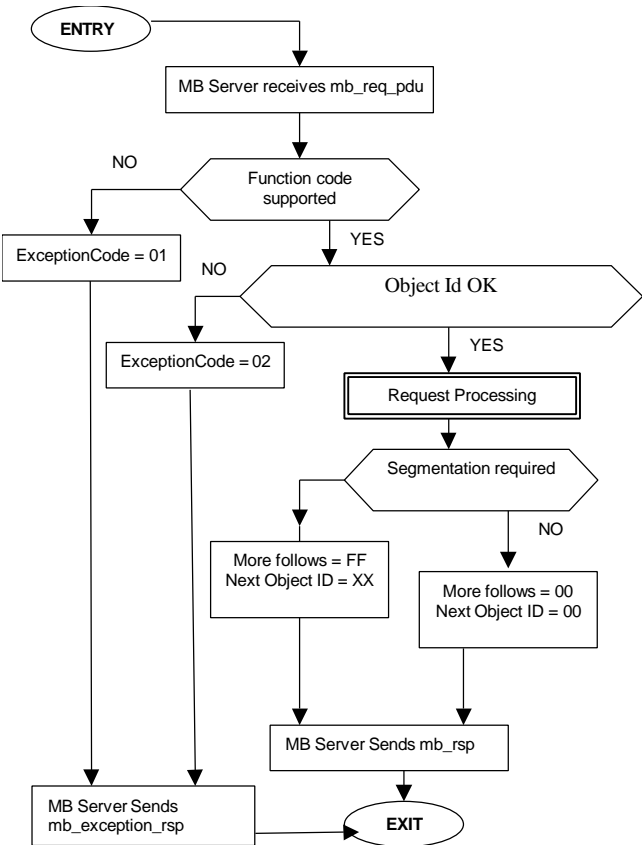


Figure 22: Read Device Identification state diagram

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 master'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 1245 (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 slave.

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 (or slave). This may be because the function code is only applicable to newer devices, and was not implemented in the unit selected. It could also indicate that the server (or slave) is in the wrong state to process a request of this type, for example because it is unconfigured and is being asked to return register values.
02	ILLEGAL DATA ADDRESS	The data address received in the query is not an allowable address for the server (or slave). More specifically, the combination of reference number and transfer length is invalid. For a controller with 100 registers, a request with offset 96 and length 4 would succeed, a request with offset 96 and length 5 will generate exception 02.
03	ILLEGAL DATA VALUE	A value contained in the query data field is not an allowable value for server (or slave). 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	SLAVE DEVICE FAILURE	An unrecoverable error occurred while the server (or slave) was attempting to perform the requested action.
05	ACKNOWLEDGE	Specialized use in conjunction with programming commands. The server (or slave) has accepted the request and is processing it, but a long duration of time will be required to do so. This response is returned to prevent a timeout error from occurring in the client (or master). The client (or master) can next issue a Poll Program Complete message to determine if processing is completed.
06	SLAVE DEVICE BUSY	Specialized use in conjunction with programming commands. The server (or slave) is engaged in processing a long-duration program command. The client (or master) should retransmit the message later when the server (or slave) 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 (or slave) attempted to read record file, but detected a parity error in the memory. The client (or master) can retry the request, but service may be required on the server (or slave) 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.

