MODBUS APPLICATION PROTOCOL SPECIFICATION
V1.1b3

CONTENTS

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.

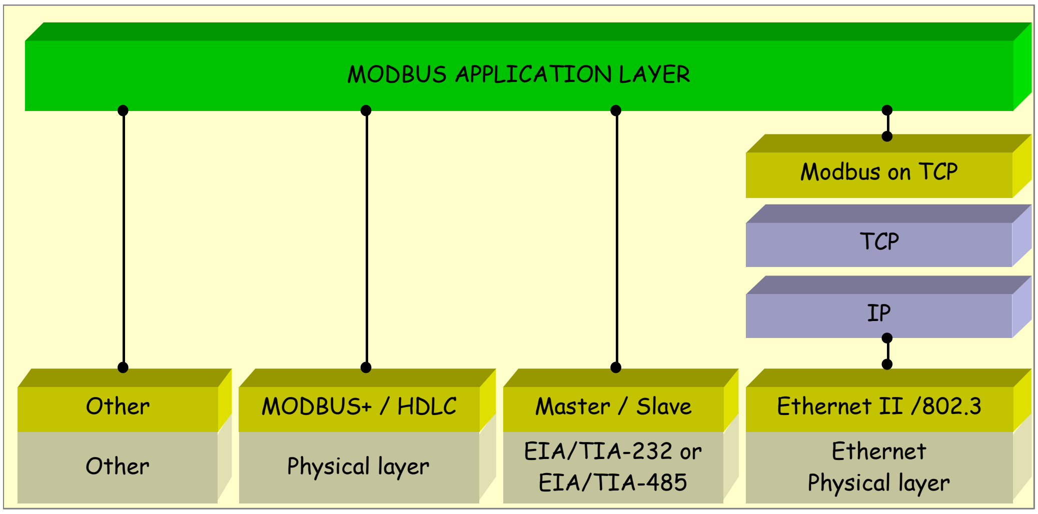


Figure 1: MODBUS communication stack

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.

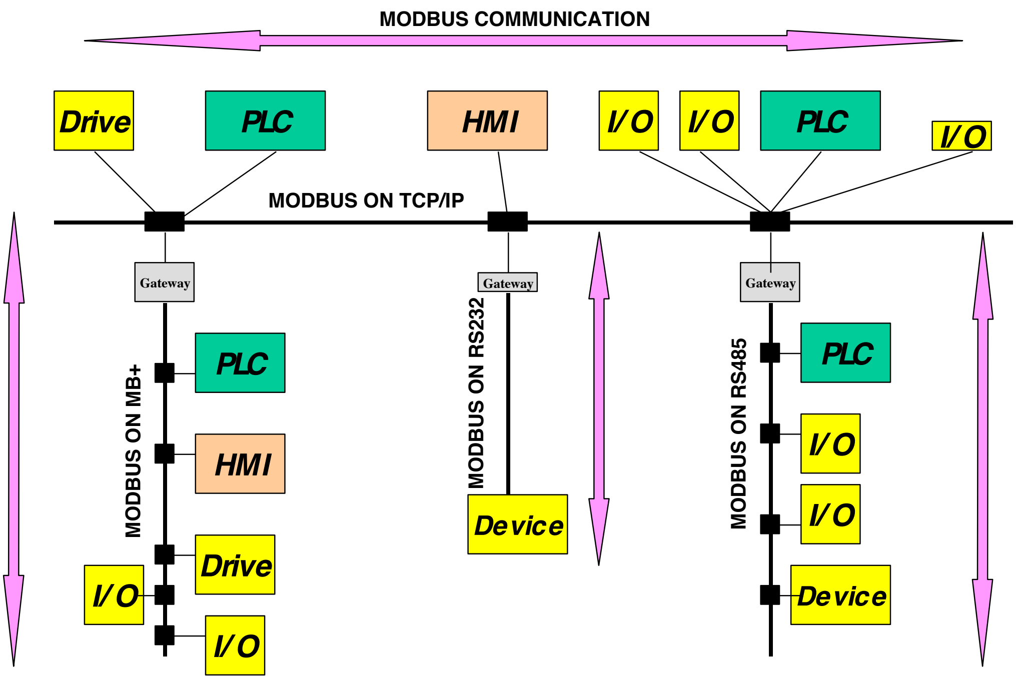


Figure 2: Example of MODBUS Network Architecture

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)**.

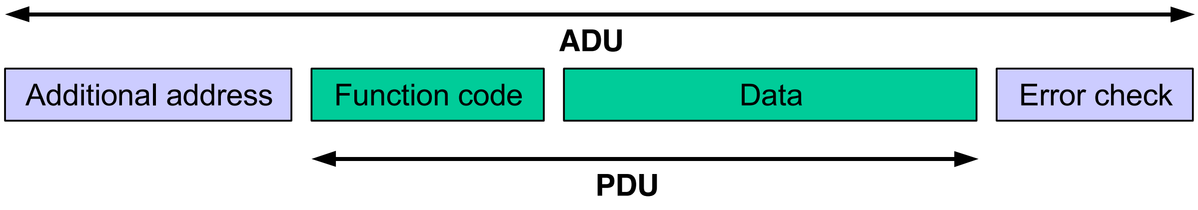


Figure 3: General MODBUS frame

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.

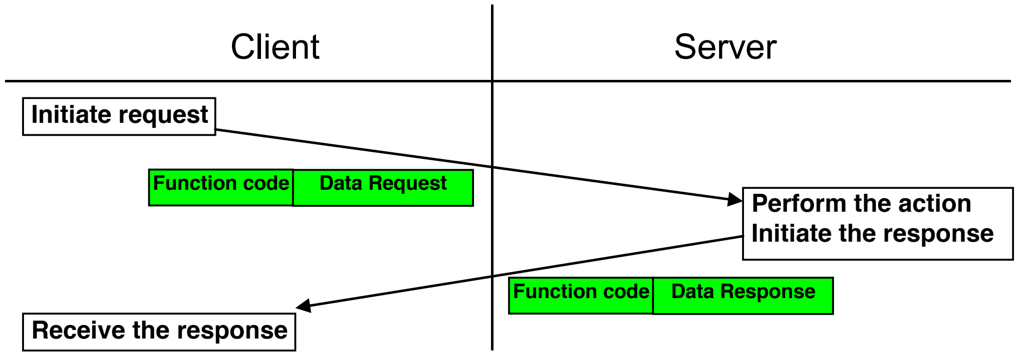


Figure 4: MODBUS transaction (error free)

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.

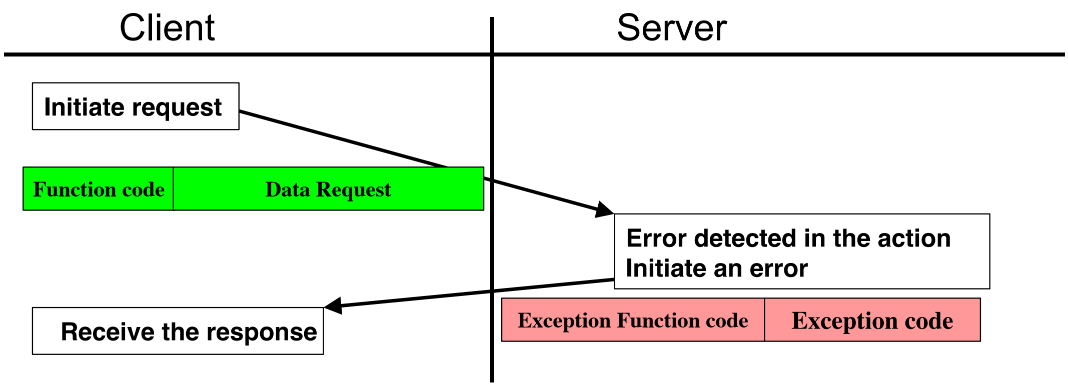


Figure 5: MODBUS transaction (exception response)

☞ **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.

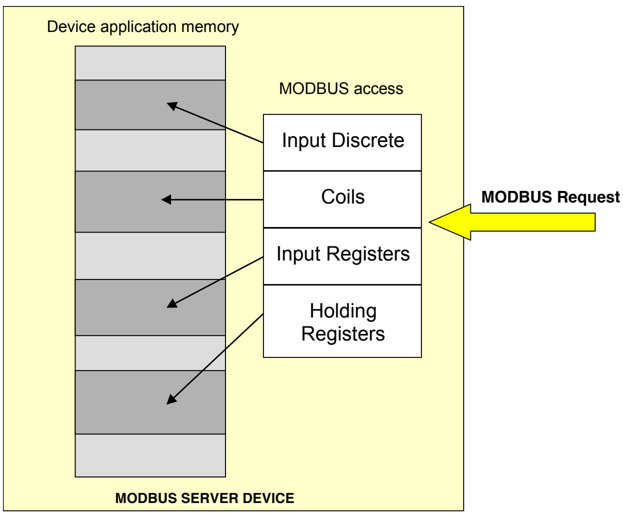


Figure 6 MODBUS Data Model with separate block

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.

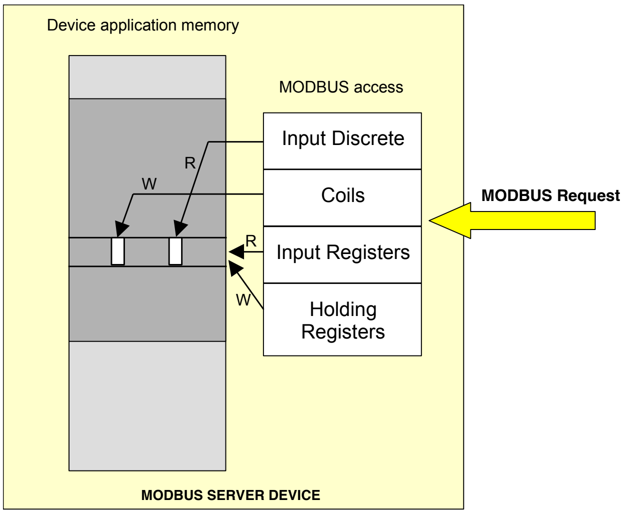


Figure 7 MODBUS Data Model with only 1 block

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.

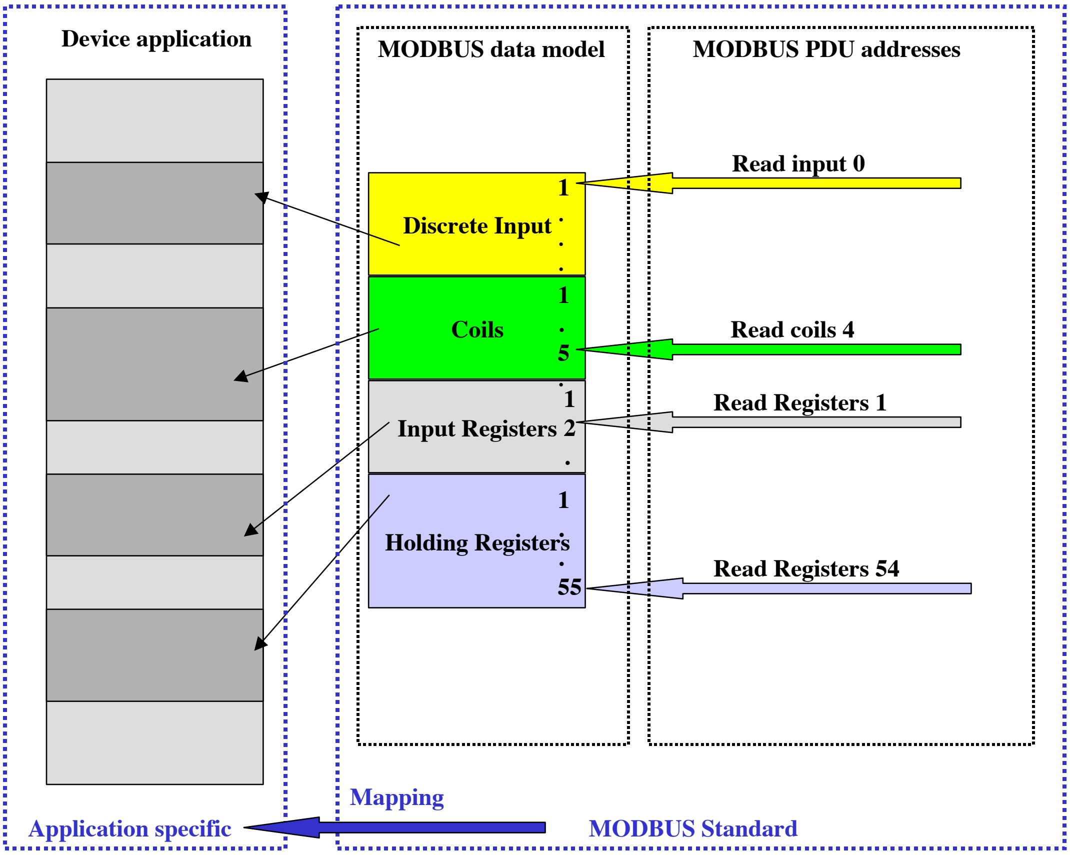


Figure 8 MODBUS Addressing model

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

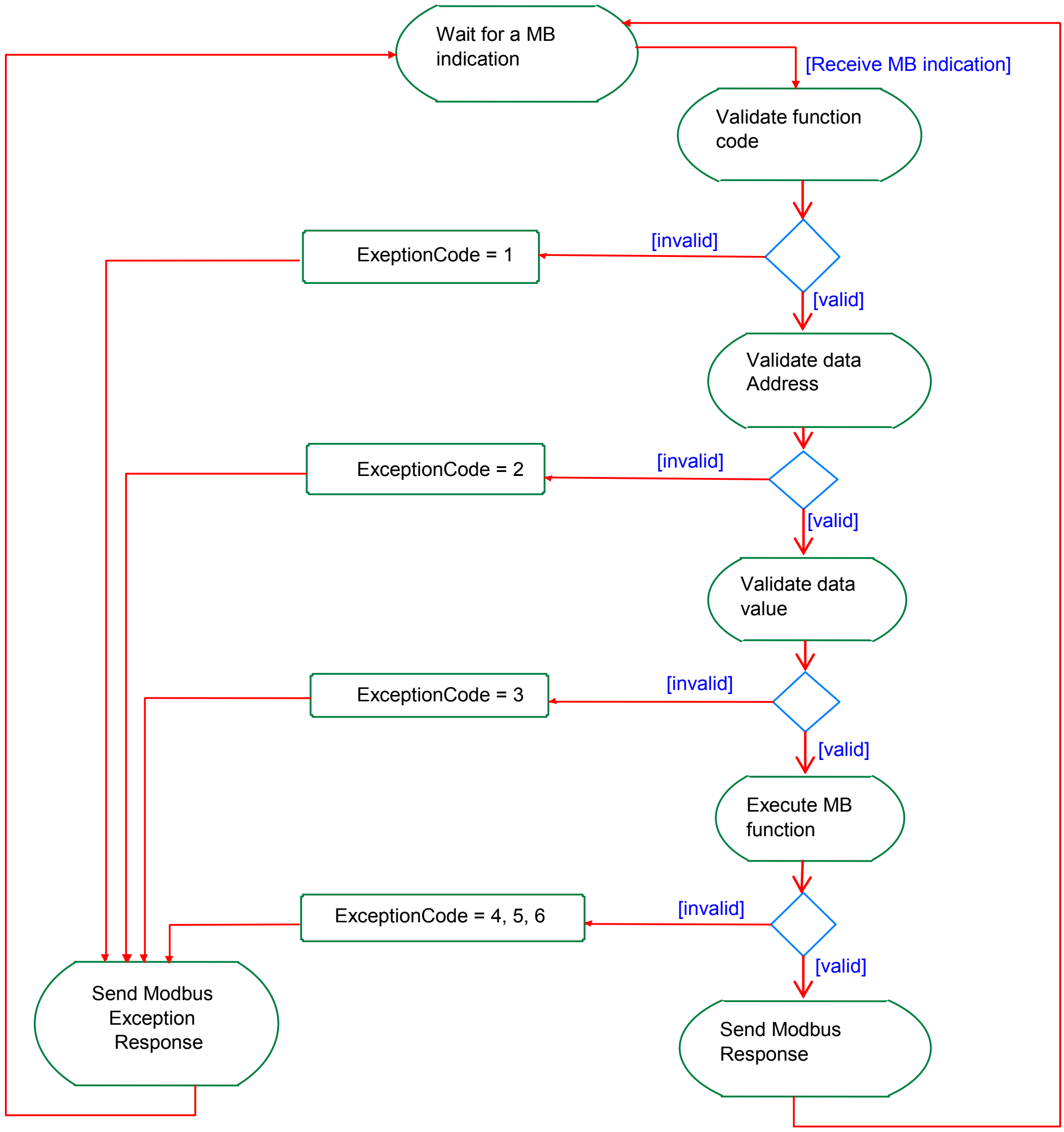


Figure 9 MODBUS Transaction state diagram

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.

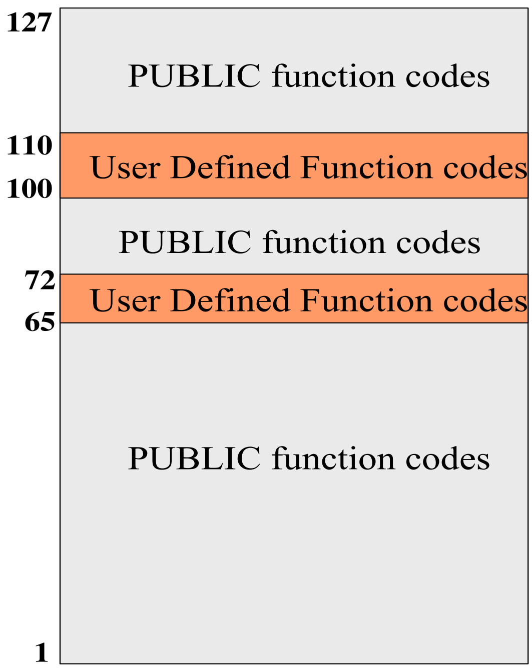


Figure 10 MODBUS Function Code Categories

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.

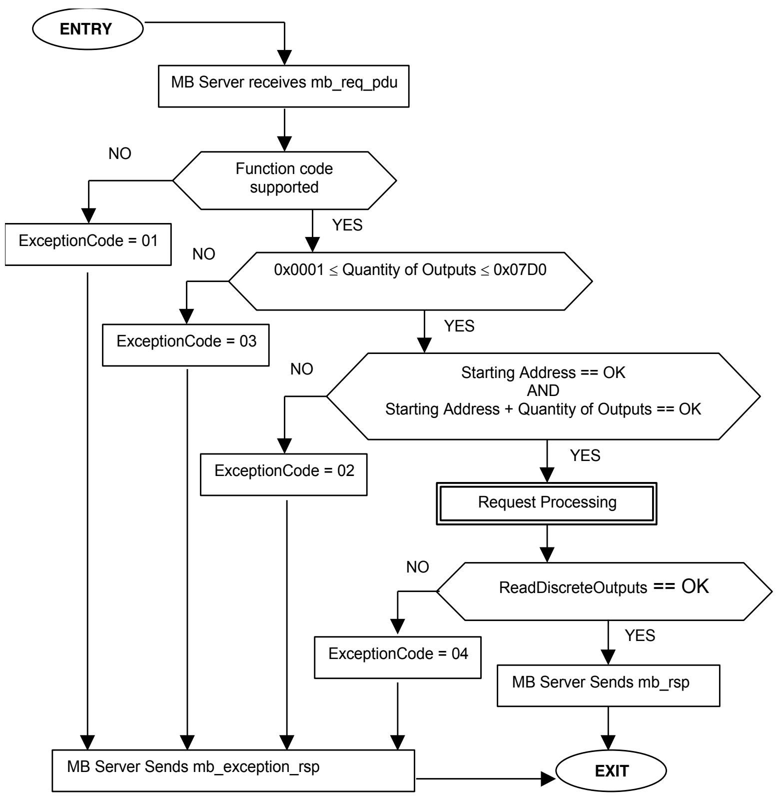


Figure 11: Read Coils state diagram

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.

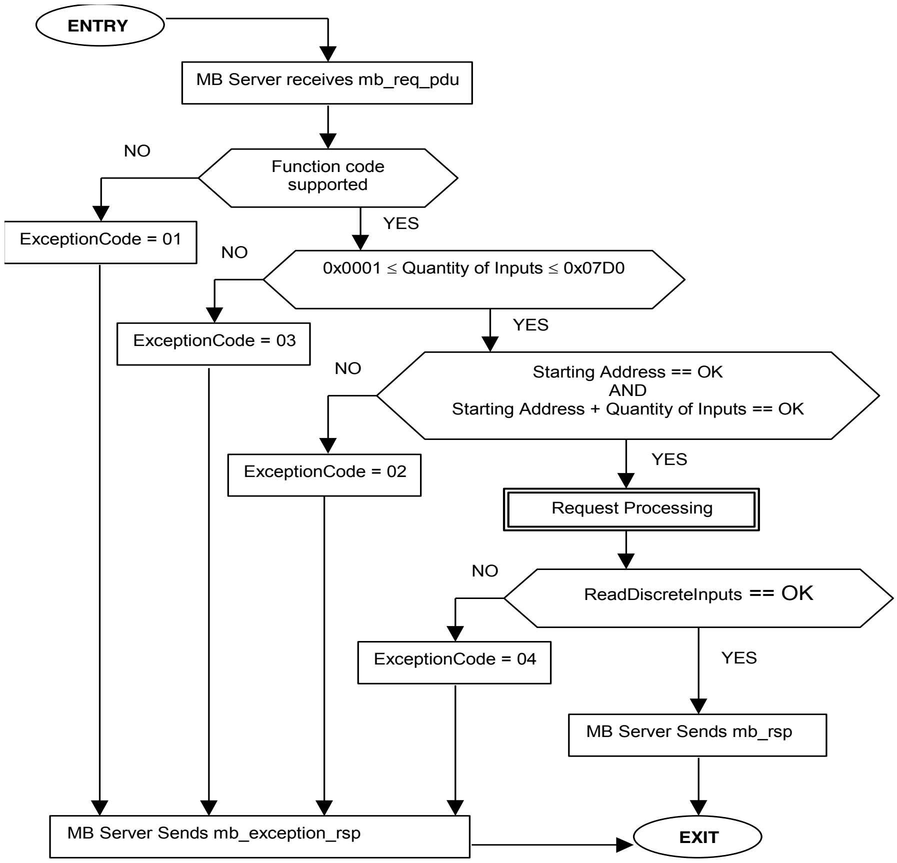


Figure 12: Read Discrete Inputs state diagram

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.

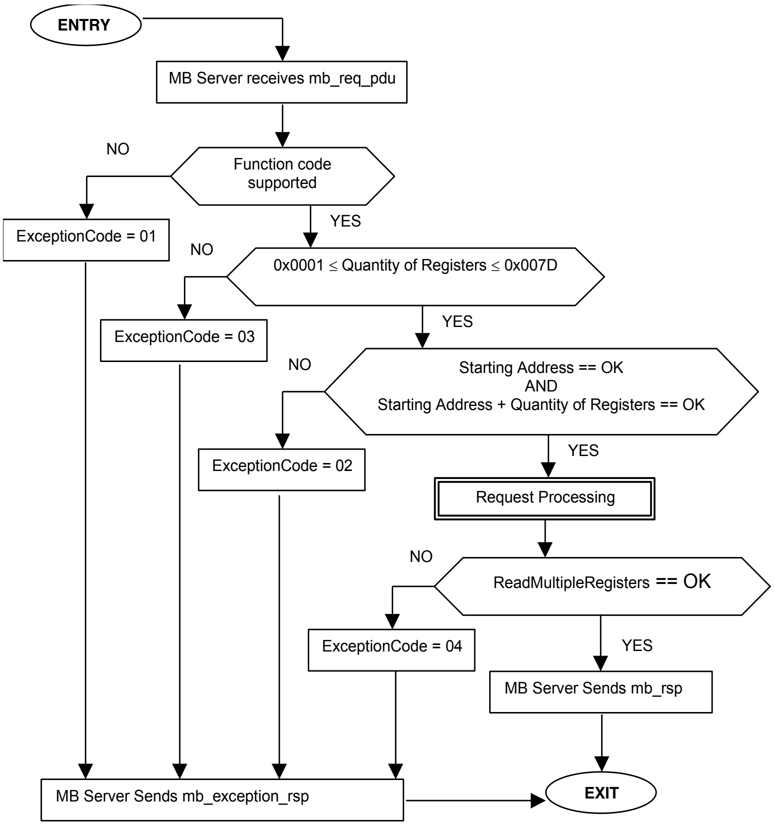


Figure 13: Read Holding Registers state diagram

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.

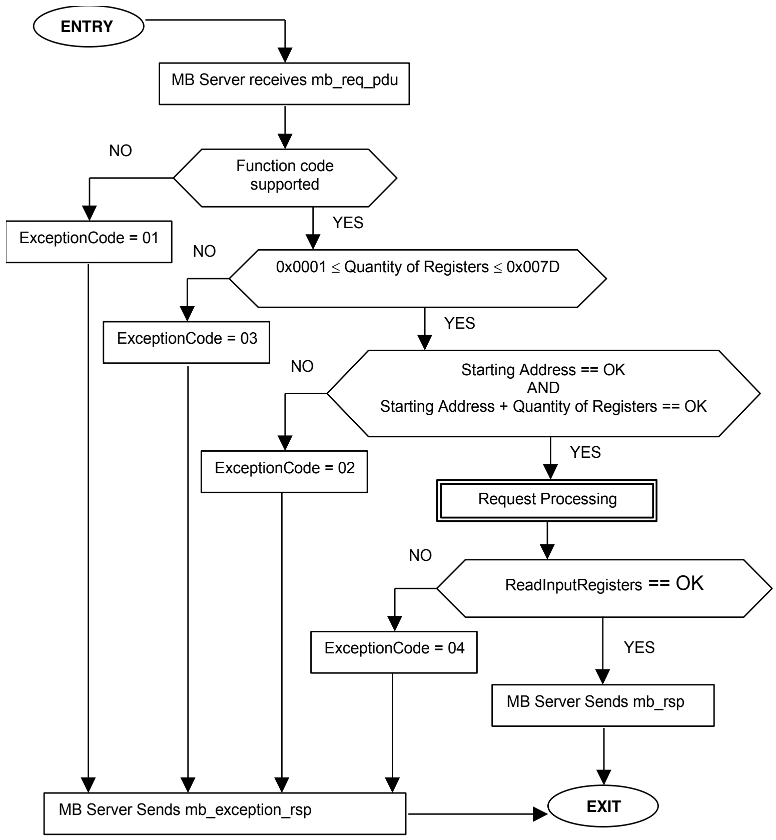


Figure 14: Read Input Registers state diagram

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**

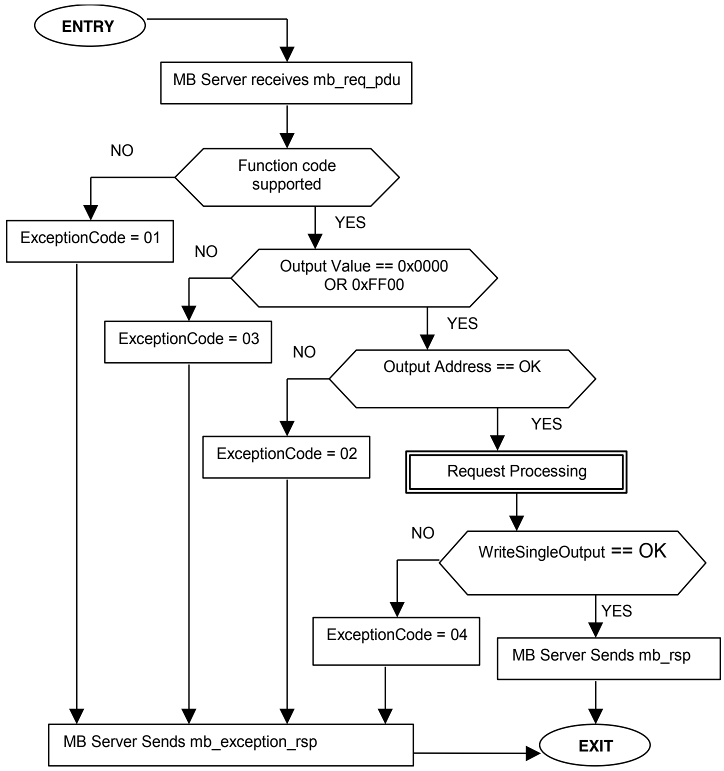


Figure 15: Write Single Output state diagram

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**

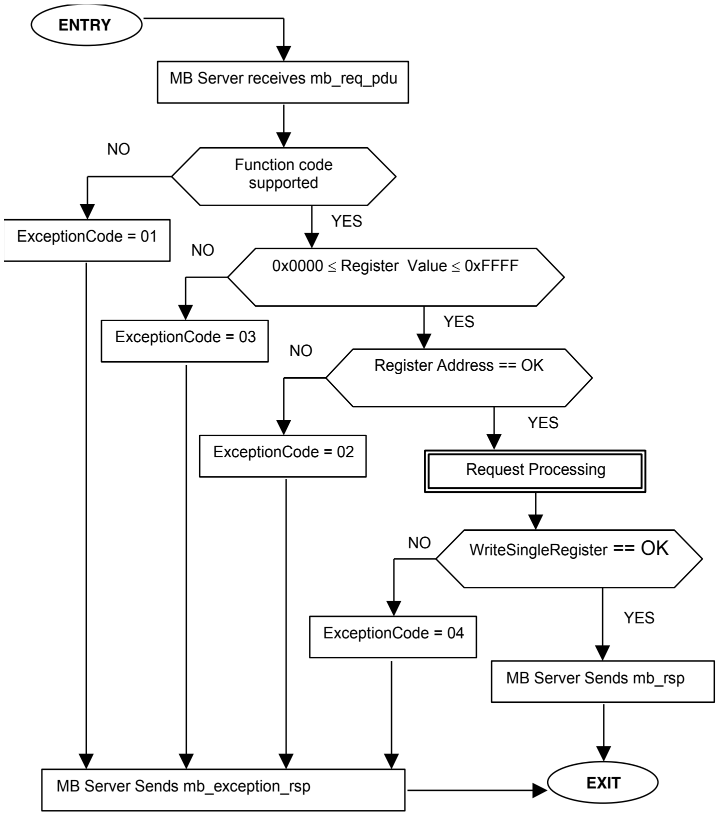


Figure 16: Write Single Register state diagram

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.

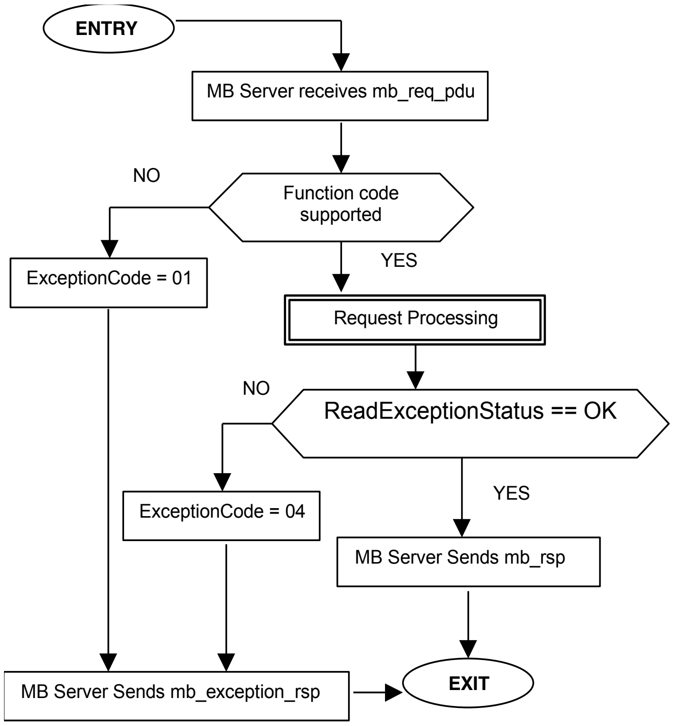


Figure 17: Read Exception Status state diagram

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.

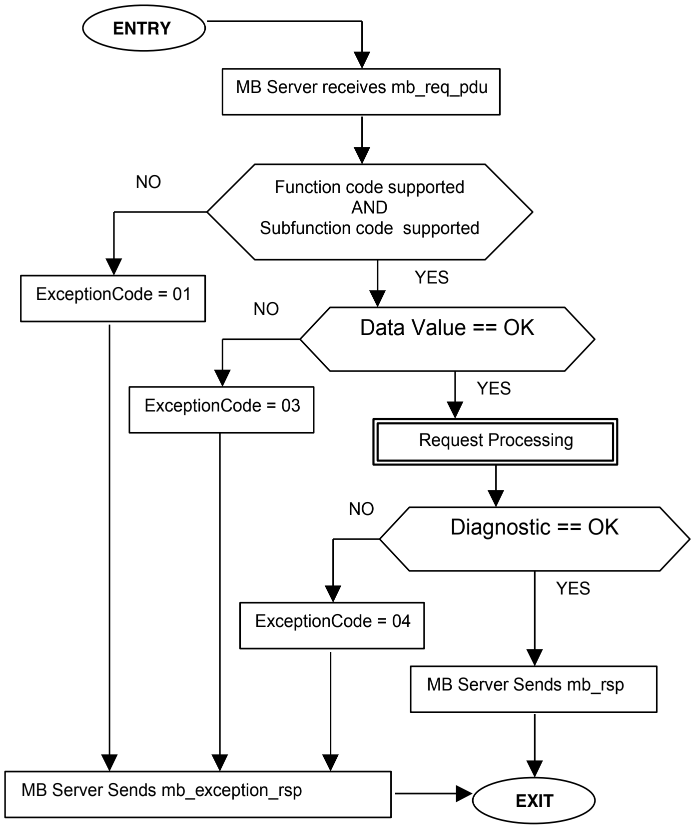


Figure 18: Diagnostic state diagram

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.

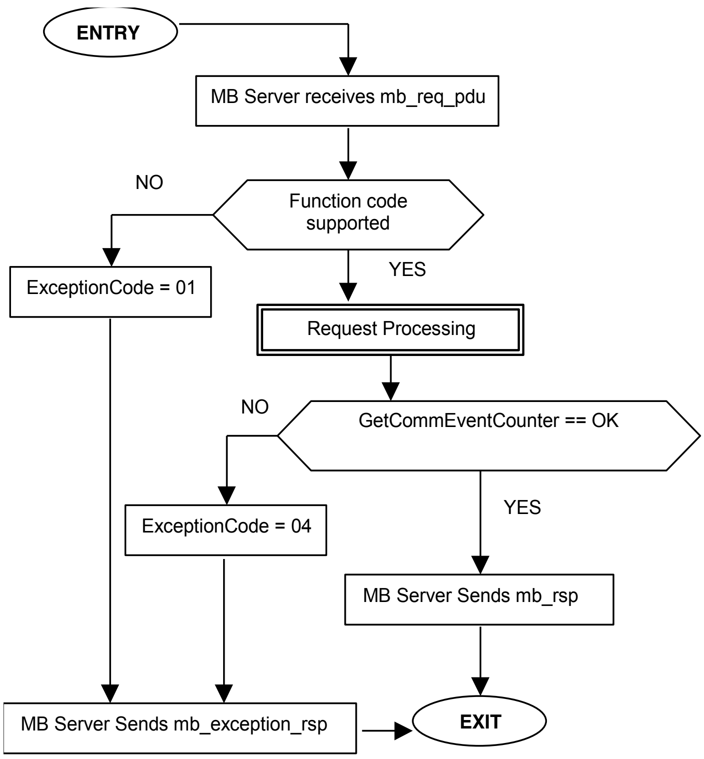


Figure 19: Get Comm Event Counter state diagram

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.

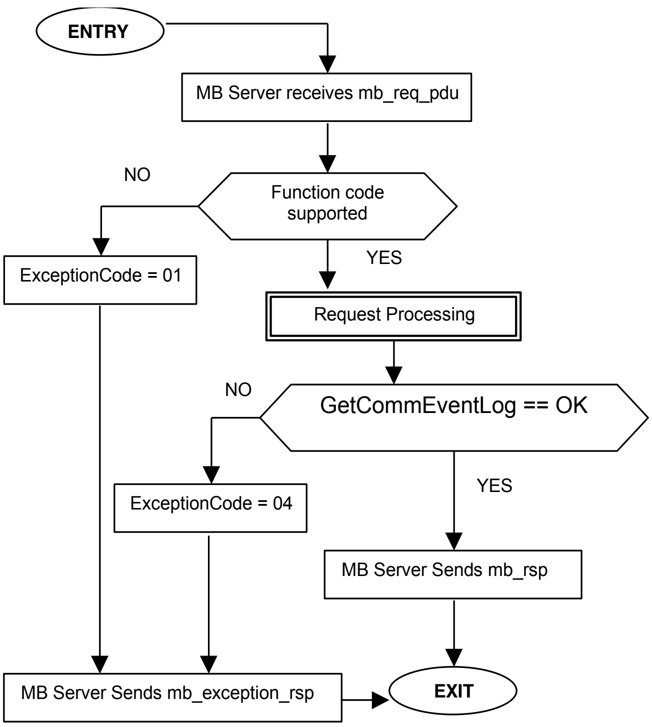


Figure 20: Get Comm Event Log state diagram

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**		

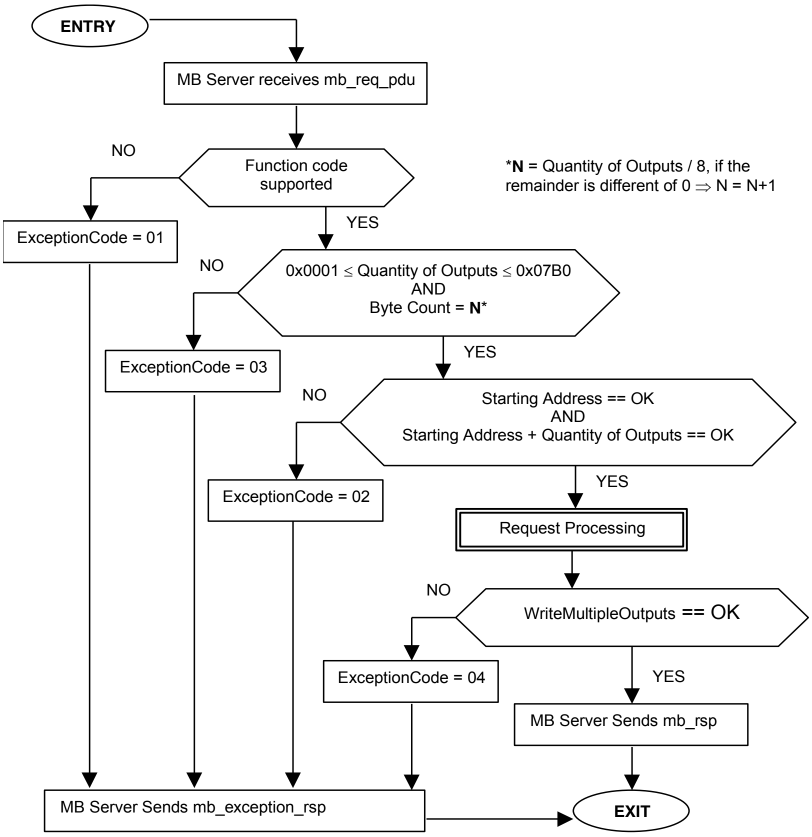


Figure 21: Write Multiple Outputs state diagram

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		

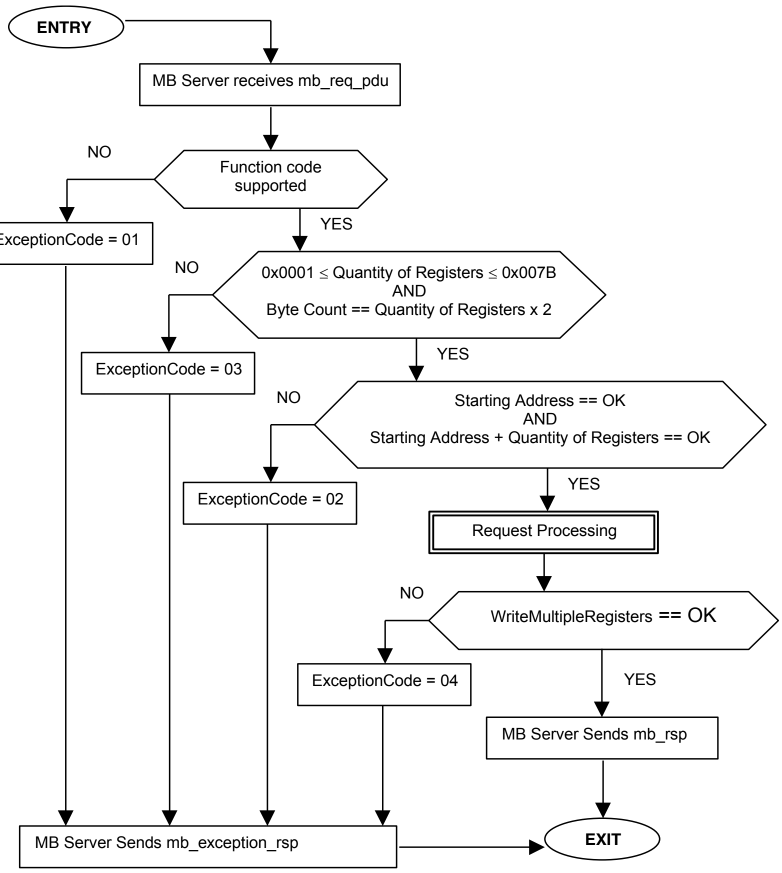


Figure 22: Write Multiple Registers state diagram

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

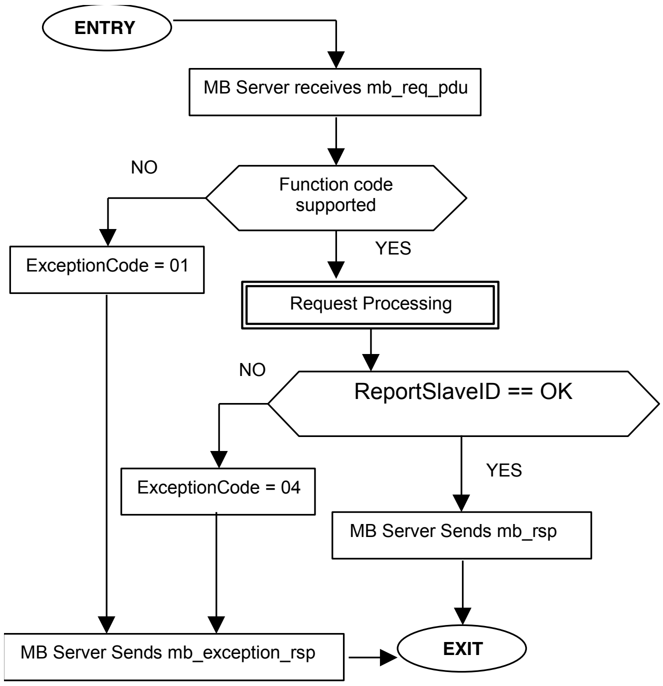


Figure 23: Report server ID state diagram

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**		

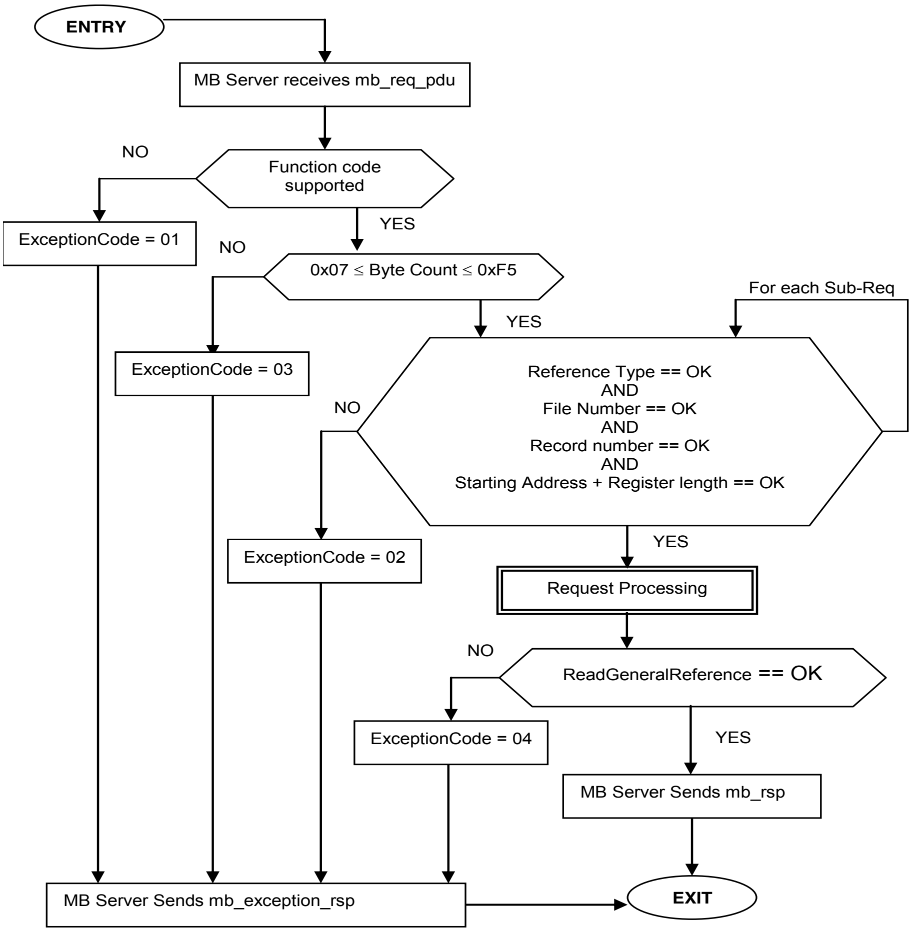


Figure 24: Read File Record state diagram

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**

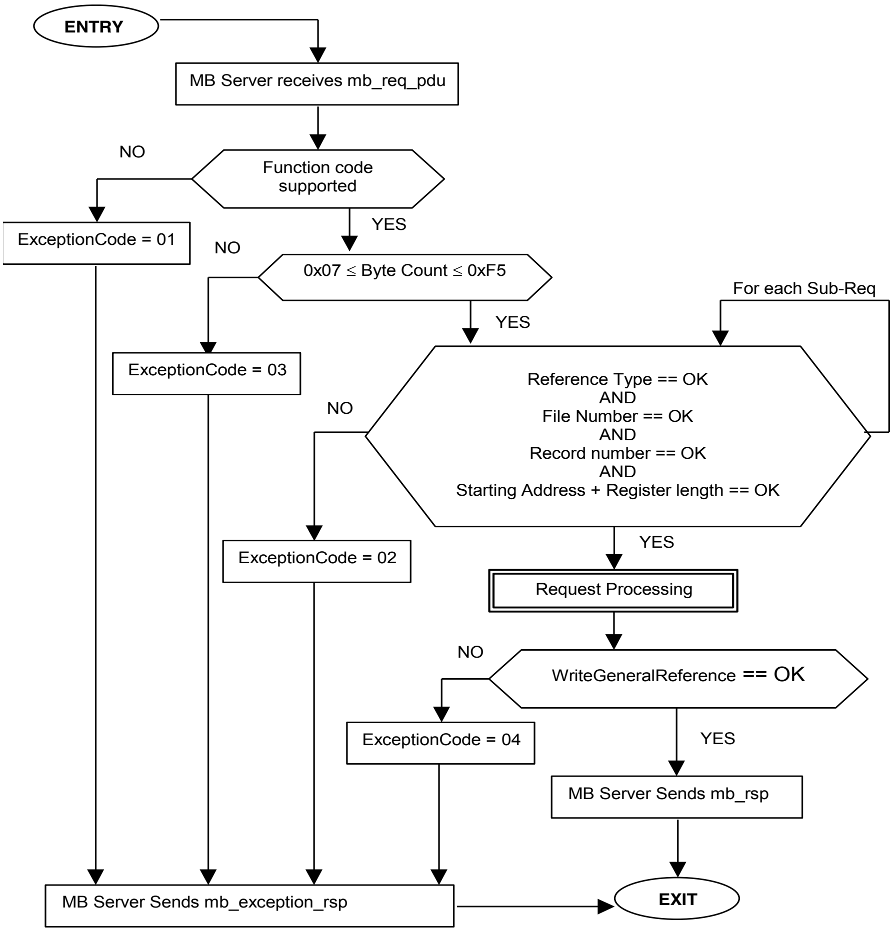


Figure 25: Write File Record state diagram

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**

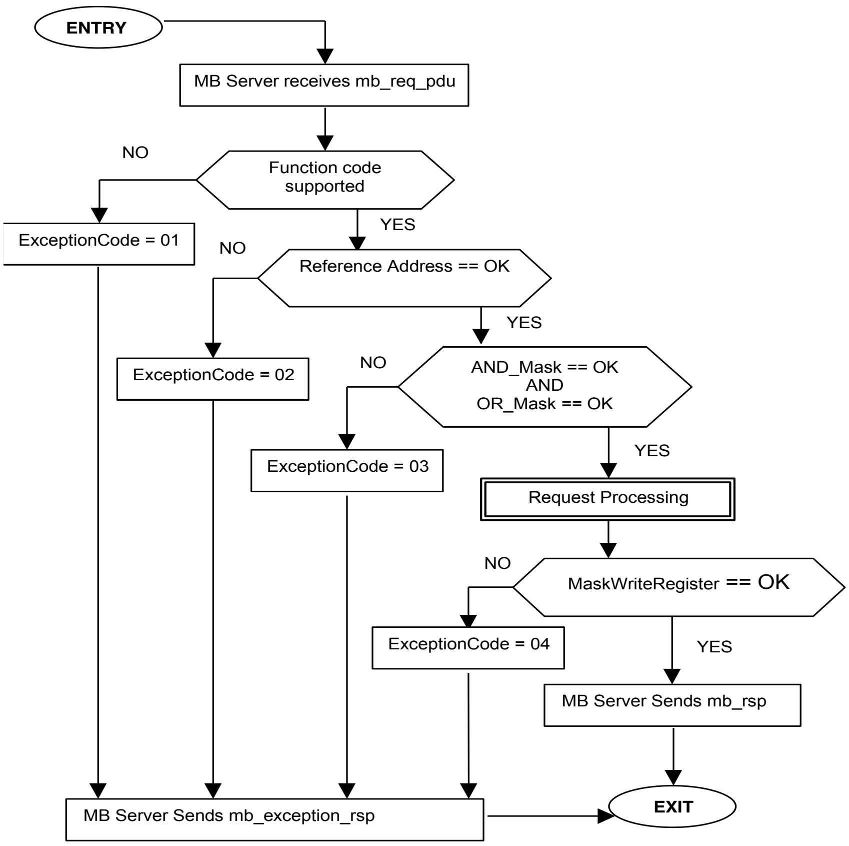


Figure 26: Mask Write Holding Register state diagram

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**		

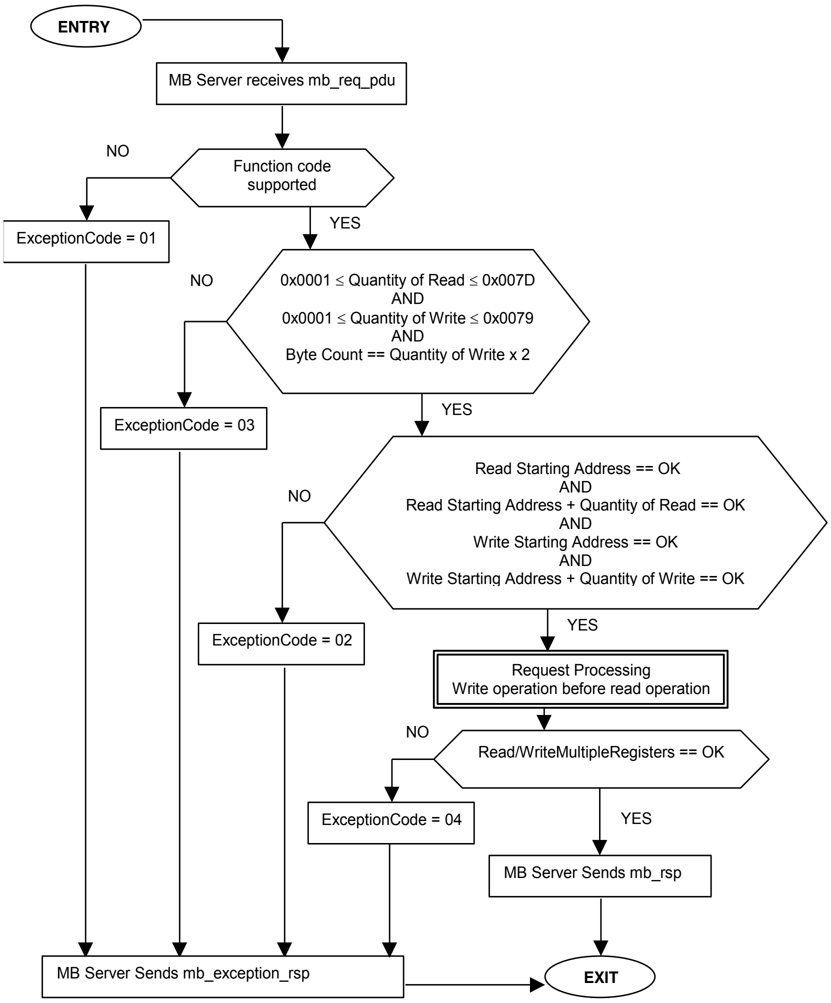


Figure 27: Read/Write Multiple Registers state diagram

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).

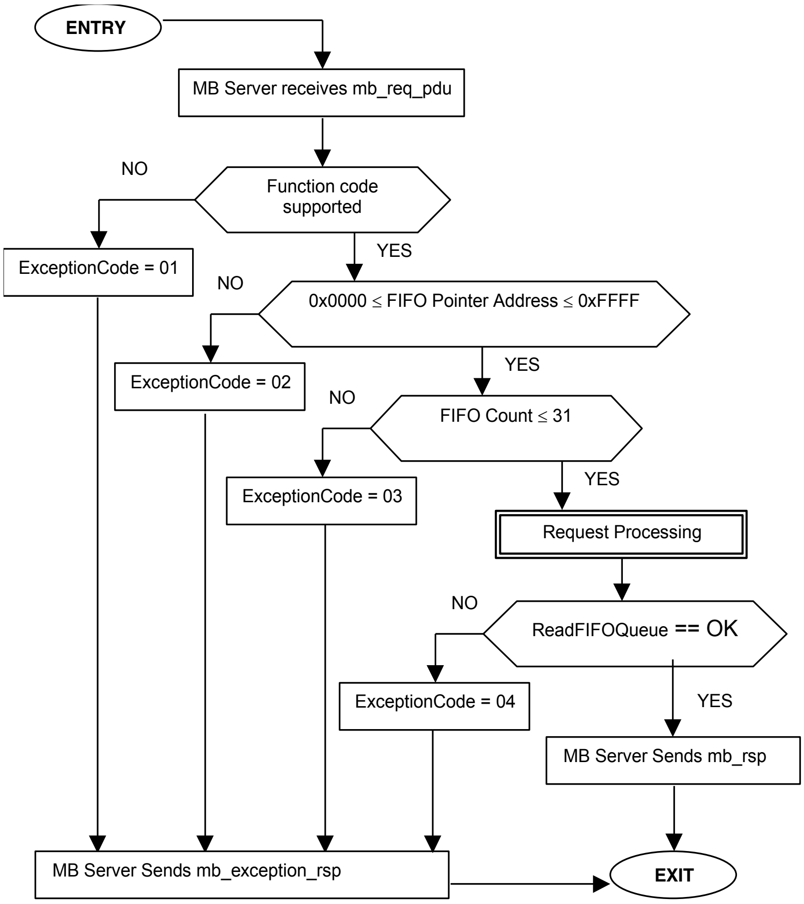


Figure 28: Read FIFO Queue state diagram

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.

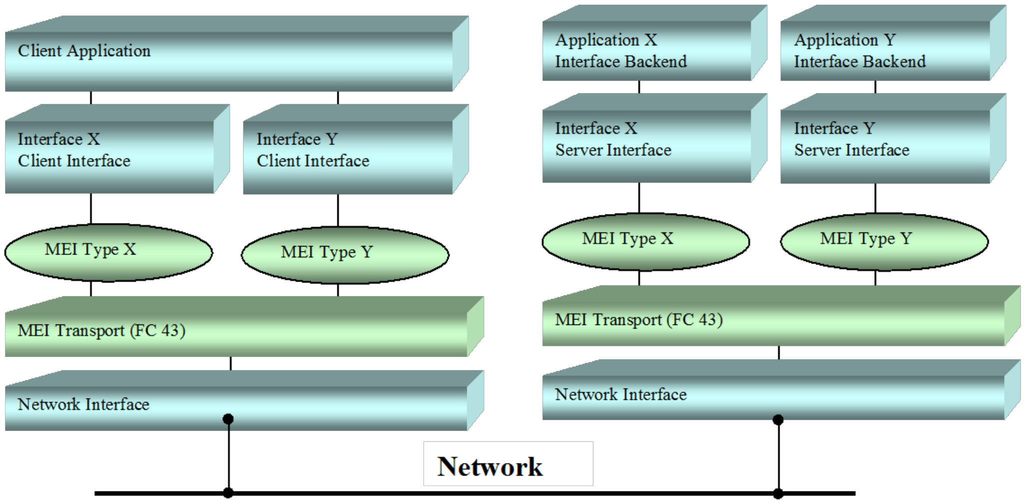


Figure 29: MODBUS encapsulated Interface Transport

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"**

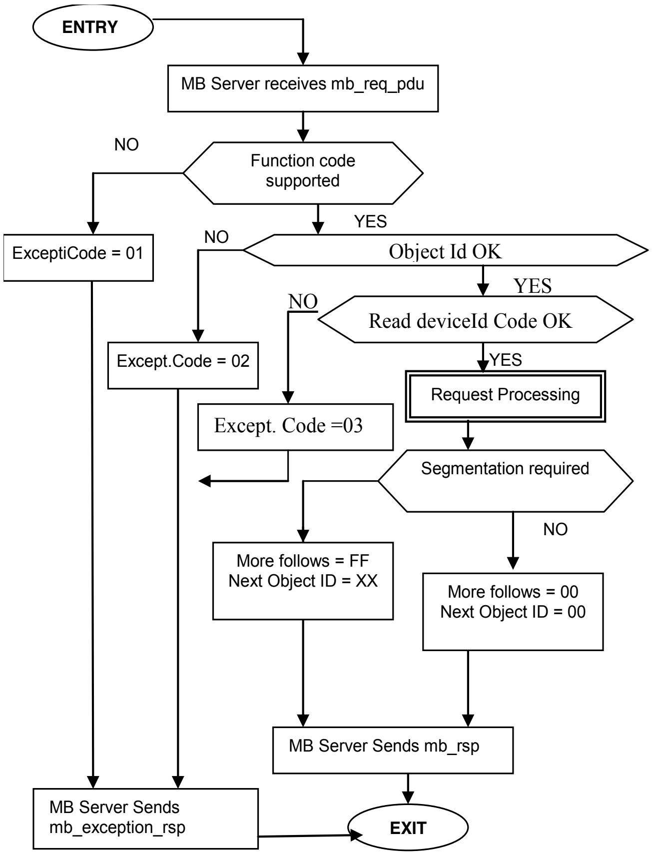


Figure 30: Read Device Identification state diagram

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.

MODBUS MESSAGING ON TCP/IP IMPLEMENTATION GUIDE V1.0b

CONTENTS

1 INTRODUCTION

1.1 OBJECTIVES

The objective of this document is to present the MODBUS messaging service over TCP/IP , in order to provide reference information that helps software developers to implement this service. The encoding of the MODBUS function codes is not described in this document, for this information please read the MODBUS Application Protocol Specification [1].

This document gives accurate and comprehensive description of a MODBUS messaging service implementation. Its purpose is to facilitate the interoperability between the devices using the MODBUS messaging service.

This document comprises mainly three parts:

- An overview of the MODBUS over TCP/IP protocol
- A functional description of a MODBUS client, server and gateway implementation.
- An implementation guideline that proposes the object model of an MODBUS implementation example.

1.2 CLIENT / SERVER MODEL

The MODBUS messaging service provides a Client/Server communication between devices connected on an Ethernet TCP/IP network.
This client / server model is based on four type of messages:

- **MODBUS Request,**
- **MODBUS Confirmation,**
- **MODBUS Indication,**
- **MODBUS Response**

A MODBUS Request is the message sent on the network by the Client to initiate a transaction,

A MODBUS Indication is the Request message received on the Server side,

A MODBUS Response is the Response message sent by the Server,

A MODBUS Confirmation is the Response Message received on the Client side

The MODBUS messaging services (Client / Server Model) are used for real time information exchange:

- between two device applications,
- between device application and other device,
- between HMI/SCADA applications and devices,
- between a PC and a device program providing on line services.

-

1.3 REFERENCE DOCUMENTS

This section gives a list of documents that are interesting to read before this one:

[1] MODBUS Application Protocol Specification V1.1a.
[2] RFC 1122 Requirements for Internet Hosts -- Communication Layers

2 ABBREVIATIONS

ADU Application Data Unit
IETF Internet Engineering Task Force
IP Internet Protocol
MAC Medium Access Control
MB MODBUS
MBAP MODBUS Application Protocol
PDU Protocol Data Unit
PLC Programmable Logic Controller
TCP Transport Control Protocol
BSD Berkeley Software Distribution
MSL Maximum Segment Lifetime

3 CONTEXT

3.1 PROTOCOL DESCRIPTION

3.1.1 General communication architecture

A communicating system over MODBUS TCP/IP may include different types of device:

- A MODBUS TCP/IP Client and Server devices connected to a TCP/IP network
- The Interconnection devices like bridge, router or gateway for interconnection between the TCP/IP network and a serial line sub-network which permit connections of MODBUS Serial line Client and Server end devices.

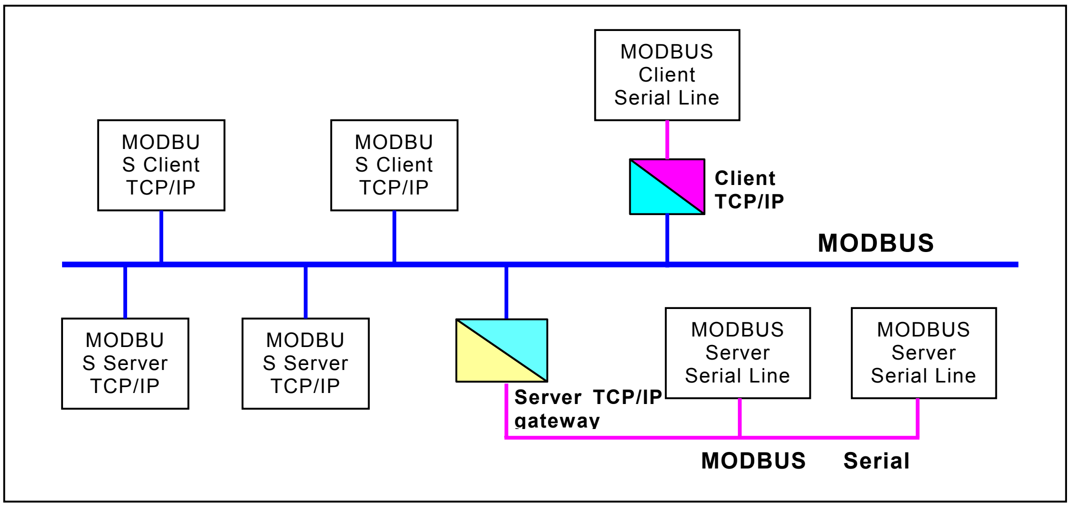


Figure 1: MODBUS TCP/IP communication architecture

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 networks can introduce some additional fields on the **Application Data Unit (ADU)**.

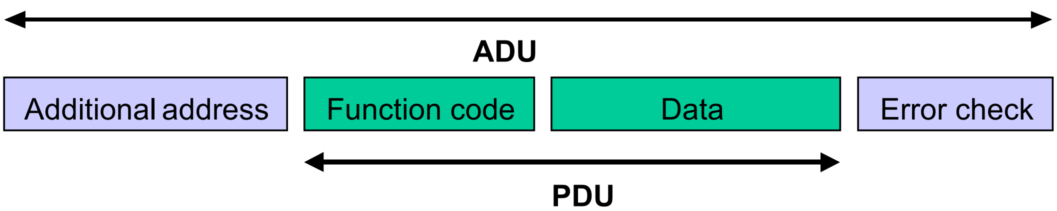


Figure 2: General MODBUS frame

The client that initiates a MODBUS transaction builds the MODBUS Application Data Unit. The function code indicates to the server which kind of action to perform.

3.1.2 MODBUS On TCP/IP Application Data Unit

This section describes the encapsulation of a MODBUS request or response when it is carried on a MODBUS TCP/IP network.

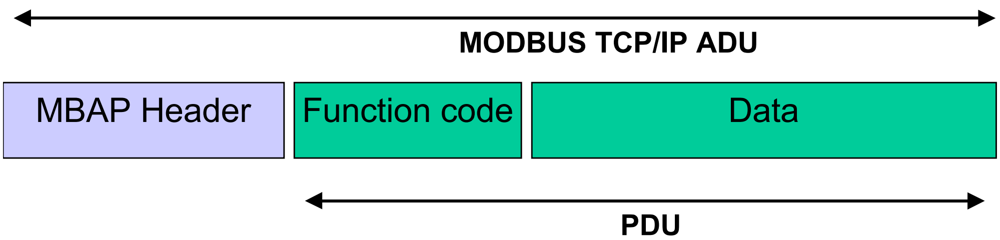


Figure 3: MODBUS request/response over TCP/IP

A dedicated header is used on TCP/IP to identify the MODBUS Application Data Unit. It is called the MBAP header (MODBUS Application Protocol header).

This header provides some differences compared to the MODBUS RTU application data unit used on serial line:

- The MODBUS 'slave address' field usually used on MODBUS Serial Line is replaced by a single byte 'Unit Identifier' within the MBAP Header. The 'Unit Identifier' is used to communicate via devices such as bridges, routers and gateways that use a single IP address to support multiple independent MODBUS end units.
- All MODBUS requests and responses are designed in such a way that the recipient can verify that a message is finished. For function codes where the MODBUS PDU has a fixed length, the function code alone is sufficient. For function codes carrying a variable amount of data in the request or response, the data field includes a byte count.
- When MODBUS is carried over TCP, additional length information is carried in the MBAP header to allow the recipient to recognize message boundaries even if the message has been split into multiple packets for transmission. The existence of explicit and implicit length rules, and use of a CRC-32 error check code (on Ethernet) results in an infinitesimal chance of undetected corruption to a request or response message.

3.1.3 MBAP Header description

The MBAP Header contains the following fields:

Fields	Length	Description -	Client	Server
Transaction Identifier	2 Bytes	Identification of a MODBUS Request / Response transaction.	Initialized by the client	Recopied by the server from the received request
Protocol Identifier	2 Bytes	0 = MODBUS protocol	Initialized by the client	Recopied by the server from the received request
Length	2 Bytes	Number of following bytes	Initialized by the client (request)	Initialized by the server (Response)
Unit Identifier	1 Byte	Identification of a remote slave connected on a serial line or on other buses.	Initialized by the client	Recopied by the server from the received request

The header is 7 bytes long:

- **Transaction Identifier** - It is used for transaction pairing, the MODBUS server copies in the response the transaction identifier of the request.
- **Protocol Identifier** – It is used for intra-system multiplexing. The MODBUS protocol is identified by the value 0.
- **Length** - The length field is a byte count of the following fields, including the Unit Identifier and data fields.

- **Unit Identifier** – This field is used for intra-system routing purpose. It is typically used to communicate to a MODBUS+ or a MODBUS serial line slave through a gateway between an Ethernet TCP-IP network and a MODBUS serial line. This field is set by the MODBUS Client in the request and must be returned with the same value in the response by the server.

All MODBUS/TCP ADU are sent via TCP to registered port 502.

Remark : the different fields are encoded in Big-endian.

3.2 MODBUS FUNCTIONS CODES DESCRIPTION

Standard function codes used on MODBUS application layer protocol are described in details in the MODBUS Application Protocol Specification [1].

4 FUNCTIONAL DESCRIPTION

The MODBUS Component Architecture presented here is a general model including both MODBUS Client and Server Components and usable on any device.

Some devices may only provide the server or the client component.

In the first part of this section a brief overview of the MODBUS messaging service component architecture is given, followed by a description of each component presented in the architectural model.

4.1 MODBUS COMPONENT ARCHITECTURE MODEL

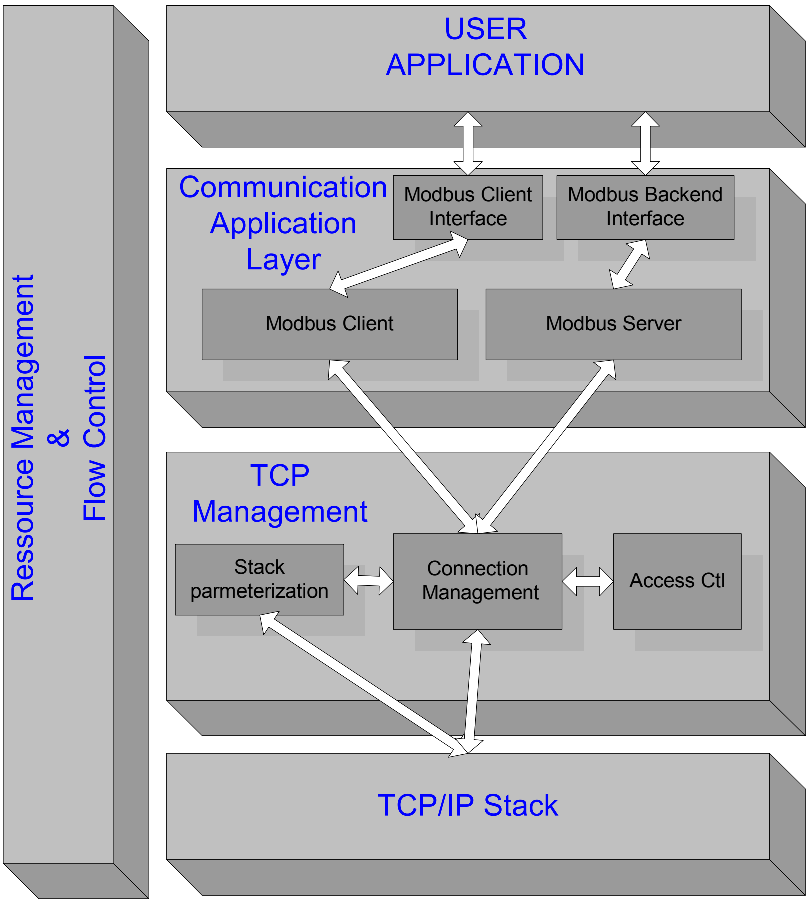


Figure 4: MODBUS Messaging Service Conceptual Architecture

- **Communication Application Layer**

A MODBUS device may provide a client and/or a server MODBUS interface.

A MODBUS backend interface can be provided allowing indirectly the access to user application objects.
Four areas can compose this interface: input discrete, output discrete (coils), input registers and output registers. A pre-mapping between this interface and the user application data has to be done (local issue).

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.

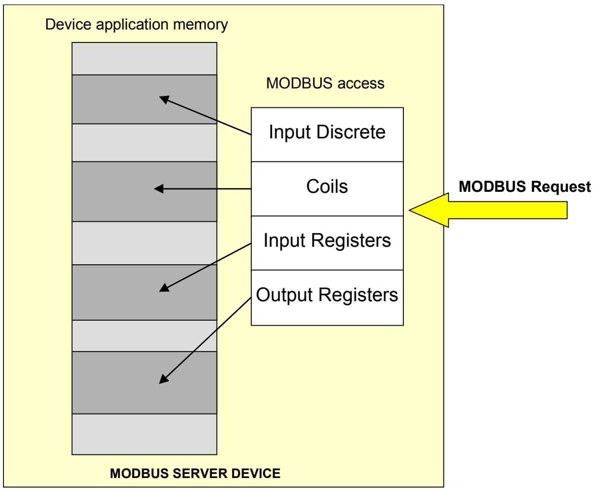


Figure 5 MODBUS Data Model with separate blocks

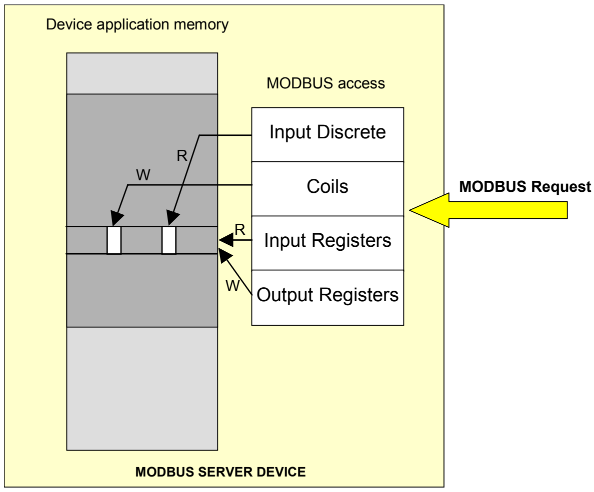


Figure 6 MODBUS Data Model with only 1 block

➢ MODBUS Client

The MODBUS Client allows the user application to explicitly control information exchange with a remote device. The MODBUS Client builds a MODBUS request from parameter contained in a demand sent by the user application to the MODBUS Client Interface.
The MODBUS Client uses a MODBUS transaction whose management includes waiting for and processing of a MODBUS confirmation.

➢ MODBUS Client Interface

The MODBUS Client Interface provides an interface enabling the user application to build the requests for various MODBUS services including access to MODBUS application objects. The MODBUS Client interface (API) is not part of this Specification, although an example is described in the implementation model.

➢ MODBUS Server

On reception of a MODBUS request this module activates a local action to read, to write or to achieve some other actions. The processing of these actions is done totally transparently for the application programmer. The main MODBUS server functions are to wait for a MODBUS request on 502 TCP port, to treat this request and then to build a MODBUS response depending on device context.

➢ MODBUS Backend Interface

The MODBUS Backend Interface is an interface from the MODBUS Server to the user application in which the application objects are defined.

Informative Note: The Backend Interface is not defined in this Specification

- **TCP Management layer**

Informative Note: The TCP/IP discussion in this Specification is based in part upon reference [2] RFC 1122 to assist the user in implementing the MODBUS Application Protocol Specification [1] over TCP/IP.

One of the main functions of the messaging service is to manage communication establishment and ending and to manage the data flow on established TCP connections.

➢ Connection Management

A communication between a client and server MODBUS Module requires the use of a TCP connection management module. It is in charge to manage globally messaging TCP connections.

Two possibilities are proposed for the connection management. Either the user application itself manages TCP connections or the connection management is totally done by this module and therefore it is transparent for the user application. The last solution implies less flexibility.

The **listening TCP port 502 is reserved for MODBUS** communications. It is mandatory to listen by default on that port. However, some markets or applications might require that another port is dedicated to MODBUS over TCP. For that reason, it is highly recommended that the clients and the servers give the possibility to the user to parameterize the MODBUS over TCP port number. **It is important to note that even if another TCP server port is configured for MODBUS service in certain applications, TCP server port 502 must still be available in addition to any application specific ports.**

➢ Access Control Module

In certain critical contexts, accessibility to internal data of devices must be forbidden for undesirable hosts. That's why a security mode is needed and security process may be implemented if required.

- **TCP/IP Stack layer**

The TCP/IP stack can be parameterized in order to adapt the data flow control, the address management and the connection management to different constraints specific to a product or to a system. Generally the BSD socket interface is used to manage the TCP connections.

▪ **Resource management and Data flow control**

In order to equilibrate inbound and outbound messaging data flow between the MODBUS client and the server, data flow control mechanism is provided in all layers of MODBUS messaging stack.
The resource management and flow control module is first based on TCP internal flow control added with some data flow control in the data link layer and also in the user application level.

4.2 TCP CONNECTION MANAGEMENT

4.2.1 Connections management Module

4.2.1.1 General description

A MODBUS communication requires the establishment of a TCP connection between a Client and a Server.
The establishment of the connection can be activated either explicitly by the User Application module or automatically by the TCP connection management module.
In the first case an application-programming interface has to be provided in the user application module to manage completely the connection. This solution provides flexibility for the application programmer but it requires a good expertise on TCP/IP mechanism.
In the second case the TCP connection management is completely hidden to the user application that only sends and receives MODBUS messages. The TCP connection management module is in charge to establish a new TCP connection when it is required.
The definition of the number of TCP client and server connections is not on the scope of this document (value n in this document). Depending on the device capacities the number of TCP connections can be different.

Implementation Rules :

1) Without explicit user requirement, it is recommended to implement the automatic TCP connection management

2) It is recommended to keep the TCP connection opened with a remote device and not to open and close it for each MODBUS/TCP transaction,
Remark: However the MODBUS client must be capable of accepting a close request from the server and closing the connection. The connection can be reopened when required.

3) It is recommended for a MODBUS Client to open a minimum of TCP connections with a remote MODBUS server (with the same IP address). One connection per application could be a good choice.

4) Several MODBUS transactions can be activated simultaneously on the same TCP Connection.
Remark: If this is done then the MODBUS transaction identifier must be used to uniquely identify the matching requests and responses.

5) In case of a bi-directional communication between two remote MODBUS entities (each of them is client and server), it is necessary to open separate connections for the client data flow and for the server data flow.

6) A TCP frame must transport only one MODBUS ADU. It is advised against sending multiple MODBUS requests or responses on the same TCP PDU

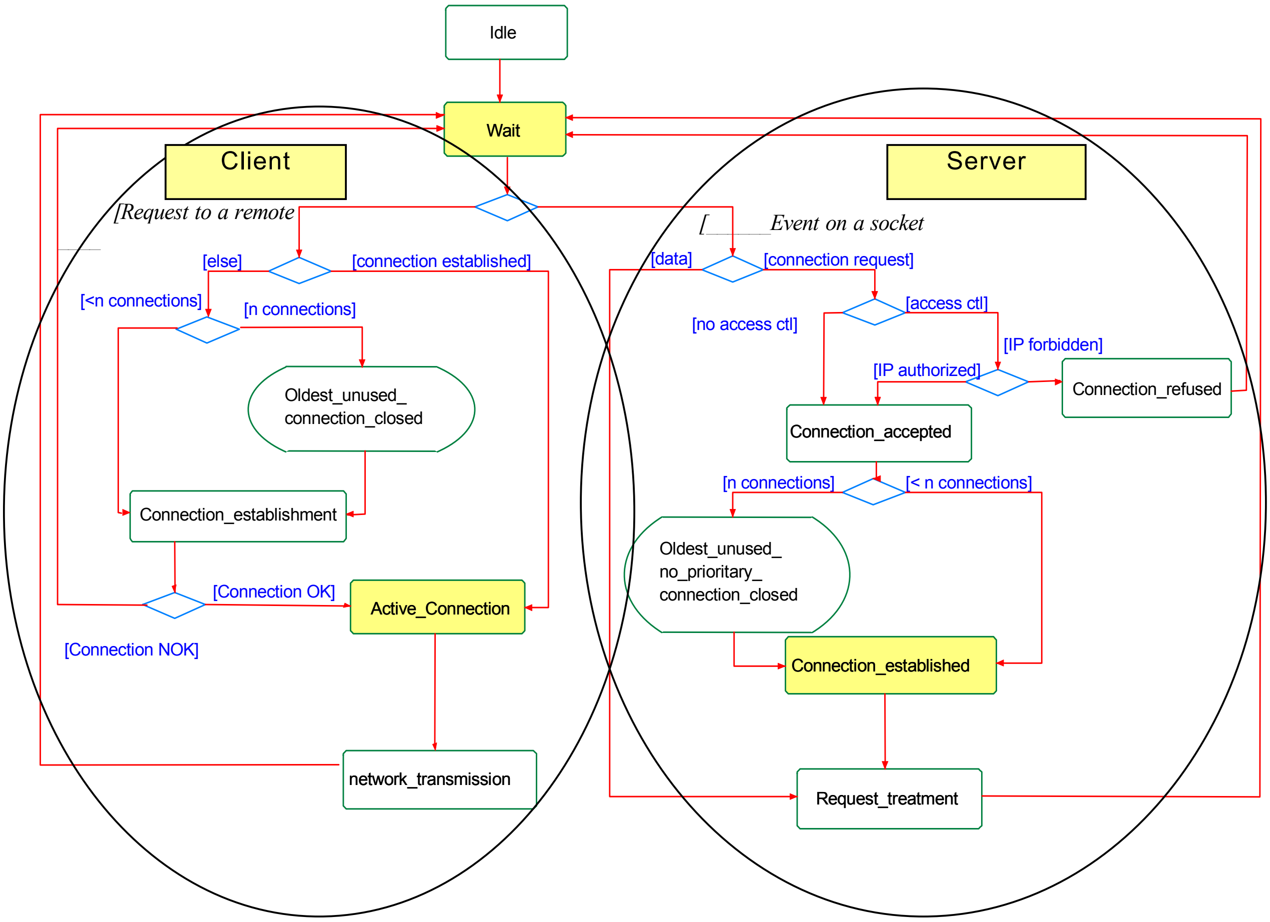


Figure 7: TCP connection management activity diagram

1. Explicit TCP connection management

The user application module is in charge of managing all the TCP connections: active and passive establishment, connection ending, etc. This management is done for all MODBUS communication between a client and a server. The BSD Socket interface is used in the user application module to manage the TCP connection. This solution offers a total flexibility but it implies that the application programmer has sufficient TCP knowledge.

A limit of number of client and server connections has to be configured taking into account the device capabilities and requirement.

2. Automatic TCP connection management

The TCP connection management is totally transparent for the user application module. The connection management module may accept a sufficient number of client and server connections.

Nevertheless a mechanism must be implemented in case of exceeding the number of authorized connection. In such a case we recommend to close the oldest unused connection.

A connection with a remote partner is established at the first packet received from a remote client or from the local user application. This connection will be closed if a termination arrived from the network or decided locally on the device. On reception of a connection request, the access control option can be used to forbid device accessibility to unauthorized clients.

The TCP connection management module uses the Stack interface (usually BSD Socket interface) to communicate with the TCP/IP stack.

In order to maintain compatibility between system requirements and server resources, the TCP management will maintain 2 pools of connection.

- The first pool (**priority connection pool**) is made of connections that are never closed on a local initiative. A configuration must be provided to set this pool up. The principle to be implemented is to associate a specific IP address with each possible connection of this pool. The devices with such IP addresses are said to be "marked". Any new connection that is requested by a marked device must be accepted, and will be taken from the priority connection pool. It is also necessary to configure the maximum number of Connections allowed for each remote device to avoid that the same device uses all the connections of the priority pool.
- The second pool (**non-priority connection pool**) contains connections for non marked devices. The rule that takes over here is to close the oldest connection when a new connection request arrives from a non-marked device and when there is no more connection available in the pool.

A configuration might be optionally provided to assign the number of connections available in each pool. However (It is not mandatory) the designers can set the number of connections at design time if required.

4.2.1.2 Connection management description

- **<u>Connection establishment :</u>**

The MODBUS messaging service must provide a listening socket on Port 502, which permits to accept new connection and to exchange data with other devices.
When the messaging service needs to exchange data with a remote server, it must open a new client connection with a remote Port 502 in order to exchange data with this distant. The local port must be higher than 1024 and different for each client connection.

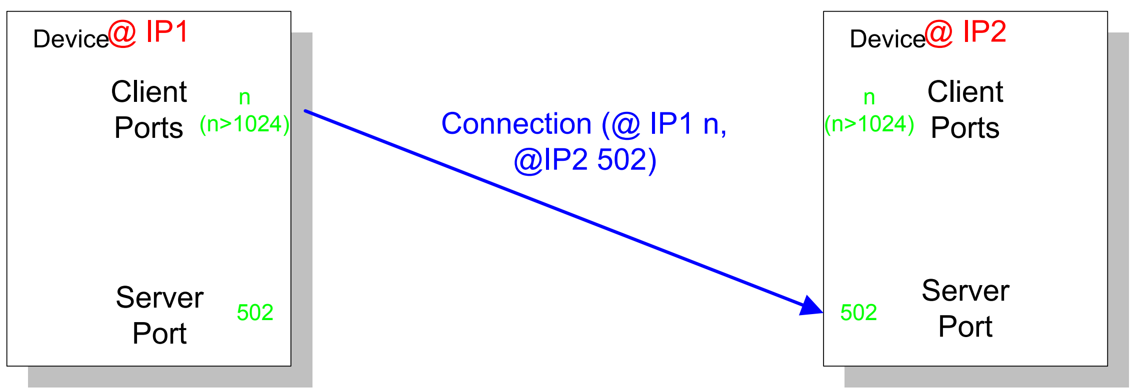


Figure 8: MODBUS TCP connection establishment

If the number of client and server connections is greater than the number of authorized connections the oldest unused connection is closed. The access control mechanism can be activated to check if the IP address of the remote client is authorized. If not the new connection is refused.

- **MODBUS data transfer**

A MODBUS request has to be sent on the right TCP connection already opened. The IP address of the remote is used to find the TCP connection. In case of multiple TCP connections opened with the same remote, one connection has to be chosen to send the MODBUS message, different choice criteria can be used like the oldest one, the first one. The connection has to maintain open during all the MODBUS communications. As described in the following sections a client can initiate several MODBUS transactions with a server without waiting the ending of the previous one.

▪ **Connection closing**

When the MODBUS communications are ended between a Client and a Server, the client has to initiate a connection closing of the connection used for these communications.

4.2.2 Impact of Operating Modes on the TCP Connection

Some Operating Modes (communication break between two operational End Points, Crash and Reboot on one of the End Point, ...) may have impacts on the TCP Connections. A connection can be seen closed or aborted on one side without the knowledge of the other side. The connection is said to be "**half-open**".
This section describes the behavior for each main Operating Modes. It is assumed that the **Keep Alive** TCP mechanism is used on both end points (See section 4.3.2)

4.2.2.1 Communication break between two operational end points:

The origin of the communication break can be the disconnection of the Ethernet cable on the Server side. The expected behavior is:

- If no packet is currently sent on the connection:
  The communication break will not be seen if it lasts less than the Keep Alive timer value. If the communication break lasts more than the Keep Alive timer value, an error is returned to the TCP Management layer that can reset the connection.
- If Some packets are sent before and after the disconnection:
  The TCP retransmission algorithms (Jacobson's, Karn's algorithms and exponential backoff. See section 4.3.2) are activated. This may lead to a stack TCP layer Reset of the Connection before the Keep Alive timer is over.

4.2.2.2 Crash and Reboot of the Server end point

After the crash and Reboot of the Server, the connection is "**half-open**" on Client side. The expected behavior is:

- If no packet is sent on the half-open connection:
  The TCP half-open connection is seen opened from the Client side as long as the Keep Alive timer is not over. After that an error is returned to the TCP Management layer that can reset the connection.
- If some packets are sent on the half-open connection:
  The Server receives data on a connection that doesn't exist anymore. The stack TCP layer sends a Reset to close the half-open connection on the Client side

4.2.2.3 Crash and Reboot of the Client

After the crash and Reboot of the Client, the connection is "**half-open**" on Server side. The expected behavior is:

- No packet is sent on the half-open connection:
  The TCP half-open connection is seen opened from the Server side as long as the Keep Alive timer is not over. After that an error is returned to the TCP Management layer that can reset the connection.
- If the Client opens a new connection before the Keep Alive timer is over :
  Two cases have to be studied:
  - The connection opening has the **same characteristics as the half-open connection** on the server side (same source and destination Ports, same source and destination IP Addresses), therefore the connection opening will fail at the TCP stack level after the Time-Out on Connection Establishment (75s on most of Berkeley implementations). To avoid this long Time-Out during which it is not possible to communicate, it is advised to ensure that different source port numbers than the previous one are used for a connection opening after a reboot on the client side.

  - The connection opening has **not the same characteristics as the half-open connection** on the server side (different source Ports, same destination Port, same source and destination IP Address), therefore the connection is opened at the stack TCP level and signaled to the Server TCP Management layer.
    If the Server TCP Management layer only supports one connection from a remote Client IP Address, it can close the old half-opened connection and use the new one.
    If the Server TCP Management layer supports several connections from a remote Client IP Address, the new connection stays opened and the old one also stays half-opened until the expiration of the Keep Alive Timer that will return an error to the TCP Management layer. After that the TCP Management layer will be able to Reset the old connection.

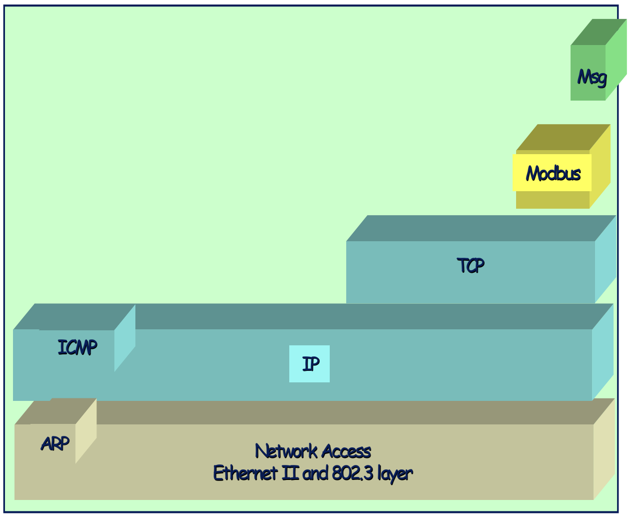
4.2.3 Access Control Module

The goal of this module is to check every new connection and using a list of authorized remote IP addresses the module can authorize or forbid a remote Client TCP connection.

In critical context the application programmer needs to choose the Access Control mode in order to secure its network access. In such a case he needs to Authorize/forbid access for each remote @IP. The user needs to provide a list of IP addresses and to specify for each IP address if it's authorized or not. By default, on security mode, the IP addresses not configured by the user are forbidden. Therefore with the access control mode a connection coming from an unknown IP address is closed.

4.3 USE of TCP/IP STACK

A TCP/IP stack provides an interface to manage connections, to send and receive data, and also to do some parameterizations in order to adapt the stack behavior to the device or system constraints.



The goal of this section is to give an overview of the Stack interface and also some information concerning the parameterization of the stack. This overview focuses on the features used by the MODBUS Messaging.

For more information, the advice is to read the RFC 1122 that provides guidance for vendors and designers of Internet communication software. It enumerates standard protocols that a host connected to the Internet must use as well as an explicit set of requirements and options.
The stack interface is generally based on the BSD (Berkeley Software Distribution) Interface that is described in this document.
.

4.3.1 Use of BSD Socket interface

Remark : some TCP/IP stacks propose other types of interfaces for performance issues. A MODBUS client or server can use these specific interfaces, but this use will be not described in this specification.

A socket is an endpoint of communication. It is the basic building block for communication. A MODBUS communication is executed by sending and receiving data through sockets. The TCPIP library provides only stream sockets using TCP and providing a connection-based communication service.
The Sockets are created via the **socket ()** function. A socket number is returned, which is then used by the creator to access the socket. Sockets are created without addresses (IP address and port number). Until a port is bound to a socket, it cannot be used to receive data.
The **bind ()** function is used to bind a port number to a socket. The **bind ()** creates an association between the socket and the port number specified.
In order to initiate a connection, the client must issue the **connect ()** function specifying the socket number, the remote IP address and the remote listening port number (active connection establishment).
In order to complete a connection, the server must issue the **accept ()** function specifying the socket number that was specified in the prior **listen ()** call (passive connection establishment). A new socket is created with the same properties as the initial one. This new socket is connected to the client's socket, and its number is returned to the server. The initial socket is thereby free for other clients that might want to connect with the server.

After the establishment of the TCP connection the data can be transferred. The **send()** and **recv()** functions are designed specifically to be used with sockets that are already connected.

The **setsockopt ()** function allows a socket's creator to associate options with a socket. These options modify the behavior of the socket. The description of these options is given in the section 4.3.2.

The **select ()** function allows the programmer to test events on all sockets.

The **shutdown ()** function allows a socket user to disable send () and/or receive () on the socket.

Once a socket is no longer needed, its socket descriptor can be discarded by using the **close ()** function.

Figure 39: MODBUS Exchanges describes a full MODBUS communication between a client and a s server. The Client establishes the connection and sends 3 MODBUS requests to the server without waiting the response of the first one. After receiving all the responses the Client closes the connection properly.

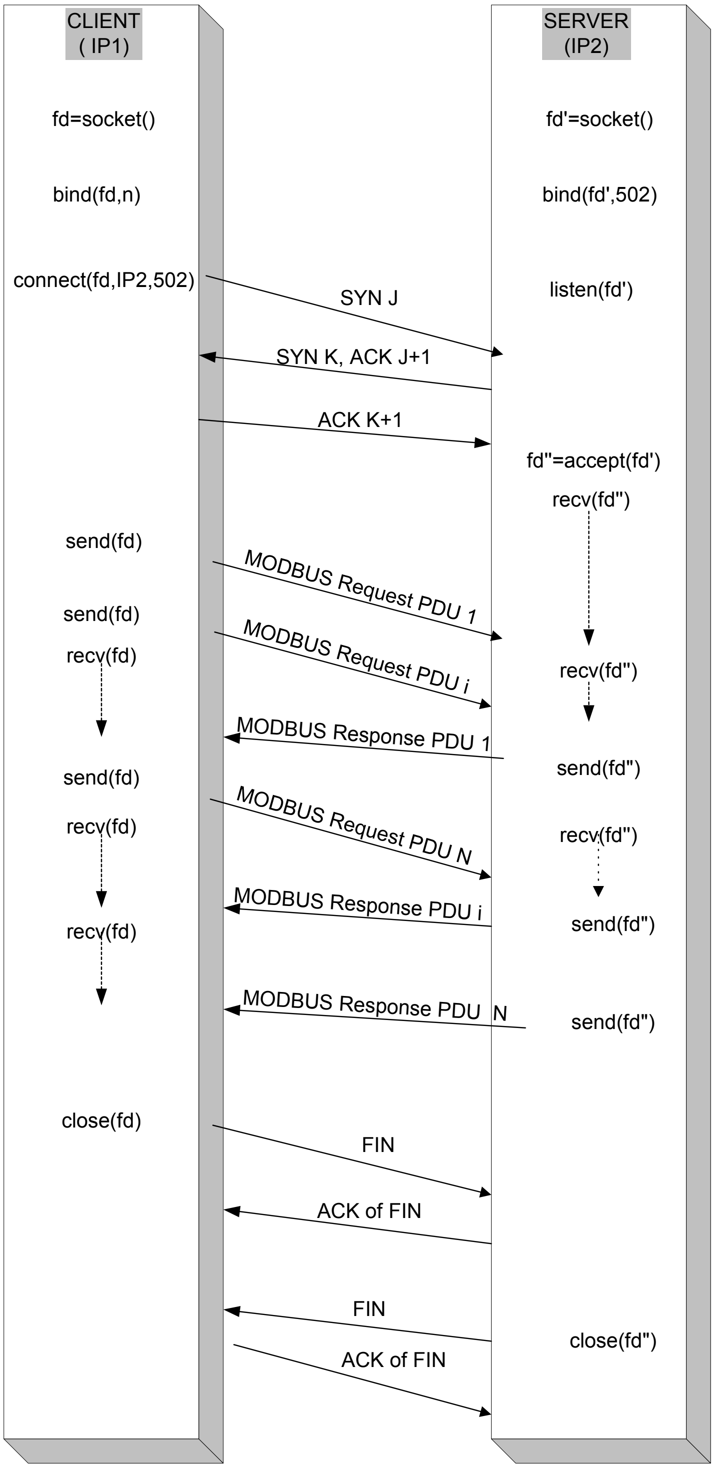


Figure 9: MODBUS Exchanges

4.3.2 TCP layer parameterization

Some parameters of the TCP/IP stack can be adjusted to adapt its behavior to the product or system constraints. The following parameters can be adjusted in the TCP layer:

- **Parameters for each connection:**

SO-RCVBUF, SO-SNDBUF:

These parameters allow setting the high water mark for the Send and the Receive Socket. They can be adjusted for flow control management. The size of the received buffer is the maximum size advertised window for that connection. Socket buffer sizes must be increased in order to increase performances. Nevertheless these values must be smaller than internal driver resources in order to close the TCP window before exhausting internal driver resources.

The received buffer size depends on the TCP Windows size, the TCP Maximum segment size and the time needed to absorb the incoming frames. With a Maximum Segment Size of 300 bytes (a MODBUS request needs a maximum of 256 bytes + the MBAP header size), if we need 3 frames buffering, the socket buffer size value can be adjusted to 900 bytes. For biggest needs and best-scheduled time, the size of the TCP window may be increased.

TCP-NODELAY:

Small packets (called tinygrams) are normally not a problem on LANs, since most LANs are not congested, but these tinygrams can lead to congestion on wide area networks. A simple solution, called the "NAGLE algorithm", is to collect small amounts of data and sends them in a single segment when TCP acknowledgments of previous packets arrive.
In order to have better real-time behavior it is recommended to send small amounts of data directly without trying to gather them in a single segment. That is why it is recommended to force the TCP-NODELAY option that disables the "NAGLE algorithm" on client and server connections.

SO-REUSEADDR:

When a MODBUS server closes a TCP connection initialized by a remote client, the local port number used for this connection cannot be reused for a new opening while that connection stays in the "Time-wait" state (during two MSL : Maximum Segment Lifetime).

It is recommended specifying the SO-REUSEADDR option for each client and server connection to bypass this restriction. This option allows the process to assign itself a port number that is part of a connection that is in the 2MSL wait for client and listening socket.

SO-KEEPALIVE:

By default on TCP/IP protocol no data are sent across an idle TCP connection. Therefore if no process at the ends of a TCP connection is sending data to the other, nothing is exchanged between the two TCP modules. This assumes that either the client application or the server application uses timers to detect inactivity in order to close a connection.
It is recommended to enable the KEEPALIVE option on both client and server connection in order to poll the other end to know if the distant has either crashed and is down or crashed and rebooted.
Nevertheless we must keep on mind that enabling KEEPALIVE can cause perfectly good connections to be dropped during transient failures, that it consumes unnecessary bandwidth on the network if the keep alive timer is too short.

- **Parameters for the whole TCP layer:**

Time Out on establishing a TCP Connection:

Most Berkeley-derived systems set a time limit of 75 seconds on the establishment of a new connection, this default value should be adapted to the real time constraint of the application.

Keep Alive parameters:

The default idle time for a connection is 2 hours. Idles times in excess of this value trigger a keep alive probe. After the first keep alive probe, a probe is sent every 75 seconds for a maximum number of times unless a probe response is received.
The maximum number of keep Alive probes sent out on an idle connection is 8. If no probe response is received after sending out the maximum number of keep Alive probes,TCP signals an error to the application that can decide to close the connection

Time-out and retransmission parameters:

A TCP packet is retransmitted if its loss has been detected. One way to detect the loss is to manage a Retransmission Time-Out (RTO) that expires if no acknowledgement has been received from the remote side.
TCP manages a dynamic estimation of the RTO. For that purpose a Round-Trip Time (RTT) is measured after the send of every packet that is not a retransmission. The Round-Trip Time (RTT) is the time taken for a packet to reach the remote device and to get back an acknowledgement to the sending device. The RTT of a connection is calculated dynamically, nevertheless if TCP cannot get an estimate within 3 seconds, the default value of the RTT is set to 3 seconds.
If the RTO has been estimated, it applies to the next packet sending. If the acknowledgement of the next packet is not received before the estimated RTO expiration, the **Exponential BackOff** is activated. A maximum number of retransmissions of the same packet is allowed during a certain amount of time. After that if no acknowledgement has been received, the connection is aborted.
The maximum number of retransmissions and the maximum amount of time before the abort of the connection (tcp_ip_abort_interval) can be set up on some stacks.

Some retransmission algorithms are defined in TCP standards :

- The **Jacobson's RTO estimation algorithm** is used to estimate the Retransmission Time-Out (RTO),
- The **Karn's algorithm** says that the RTO estimation should not be done on a retransmitted segment,
- The **Exponential BackOff** defines that the retransmission time-out is doubled for each retransmission with an upper limit of 64 seconds.
- The **fast retransmission algorithm** allows retransmitting after the reception of three duplicate acknowledgments. This algorithm is advised because on a LAN it may lead to a quicker detection of the loss of a packet than waiting for the RTO expiration.

The use of these algorithms is recommended for a MODBUS implementation.

4.3.3 IP layer parameterization

4.3.3.1 IP Parameters

The following parameters must be configured in the IP layer of a MODBUS implementation :

- Local IP Address : the IP address can be part of a Class A, B or C.

- Subnet Mask, : Subnetting an IP Network can be done for a variety of reasons : use of different physical media (such as Ethernet, WAN, etc.), more efficient use of network addresses, and the capability to control network traffic. The Subnet Mask has to be consistent with the IP address class of the local IP address.

- Default Gateway: The IP address of the default gateway has to be on the same subnet as the local IP address. The value 0.0.0.0 is forbidden. If no gateway is to be defined then this value is to be set to either 127.0.0.1 or the Local IP address.

Remark : The MODBUS messaging service doesn't require the fragmentation function in the IP layer.

The local IP End Point shall be configured with a **local IP Address** and with **a Subnet Mask** and a **Default Gateway** (different from 0.0.0.0) .

4.4 COMMUNICATION APPLICATION LAYER

4.4.1 MODBUS Client

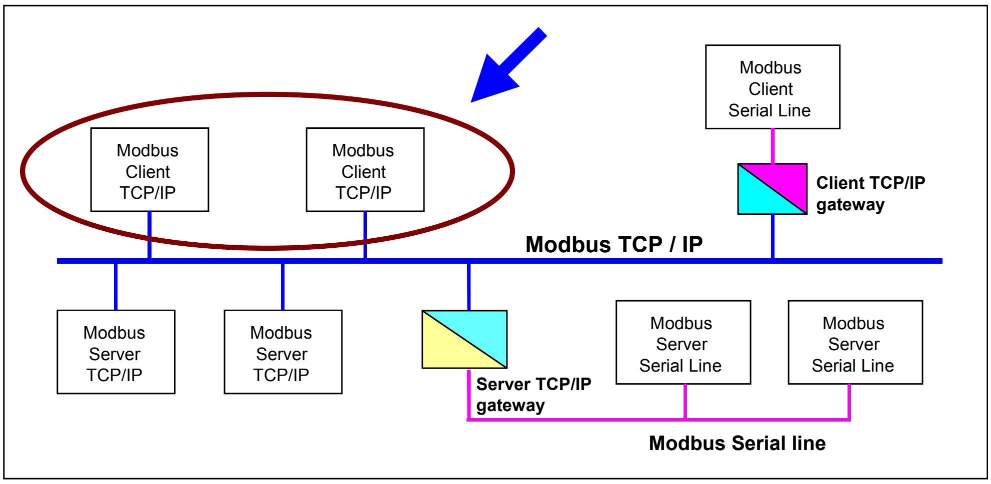


Figure 10: MODBUS Client unit

4.4.1.1 MODBUS client design

The definition of the MODBUS/TCP protocol allows a simple design of a client. The following activity diagram describes the main treatments that are processed by a client to send a MODBUS request and to treat a MODBUS response.

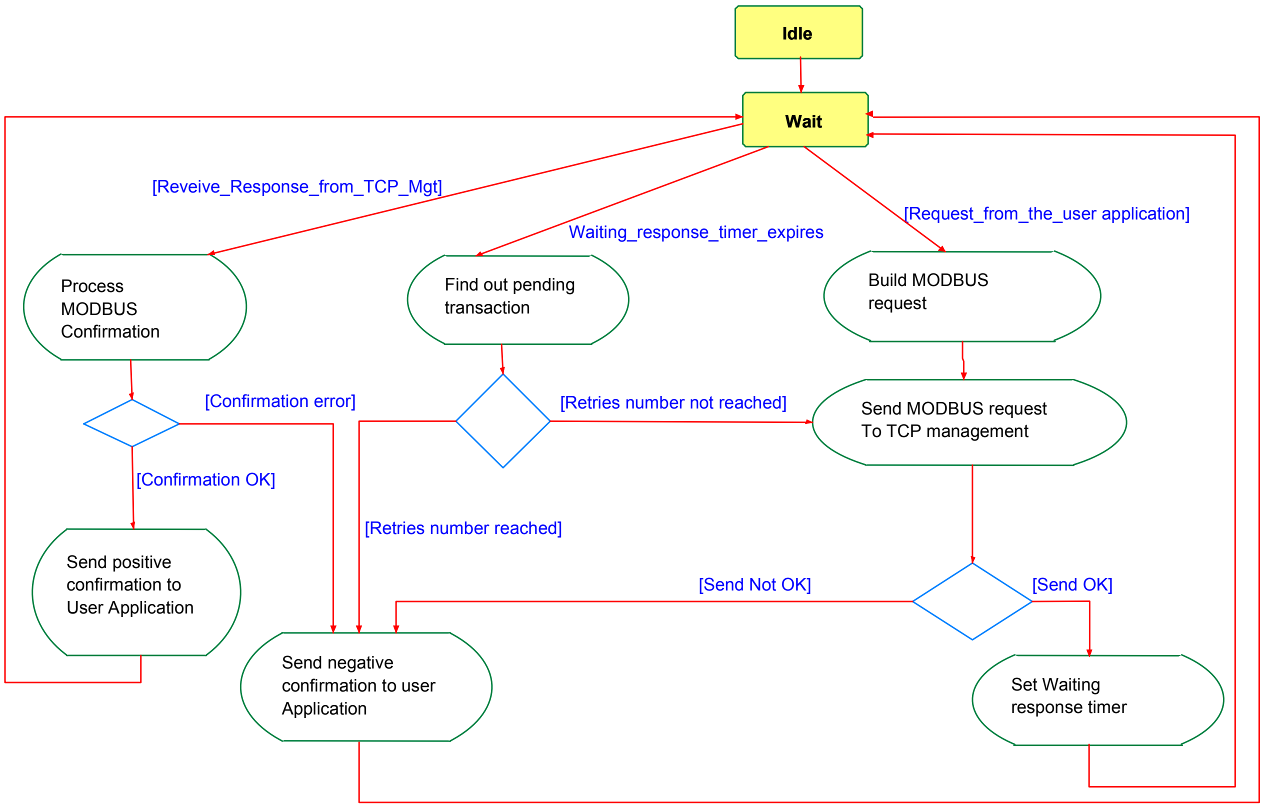


Figure 11: MODBUS Client Activity Diagram

A MODBUS client can receive three events:

- A new demand from the user application to send a request, in this case a MODBUS request has to be encoded and be sent on the network using the TCP management component service. The lower layer (TCP management module) can give back an error due to a TCP connection error, or some other errors.
- A response from the TCP management, in this case the client has to analyze the content of the response and send a confirmation to the user application
- The expiration of a Time out due to a non-response. A new retry can be sent on the network or a negative confirmation can be sent to the User Application. *Remark : These retries are initiated by the MODBUS client, some other retries can also be done by the TCP layer in case of TCP acknowledge lack.*

4.4.1.2 Build a MODBUS Request

Following the reception of a demand from the user application, the client has to build a MODBUS request and to send it to the TCP management.
Building the MODBUS request can be split in several sub-tasks:

- The instantiation of a MODBUS transaction that enables the Client to memorize all required information in order to bind later the response to the request and to send the confirmation to the user application.
- The encoding of the MODBUS request (PDU + MPAB header). The user application that initiates the demand has to provide all required information which enables the Client to encode the request. The MODBUS PDU is encoded according to the MODBUS Application Protocol Specification [1]. (MB function code, associated parameters and application data). All fields of the MBAP header are filled. Then, the MODBUS request ADU is built prefixing the PDU with the MBAP header

- The sending of the MODBUS request ADU to the TCP management module which is in charge of finding the right TCP socket towards the remote Server. In addition to the MODBUS ADU the Destination IP address must also be passed.

The following activity diagram describes, more deeply than in Figure 11 MODBUS Client Activity Diagram, the request building phase.

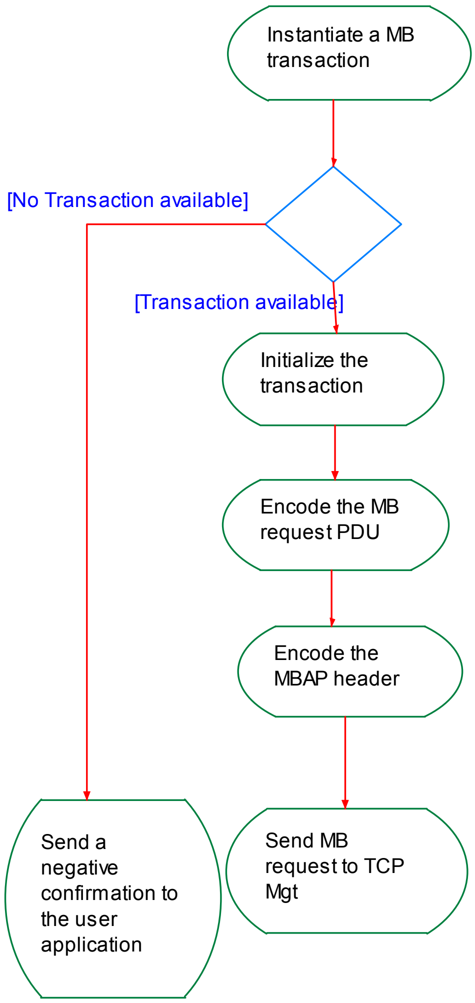


Figure 12: Request building activity diagram

The following example describes the MODBUS request ADU encoding for reading the register # 5 in a remote server :

- MODBUS Request ADU encoding :

	Description	Size	Example
MBAP Header	Transaction Identifier Hi	1	0x15
	Transaction Identifier Lo	1	0x01
	Protocol Identifier	2	0x0000
	Length	2	0x0006
	Unit Identifier	1	0xFF

MODBUS request	*Function Code (*)*	*1*	*0x03*
	Starting Address	*2*	*0x0004*
	Quantity of Registers	*2*	*0x0001*

(*) please see the MODBUS Application Protocol Specification [1].

- **Transaction Identifier**

  The transaction identifier is used to associate the future response with the request. So, at a time, on a TCP connection, this identifier must be unique. There are several manners to use the transaction identifier:
  - For example, it can be used as a simple "TCP sequence number" with a counter which is incremented at each request.
  - It can also be judiciously used as a smart index or pointer to identify a transaction context in order to memorize the current remote server and the pending MODBUS request.

Normally, on MODBUS serial line a client must send one request at a time. This means that the client must wait for the answer to the first request before sending a second request. On TCP/MODBUS, several requests can be sent without waiting for a confirmation to the same server. The MODBUS/TCP to MODBUS serial line gateway is in charge of ensuring compatibility between these two behaviors.
The number of requests accepted by a server depends on its capacity in term of number of resources and size of the TCP windows. In the same way the number of transactions initialized simultaneously by a client depends also on its resource capacity. This implementation parameter is called "**NumberMaxOfClientTransaction**" and must be described as one of the MODBUS client features. Depending of the device type this parameter can take a value from 1 to 16.

- **Unit Identifier**

  This field is used for routing purpose when addressing a device on a MODBUS+ or MODBUS serial line sub-network. In that case, the “Unit Identifier” carries the MODBUS slave address of the remote device:

  If the MODBUS server is connected to a MODBUS+ or MODBUS Serial Line sub-network and addressed through a bridge or a gateway, the MODBUS Unit identifier is necessary to identify the slave device connected on the sub-network behind the bridge or the gateway. The destination IP address identifies the bridge itself and the bridge uses the MODBUS Unit identifier to forward the request to the right slave device.
  The MODBUS slave device addresses on serial line are assigned from 1 to 247 (decimal). Address 0 is used as broadcast address.

  On TCP/IP, the MODBUS server is addressed using its IP address; therefore, the MODBUS Unit Identifier is useless. The value 0xFF has to be used.

  When addressing a MODBUS server connected directly to a TCP/IP network, it’s recommended not using a significant MODBUS slave address in the “Unit Identifier” field. In the event of a re-allocation of the IP addresses within an automated system and if a IP address previously assigned to a MODBUS server is then assigned to a gateway, using a significant slave address may cause trouble because of a bad routing by the gateway. Using a non-significant slave address, the gateway will simply discard the MODBUS PDU with no trouble. 0xFF is recommended for the “Unit Identifier" as non-significant value.
  Remark : The value 0 is also accepted to communicate directly to a MODBUS/TCP device.

4.4.1.3 Process MODBUS Confirmation

When a response frame is received on a TCP connection, the Transaction Identifier carried in the MBAP header is used to associate the response with the original request previously sent on that TCP connection:

- If the Transaction Identifier doesn't refer to any MODBUS pending transaction, the response must be discarded.
- If the Transaction Identifier refers to a MODBUS pending transaction, the response must be parsed in order to send a MODBUS Confirmation to the User Application (positive or negative confirmation)

Parsing the response consists in verifying the MBAP Header and the MODBUS PDU response:

- **MBAP Header**

  After the verification of the Protocol Identifier that must be 0x0000, the length gives the size of the MODBUS response.

  If the response comes from a MODBUS server device directly connected to the TCP/IP network, the TCP connection identification is sufficient to unambiguously identify the remote server. Therefore, the Unit Identifier carried in the MBAP header is not significant (value 0xFF) and must be discarded.

  If the remote server is connected on a Serial Line sub-network and the response comes from a bridge, a router or a gateway, then the Unit Identifier (value != 0xFF) identifies the remote MODBUS server which has originally sent the response.

- **MODBUS Response PDU**

  The function code must be verified and the MODBUS response format analyzed according to the MODBUS Application Protocol:

  - if the function code is the same as the one used in the request, and if the response format is correct, then the MODBUS response is given to the user application as a **Positive Confirmation**.
  - If the function code is a MODBUS exception code (Function code + 80H), the MODBUS exception response is given to the user application as a **Positive Confirmation**.
  - If the function code is different from the one used in the request (=non expected function code), or if the format of the response is incorrect, then an error is signaled to the user application using a **Negative Confirmation**.

Remark: A positive confirmation is a confirmation that the command was received and responded to by the server. It does not imply that the server was able to successfully act on the command (failure to successfully act on the command is indicated by the MODBUS Exception response).

The following activity diagram describes, more deeply than in Figure 11 MODBUS Client Activity Diagram, the confirmation processing phase.

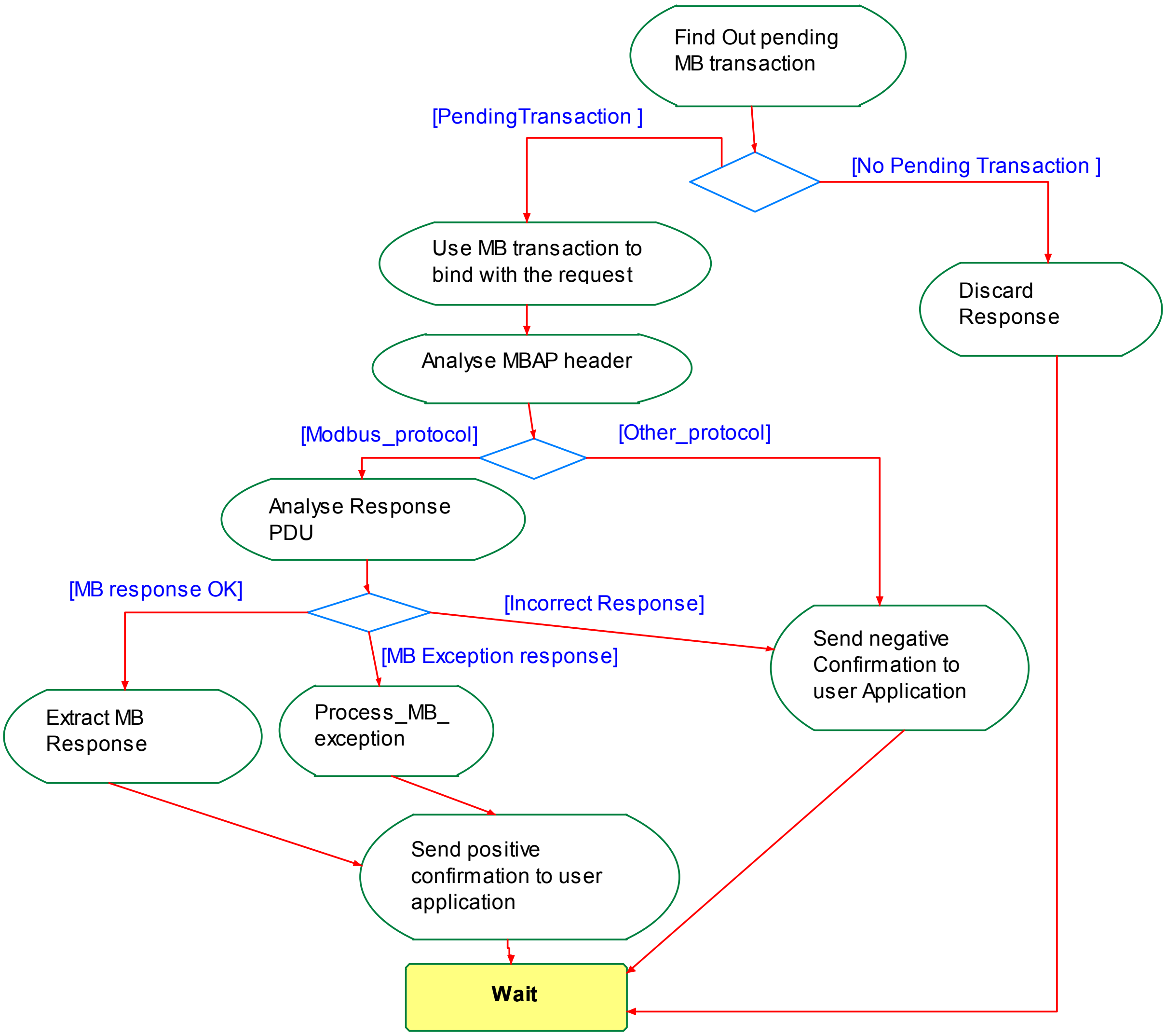


Figure 13: Process MODBUS Confirmation activity diagram

4.4.1.4 Time-out management

There is deliberately NO specification of required response time for a transaction over MODBUS/TCP.
This is because MODBUS/TCP is expected to be used in the widest possible variety of communication situations, from I/O scanners expecting sub-millisecond timing to long distance radio links with delays of several seconds.
From a client perspective, the timeout must take into account the expected transport delays across the network, to determine a 'reasonable' response time. Such transport delays might be milliseconds for a switched Ethernet, or hundreds of milliseconds for a wide area network connection.
In turn, any 'timeout' time used at a client to initiate an application retry should be larger than the expected maximum 'reasonable' response time. If this is not followed, there is a potential for excessive congestion at the target device or on the network, which may in turn cause further errors. This is a characteristic, which should always be avoided.

So in practice, the client timeouts used in high performance applications are always likely to be somewhat dependent on network topology and expected client performance. Applications which are not time critical can often leave timeout values to the normal TCP defaults, which will report communication failure after several seconds on most platforms.

4.4.2 MODBUS Server

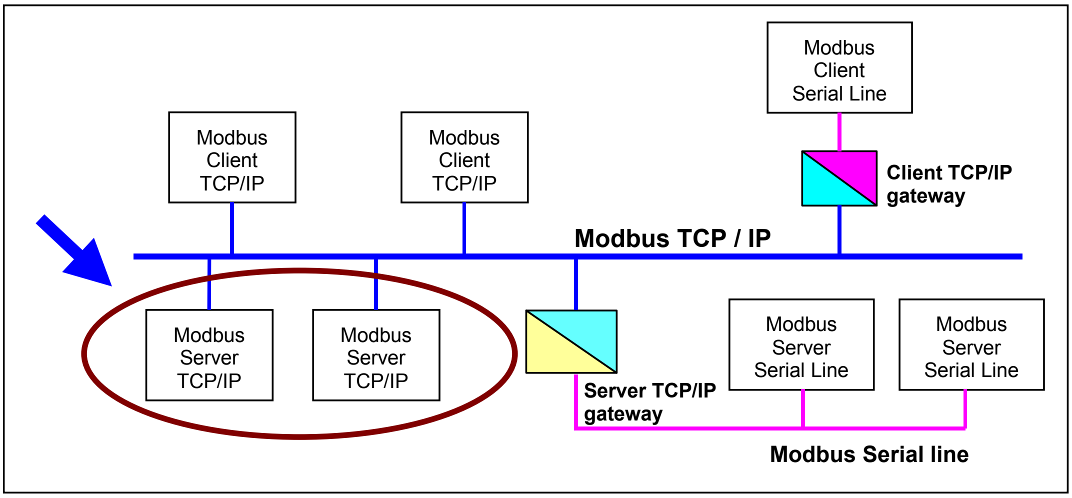


Figure 14: MODBUS Server unit

The role of a MODBUS server is to provide access to application objects and services to remote MODBUS clients.

Different kind of access may be provided depending on the user application :

- simple access like get and set application objects attributes
- advanced access in order to trigger specific application services

The MODBUS server has:

- To map application objects onto readable and writable MODBUS objects, in order to get or set application objects attributes.
- To provide a way to trigger services onto application objects.

In run time the MODBUS server has to analyze a received MODBUS request, to process the required action, and to send back a MODBUS response.

Informative Note: The application objects and services of the Backend Interface obtain the requested data based upon the function code, and the User is responsible.

4.4.2.1 MODBUS Server Design

The MODBUS Server design depends on both :

- the kind of access to the application objects (simple access to attributes or advanced access to services)
- the kind of interaction between the MODBUS server and the user application (synchronous or asynchronous).

The following activity diagram describes the main treatments that are processed by the Server to obtain a MODBUS request from TCP Management, then to analyze the request, to process the required action, and to send back a MODBUS response.

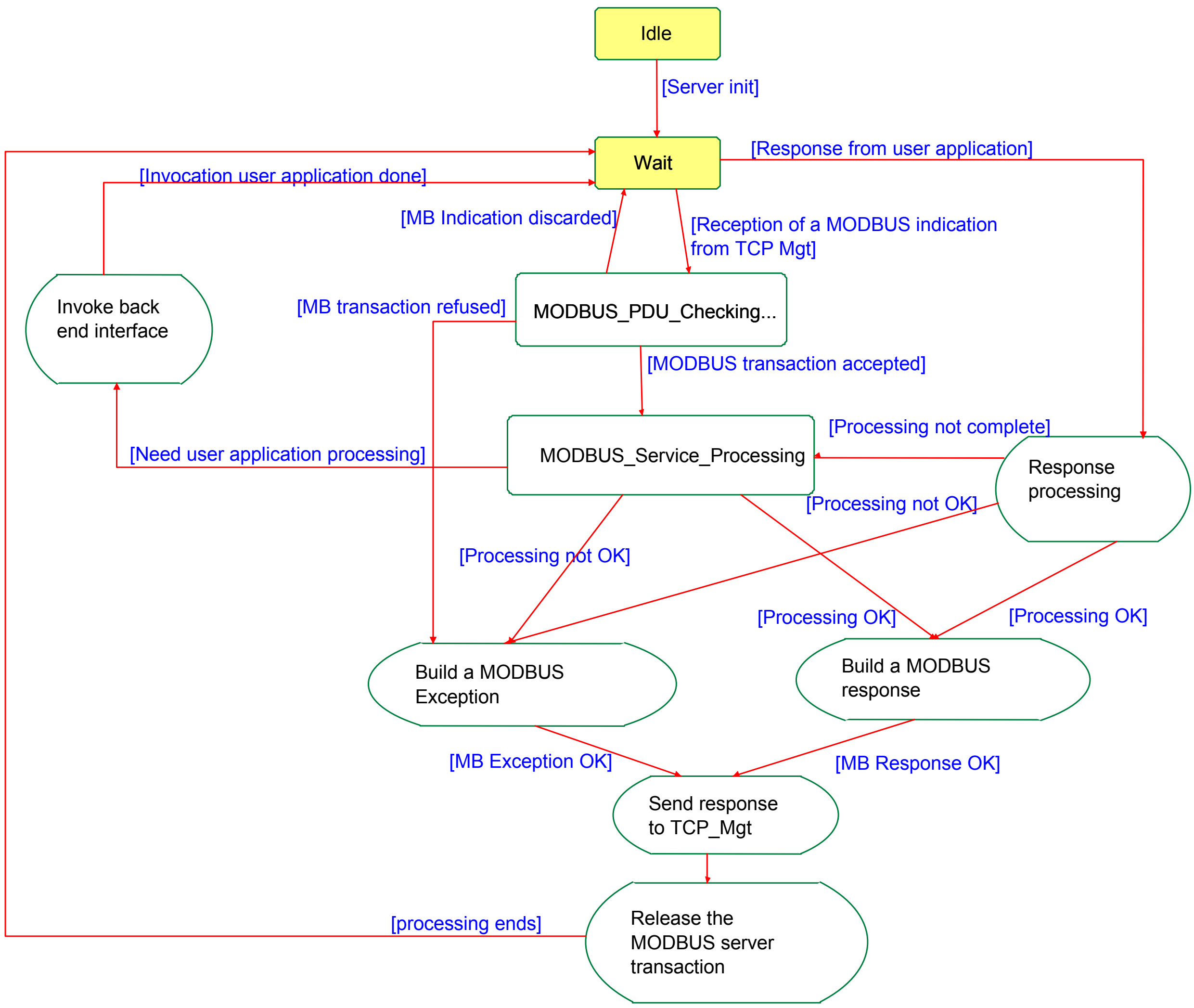


Figure 15: Process MODBUS Indication activity diagram

As shown in the previous activity diagram:

- Some services can be immediately processed by the MODBUS Server itself, with no direct interaction with the User Application ;
- Some services may require also interacting explicitly with the User Application to be processed ;
- Some other advanced services require invoking a specific interface called MODBUS Back End service. For example, a User Application service may be triggered using a sequence of several MODBUS request/response transactions according to a User Application level protocol. The Back End service is responsible for the correct processing of all individual MODBUS transactions in order to execute the global User Application service.

A more complete description is given in the following sections.

The MODBUS server can accept to serve simultaneously several MODBUS requests. The maximum number of simultaneous MODBUS requests the server can accept is one of the main characteristics of a MODBUS server. This number depends on the server design and its processing and memory capabilities. This implementation parameter is called "**NumberMaxOfSeverTransaction**" and must be described as one of the MODBUS server features. It may have a value from 1 to 16 depending on the device capabilities.

The behavior and the performance of the MODBUS server are significantly affected by the "NumberMaxOfTransaction" parameter. Particularly, it's important to note that the number of concurrent MODBUS transactions managed may affect the response time of a MODBUS request by the server.

4.4.2.2 MODBUS PDU Checking

The following diagram describes the MODBUS PDU Checking activity.

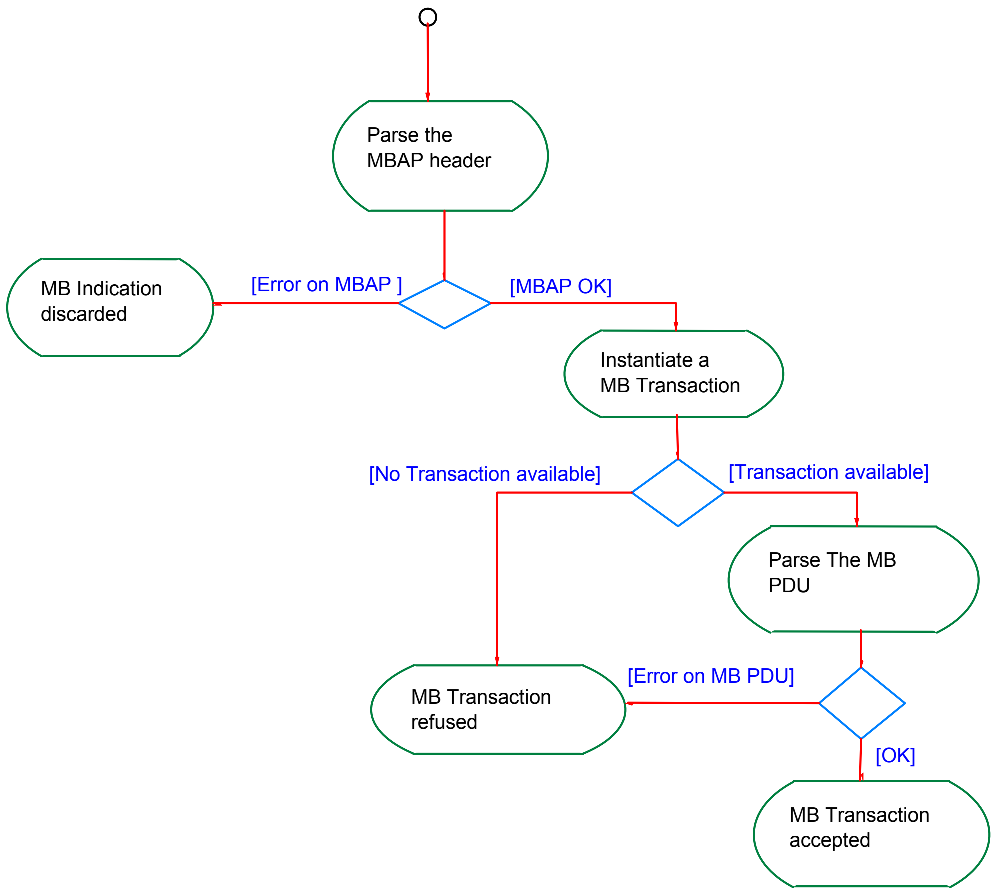


Figure 16: MODBUS PDU Checking activity diagram

The MODBUS PDU Checking function consists of first parsing the MBAP Header. The Protocol Identifier field has to be checked :

- If it is different from MODBUS protocol type, the indication is simply discarded.
- If it is correct (= MODBUS protocol type; value 0x00), a MODBUS transaction is instantiated.

The maximum number of MODBUS transactions the server can instantiate is defined by the "NumberMaxOfTransaction" parameter (A system or a configuration parameter).

In case of no more transactions available, the server builds a MODBUS exception response (Exception code 6 : Server Busy).

If a MODBUS transaction is available, it's initialized in order to memorize the following information:

- The TCP connection identifier used to send the indication (given by the TCP Management)
- The MODBUS Transaction ID (given in MBAP Header)
- The Unit Identifier (given in MBAP Header)

Then the MODBUS PDU is parsed. The function code is first controlled :

- in case of invalidity a MODBUS exception response is built (Exception code 1 : Invalid function).
- If the function code is accepted, the server initiates the "MODBUS Service processing" activity.

4.4.2.3 MODBUS service processing

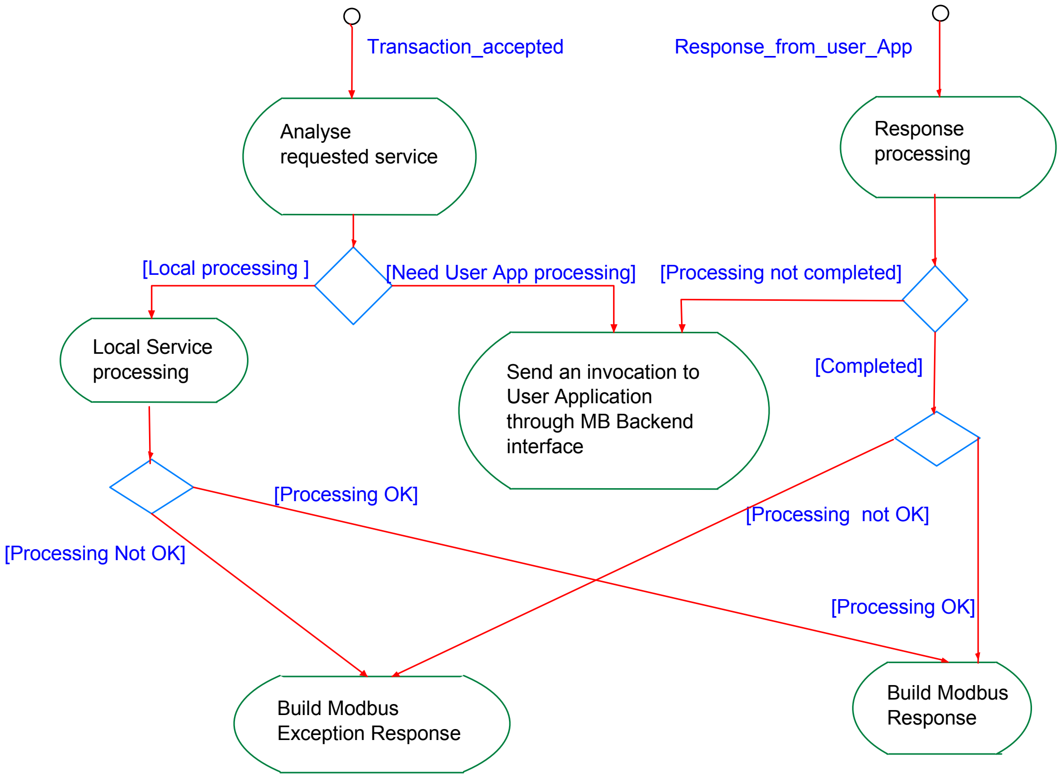


Figure 17: MODBUS service processing activity diagram

The processing of the required MODBUS service can be done in different ways depending on the device software and hardware architecture as described in the hereafter examples :

- Within a compact device or a mono-thread architecture where the MODBUS server can access directly to the user application data, the required service can be processed "locally" by the server itself without invoking the Back End service. The processing is done according to the MODBUS Application Protocol Specification [1]. In case of an error, a MODBUS exception response is built.
- Within a modular multi-processor device or a multi-thread architecture where the "communication layers" and the "user application layer" are 2 separate entities, some trivial services can be processed completely by the Communication entity while some others can require a cooperation with the User Application entity using the Back End service.

To interact with the User Application, the MODBUS Backend service must implement all appropriate mechanisms in order to handle User Application transactions and to manage correctly the User Application invocations and associated responses.

4.4.2.4 User Application Interface (Backend Interface)

Several strategies can be implemented in the MODBUS Backend service to achieve its job although they are not equivalent in terms of user network throughput, interface bandwidth usage, response time, or even design workload.

The MODBUS Backend service will use the appropriate interface to the user application :

- Either a physical interface based on a serial link, or a dual-port RAM scheme, or a simple I/O line, or a logical interface based on messaging services provided by an operating system.
- The interface to the User Application may be synchronous or asynchronous.

The MODBUS Backend service will also use the appropriate design pattern to get/set objects attributes or to trigger services. In some cases, a simple "gateway pattern" will be adequate. In some other cases, the designer will have to implement a "proxy pattern" with a corresponding caching strategy, from a simple exchange table history to more sophisticated replication mechanisms.

The MODBUS Backend service has the responsibility to implement the protocol transcription in order to interact with the User Application. Therefore, it can have to implement mechanisms for packet fragmentation/reconstruction, data consistency guarantee, and synchronization whatever is required.

4.4.2.5 MODBUS Response building

Once the request has been processed, the MODBUS server has to build the response using the adequate MODBUS server transaction and has to send it to the TCP management component.
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 :
  - 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 ;
  - The exception code is provided to indicate the reason of the error.

Exception Code	MODBUS name	Comments
01	Illegal Function Code	The function code is unknown by the server
02	Illegal Data Address	Dependant on the request
03	Illegal Data Value	Dependant on the request
04	Server Failure	The server failed during the execution
05	Acknowledge	The server accepted the service invocation but the service requires a relatively long time to execute. The server therefore returns only an acknowledgement of the service invocation receipt.
06	Server Busy	The server was unable to accept the MB Request PDU. The client application has the responsibility of deciding if and when to re-send the request.
0A	Gateway problem	Gateway paths not available.
0B	Gateway problem	The targeted device failed to respond. The gateway generates this exception

The MODBUS response PDU must be prefixed with the MBAP header which is built using data memorized in the transaction context.

- **Unit Identifier**
  The Unit Identifier is copied as it was given within the received MODBUS request and memorized in the transaction context.

- **Length**
  The server calculates the size of the MODBUS PDU plus the Unit Identifier byte. This value is set in the "Length" field.
- **Protocol Identifier**
  The Protocol Identifier field is set to 0x0000 (MODBUS protocol), as it was given within the received MODBUS request.
- **Transaction Identifier**
  This field is set to the "Transaction Identifier" value that was associated with the original request and memorized in the transaction context.

Then the MODBUS response must be returned to the right MODBUS Client using the TCP connection memorized in the transaction context. When the response is sent, the transaction context must be free.

5 IMPLEMENTATION GUIDELINE

The objective of this section is to propose an example of a messaging service implementation.
The model describes below can be used as a guideline during a client or a server implementation of a MODBUS messaging service.

Informative Note: The messaging service implementation is the responsibility of the User.

5.1 OBJECT MODEL DIAGRAM

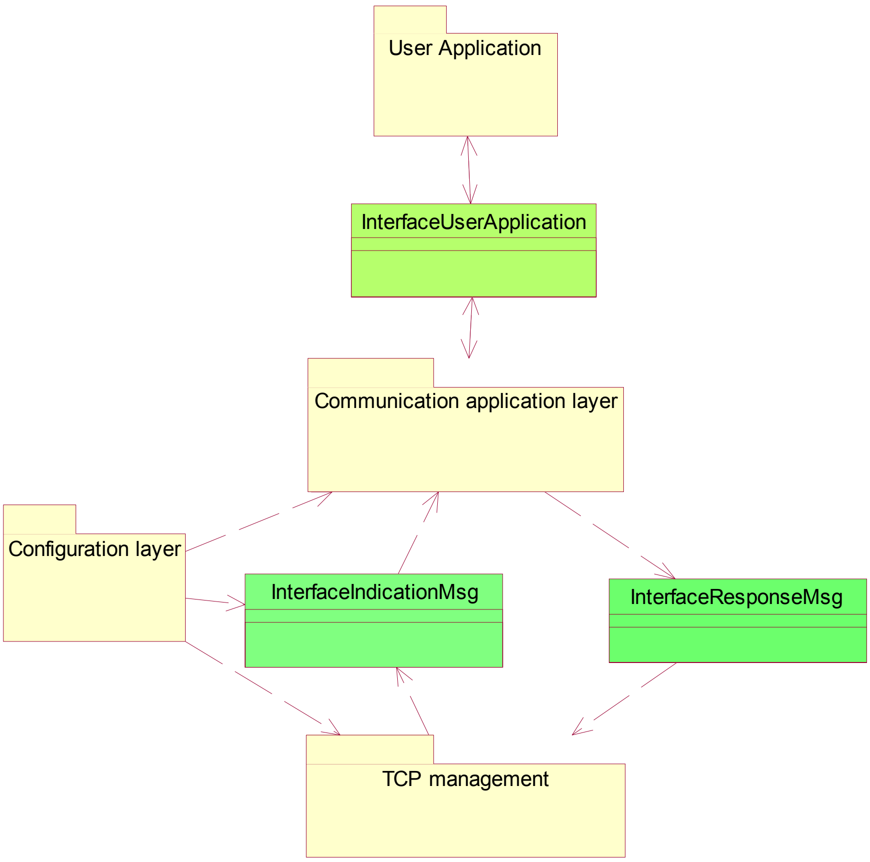


Figure 18: MODBUS Messaging Service Object Model Diagram

Four main packages compose the Object Model Diagram:

- **The Configuration layer** which configures and manages operating modes of components of other packages
- **The TCP Management** which interfaces the TCP/IP stack and the communication application layer managing TCP connection. It implies the management of socket interface.
- **The Communication application layer** which is composed by the MODBUS client on one side and the MODBUS server on the other side. This package is linked with the user application.

The User application, which corresponds to the device application, is completely dependent on the device and therefore it will be not part of this Specification.

This model is independent of implementation choices like the type of OS, the memory management, etc. In order to guarantee this independence generic Interface layers are used between the TCP management layer and the communication layer and between the communication layer and the user application layer.

Different implementations of this interface can be realized by the User: Pipe between two tasks, shared memory, serial link interface, procedural call, etc.

Some assumptions have to be taken to define the hereafter implementation model :

- Static memory management
- Synchronous treatment of the server
- One task to process the receptions on all sockets.

5.1.1 TCP management package

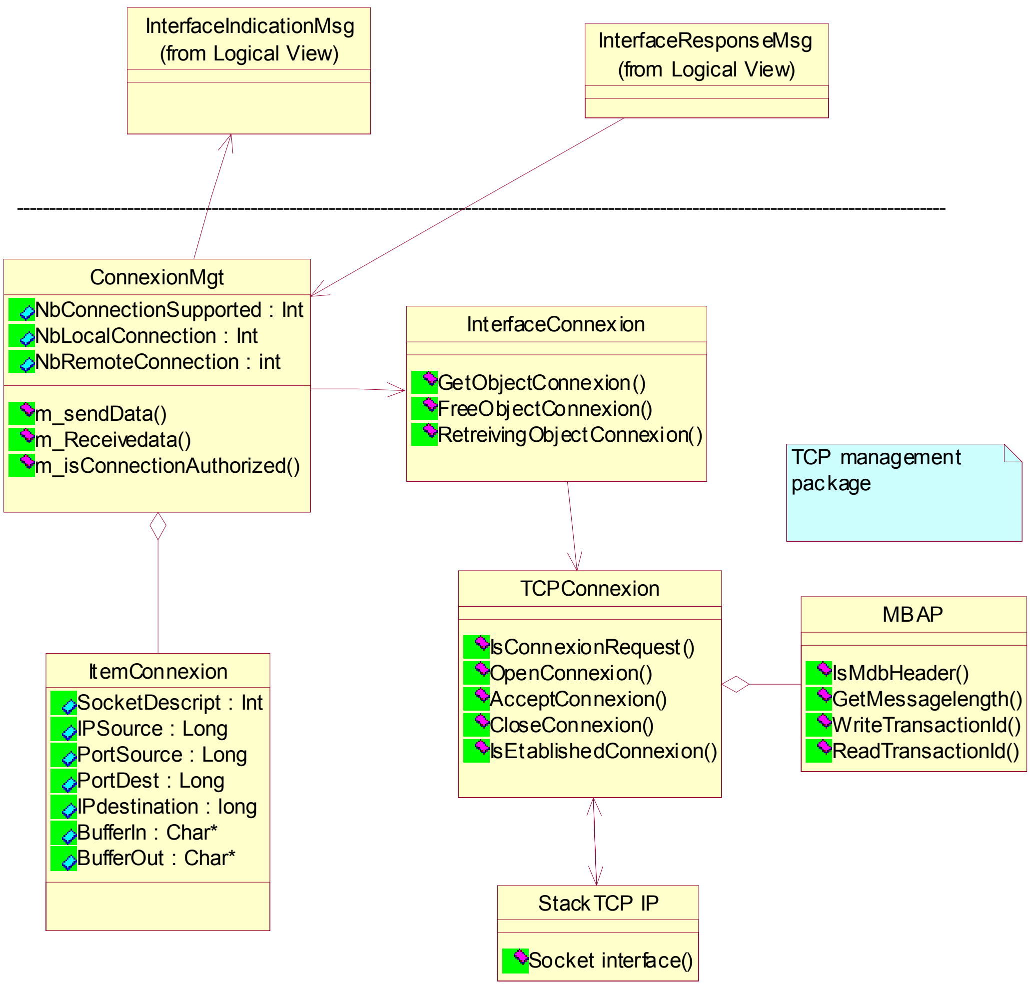


Figure 19: MODBUS TCP management package

The TCP management package comprises the following classes :

CInterfaceConnexion: The role of this class consists in managing memory pool for connections.

CItemConnexion: This class contains all information needed to describe a connection.

CTCPConnexion:, This class provides methods for managing automatically a TCP connection (Interface socket is provided by **CStackTCP_IP**).

CConnexionMngt: This class manages all connections and send query/response to MODBUS Server/MODBUS Client through **CinterfaceIndicationMsg** and **CInterfaceResponseMsg**. This class also treats the Access control for the connection opening.

CMBAP: This class provides methods for reading/writing/analyzing the MODBUS MBAP.

CStackTCP_IP: This class Implements socket services and provides parameterization of the stack.

5.1.2 Configuration layer package

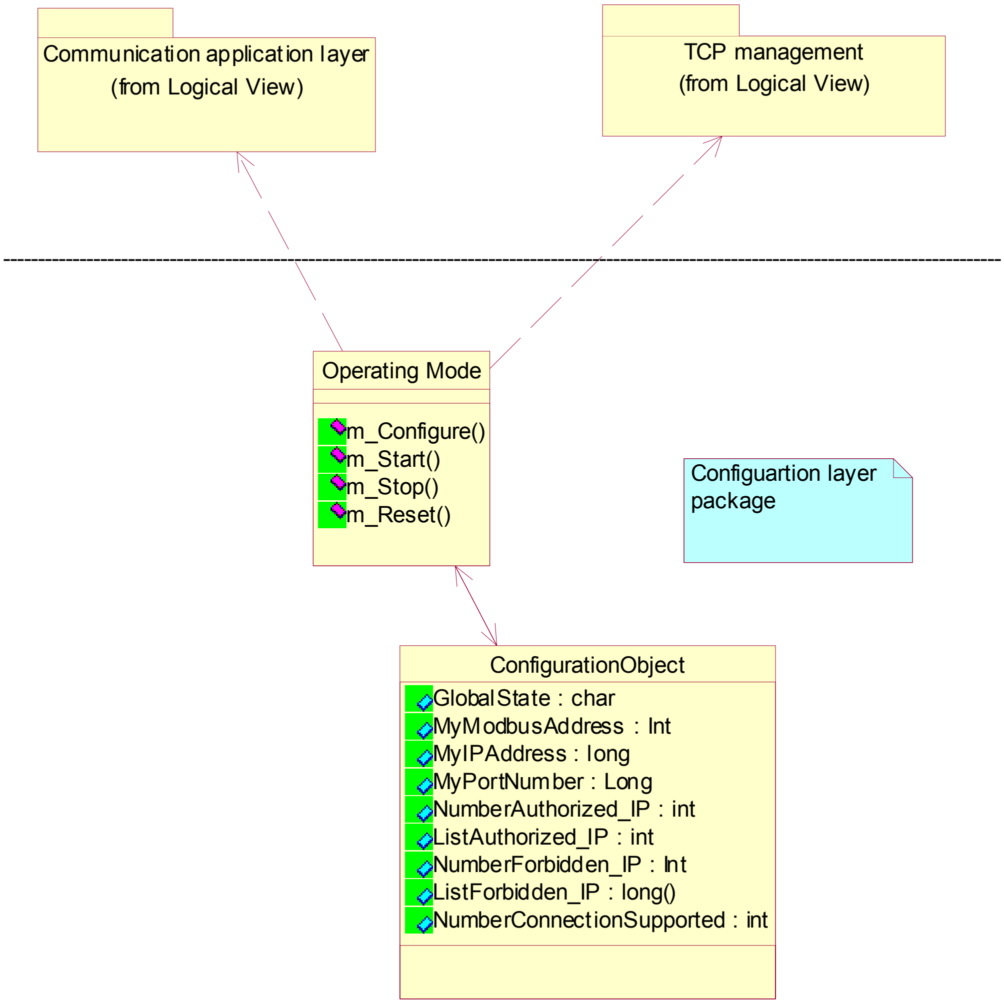


Figure 20: MODBUS Configuration layer package

The Configuration layer package comprises the following classes :

TConfigureObject: This class groups all data needed for configuring each other component. This structure is filled by the method m_Configure from the class **CoperatingMode**. Each class needing to be configured gets its own configuration data from this object. The configuration data is implementation dependent therefore the list of attributes of this class is provided as an example.

COperatingMode: The role of this class is to fill the **TConfigureObject** (according to the user configuration) and to manage the operating modes of the classes described below:

- CMODBUSServer
- CMODBUSClient

- CconnexionMngt

5.1.3 Communication layer package

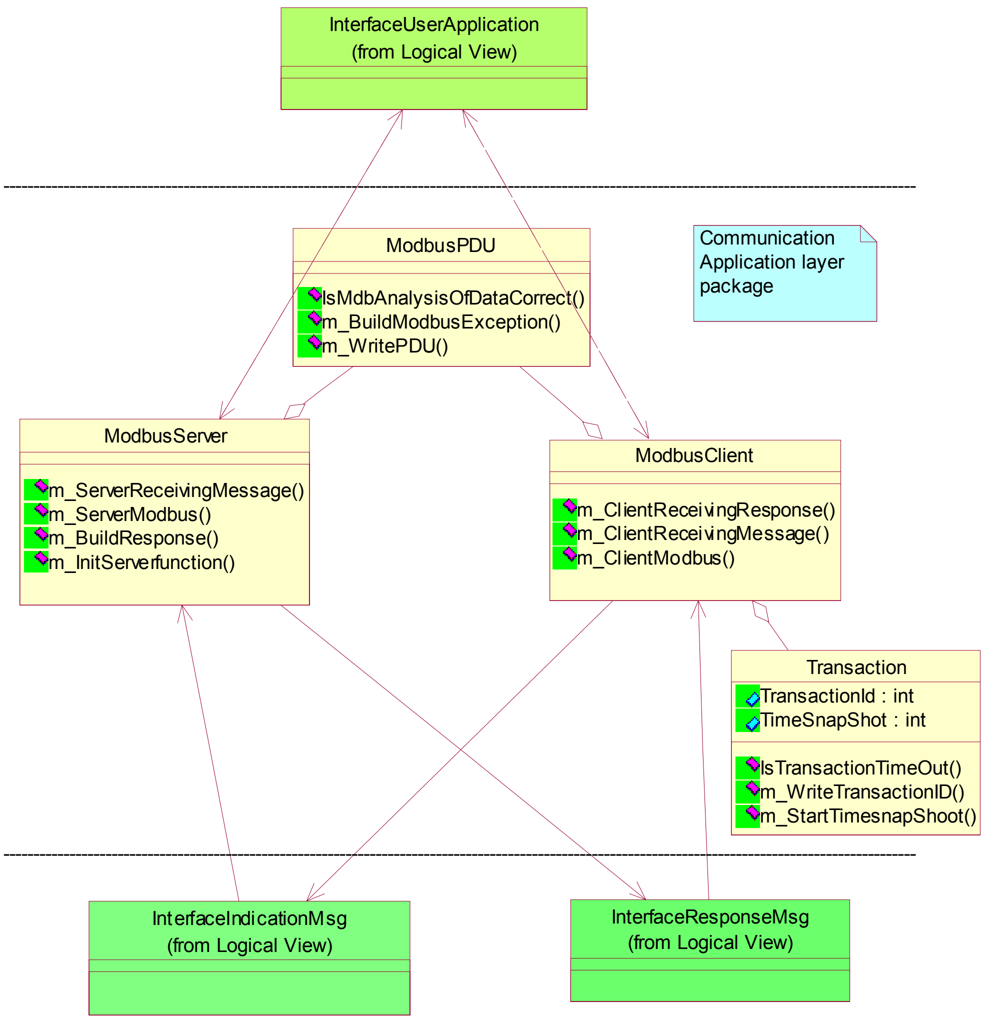


Figure 21: MODBUS Communication Application layer package

The Communication Application layer package comprises the following classes :

CMODBUSServer: MODBUS query is received from class **CInterfaceIndicationMsg** (by the method m_ServerReceivingMessage). The role of this class is to build the MODBUS response or the MODBUS Exception according the query (incoming from network). This class implements the Graph State of MODBUS server. Response can be built only if class **COperatingMode** has sent both user configuration and right operating modes.

CMODBUSClient: MODBUS query is read from class **CInterfaceUserApplication**, The client task receives query by the method m_ClientReceivingMessage. This class implements the State Graph of MODBUS client and manages transactions for linking query with response (from network). Query can be sent over network only if class **CoperatingMode** has sent both user configuration and right operating modes.

CTransaction: This class implements methods and structures for managing transactions.

5.1.4 Interface classes

CInterfaceUserApplication: This class represents the interface with the user application, it provides two methods to access to the user data. In a real implementation this method can be implemented in different way depending of the hardware and software device capabilities (equivalent to an end-driver, example access to PCMCIA, shared memory, etc).

CInterfaceIndicationMsg: This Interface class is proposed for sending query from Network to the MODBUS Server, and for sending response from Network for the Client. This class interfaces TCPManagement and 'Communication Application Layer' packages (From Network). The implementation of this class is device dependent.

CInterfaceResponseMsg: This Interface class is used for receiving response from the Server and for sending query from the client to the Network. This class interfaces packages 'Communication Application Layer' and package 'TCPManagement' (To Network). The implementation of this class is device dependent.

5.2 IMPLEMENTATION CLASS DIAGRAM

The following Class Diagram describes the complete diagram of a proposal implementation.

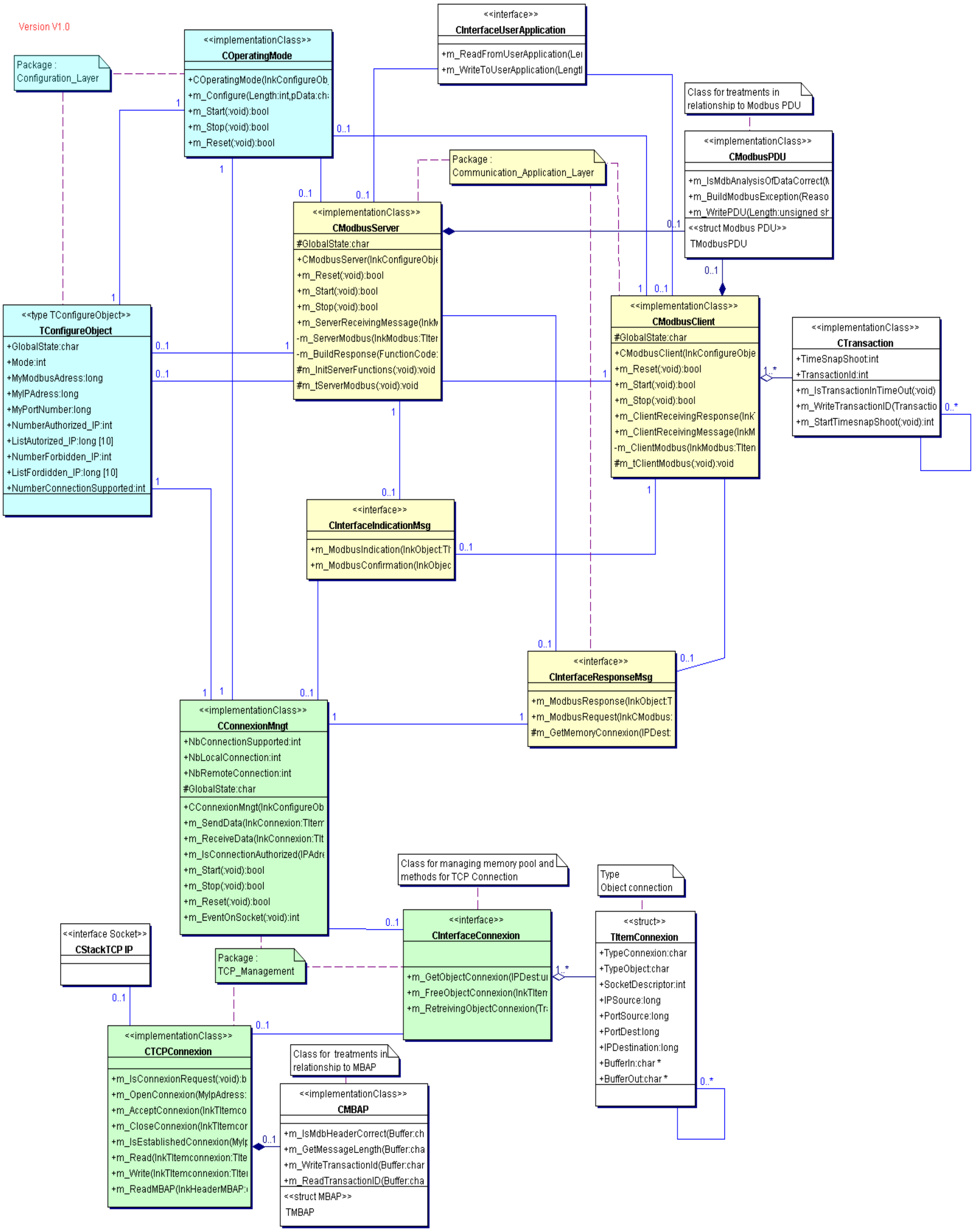


Figure 22: Class Diagram

5.3 SEQUENCE DIAGRAMS

Two Sequence diagrams are described hereafter are an example in order to illustrate a Client MODBUS transaction and a Server MODBUS transaction.

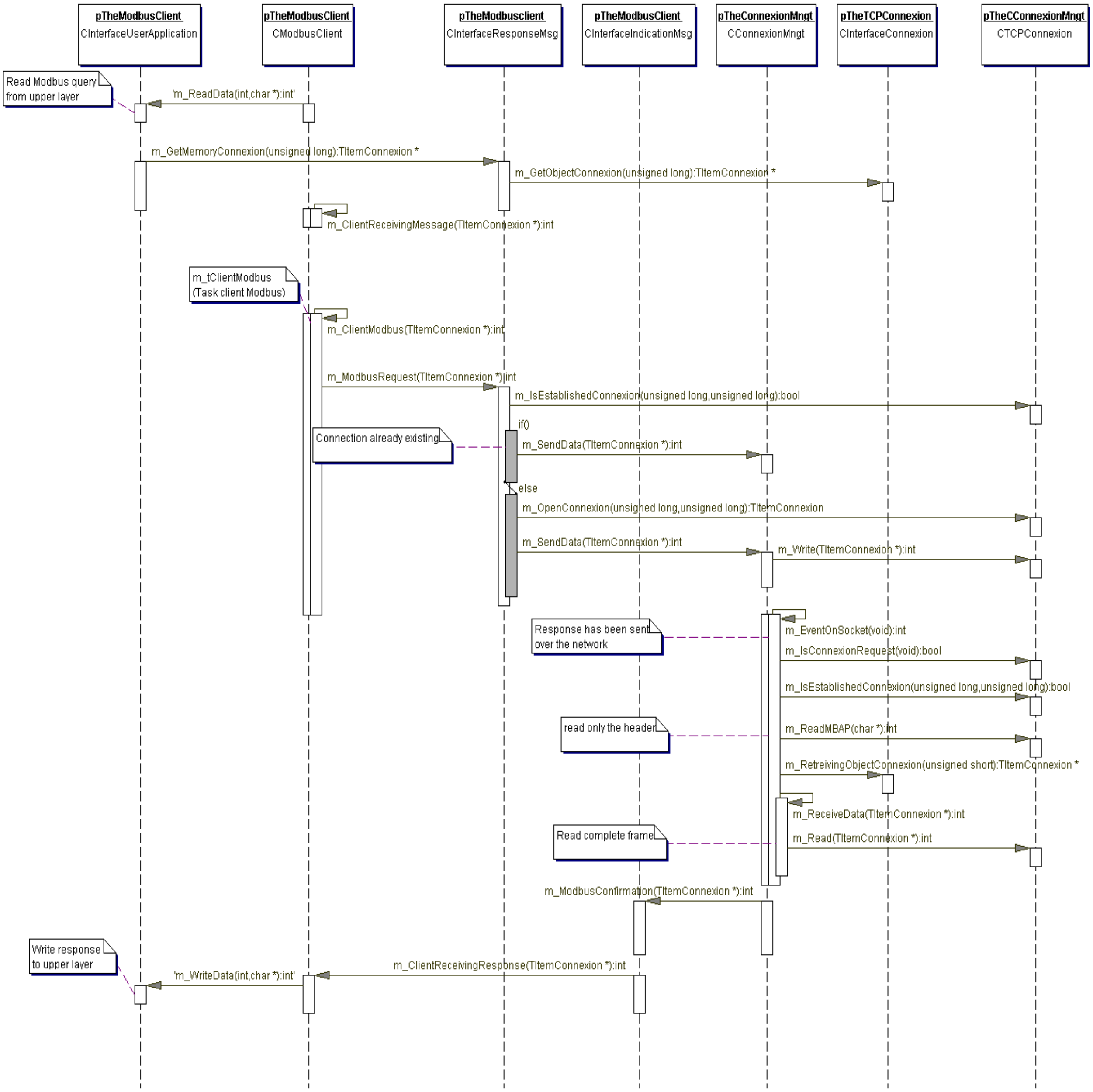


Figure 23: MODBUS client sequence diagram

General comments for a better understanding of the **Client** sequence diagram:

First step: A Reading query comes from User Application (method m_Read).

Second Step: The 'Client' task receives the MODBUS query (method **m_ClientReceivingMessage**). This is the entry point of the Client. To associate the query with the corresponding response when it will arrive, the Client uses a Transaction resource (Class **CTransaction**). The MODBUS query is sent to the TCP_Management by calling the class interface **CInterfaceResponseMsg** (method m_MODBUSRequest)

Third Step: If the connection is already established there is nothing to do on connection, the message can be send over the network. Otherwise, a connection must be opened before the message can be sent over the network.
At this time the client is waiting for a response (from a remote server)

Fourth step: Once a response has been received from the network, the TCP/IP stack receives data (method **m_EventOnSocket** is implicitly called).
If the connection is already established, then the MBAP is read for retrieving the connection object (connection object gives memory resource and other information).
Data coming from network is read and confirmation is sent to the client task via the class Interface **CInterfaceIndicationMsg** (method **m_MODBUSConfirmation**). Client task receives the MODBUS Confirmation (method **m_ClientReceivingResponse**).
Finally the response is written to the user application (method m_WriteData), and transaction resource is freed.

Hereafter is an example of a MODBUS Server exchange.

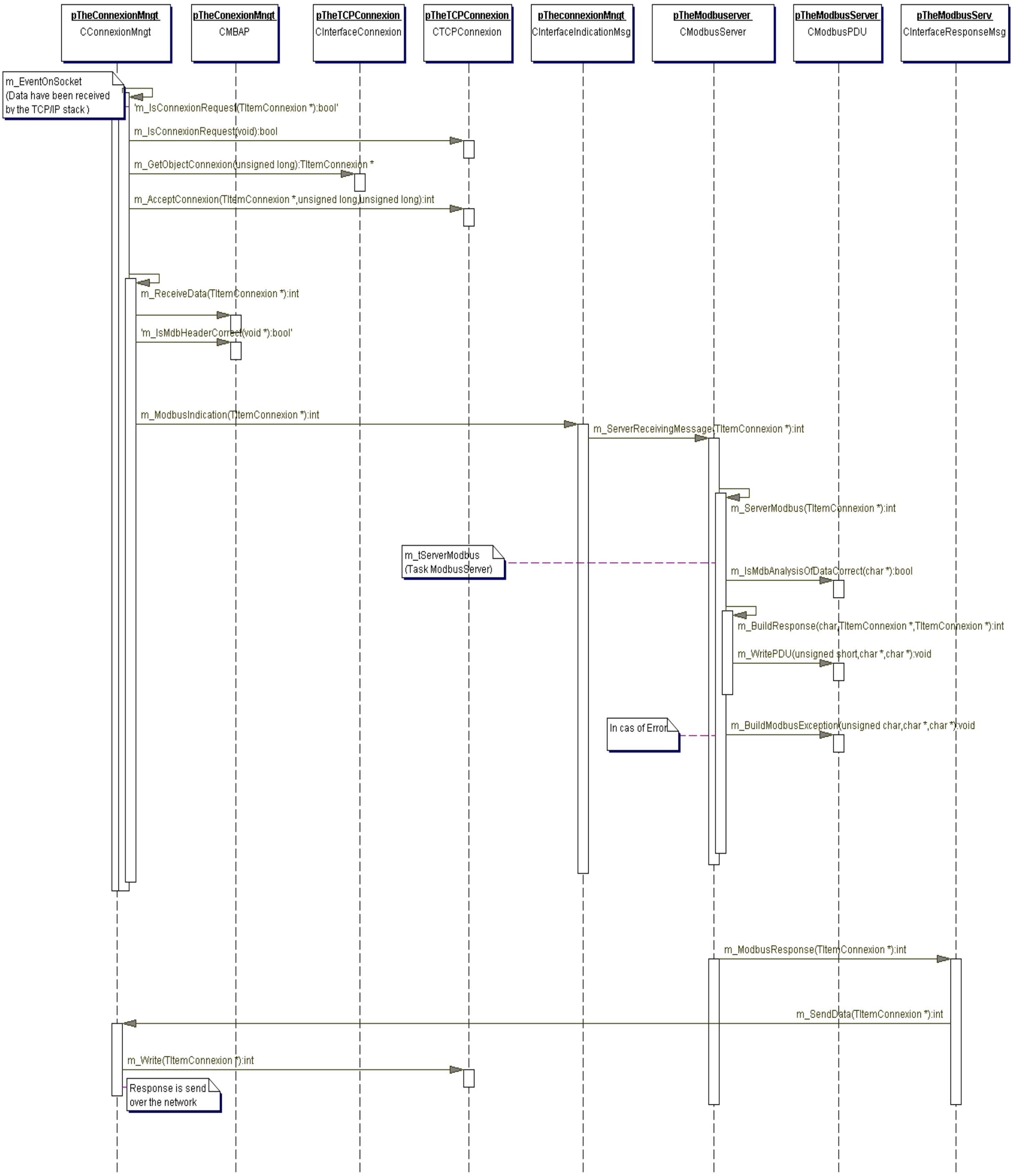


Figure 24: MODBUS server Diagram

General comments for a better understanding of the **Server** sequence diagram:

First step: a client has sent a query (MODBUS query) over the network.
The TCP/IP stack receives data (method **m_EventOnSocket** is implicitly called).

Second step: The query may be a connection request or not (method **m_IsConnexionRequest**).
If the query is a connection request, the connection object and buffers for receiving and sending the MODBUS frame are allocated (method **m_GetObjectConnexion**). Just after, the connection access control must be checked and accepted (method **m_AcceptConnexion**)

Third step: If the query is a MODBUS request, the complete MODBUS Query can be read (method **m_ReceiveData**). At this time the MBAP must be analyzed (method **m_IsMdbHeaderCorrect**). The complete frame is sent to the Server task via the **CinterfaceIndicationMessaging** Class (method **m_MODBUSIndication**). Server task receives the MODBUS Query (method **m_ServerReceivingMessage**) and analyses it.
If an error occurs (function code not supported, etc), a MODBUSException frame is built (**m_BuildMODBUSException**), otherwise the response is built.

Fourth Step: The response is sent over the network via the **CinterfaceResponseMessaging** (method **m_MODBUSResponse**). Treatment on the connection object is done by the method **m_SendData** (retrieve the connection descriptor, etc) and data is sent over the network.

5.4 CLASSES AND METHODS DESCRIPTION

5.4.1 MODBUS Server Class

Class CMODBUSServer

---

class **CMODBUSServer**

Stereotype implementationClass

Provides methods for managing MODBUS Messaging in Server Mode

---

Field Summary	
protected char	**GlobalState** state of the MODBUS Server

Constructor Summary	
CMODBUSServer(TConfigureObject * lnkConfigureObject) Constructor : Create internal object	

Method Summary

protected void	**m_InitServerFunctions**(void) Function called by the constructor for filling array of functions 'm_ServerFunction'
bool	**m_Reset**(void) Method for Resetting Server, return true if reset
int	**m_ServerReceivingMessage**(TItemConnexion * lnkMODBUS) Interface with CindicationMsg::m_MODBUSIndication for receiving Query from NetWork return negative value if problem
bool	**m_Start**(void) Method for Starting Server, return true if Started
bool	**m_Stop**(void) Method for Stopping Server, return true if Stopped
protected void	**m_tServerMODBUS**(void) Server MODBUS task ...

5.4.2 MODBUS Client Class

Class CMODBUSClient

---

class **CMODBUSClient**

Provides methods for managing MODBUS Messaging in Client Mode

Stereotype implementationClass

Field Summary	
protected char	**GlobalState** State of the MODBUS Client

Constructor Summary	
CMODBUSClient(TConfigureObject * lnkConfigureObject) Constructor : Create internal object , initialize to 0 variables.	

Method Summary	
int	**m_ClientReceivingMessage**(TItemConnexion * lnkMODBUS) Interface provided for receiving message from application Layer Typically : Call CinterfaceUserApplication::m_Read for reading data call CInterfaceConnexion::m_GetObjectConnexion for getting memory for a transaction. Return negative value if problem
int	**m_ClientReceivingResponse**(TitemConnexion * lnkTItemConnexion) Interface with CindicationMsg::m_Confirmation for receiving response from network return negative value if problem
bool	**m_Reset**(void) Method for Resetting component, return true if reset
bool	**m_Start**(void) Method for Starting component, return true if started
bool	**m_Stop**(void)

	Method for Stopping component, return true if stopped
protected void	**m_tClientMODBUS**(void) Client MODBUS task....

5.4.3 Interface Classes

5.4.3.1 Interface Indication class

Class CInterfaceIndicationMsg

Direct Known Subclasses:

CConnexionMngt

---

class **CInterfaceIndicationMsg**

Class for sending message from TCP_Management to MODBUS Server or Client

Stereotype interface

Method Summary	
int	**m_MODBUSConfirmation**(TItemConnexion * lnkObject) Method for Receiving incoming Response, calling the Client : could be by reference, by Message Queue, Remote procedure Call, ...
int	**m_MODBUSIndication**(TItemConnexion * lnkObject) Method for reading incoming MODBUS Query and calling the Server : could be by reference, by Message Queue, Remote procedure Call, ...

5.4.3.2 Interface Response Class

Class CInterfaceResponseMsg

Direct Known Subclasses:

CMODBUSClient, CMODBUSServer

---

class **CInterfaceResponseMsg**

Class for sending response or sending query to TCP_Management from Client or Server

Stereotype interface

Method Summary	
TitemConnexion *	**m_GetMemoryConnexion**(unsigned long IPDest) Get an object ITemConnexion from memory pool. Return -1 if not enough memory
int	**m_MODBUSRequest**(TItemConnexion * lnkCMODBUS) Method for Writing incoming MODBUS Query Client to ConnexionMngt :

	could be by reference, by Message Queue, Remote procedure Call, ...
int	**m_MODBUSResponse**(TItemConnexion * lnkObject) Method for writing Response from MODBUS Server to ConnexionMngt could be by reference, by Message Queue, Remote procedure Call, ...

5.4.4 Connexion Management class

Class CConnexionMngt

---

class **CConnexionMngt**

Class that manages all TCP Connections

Stereotype implementationClass

Field Summary	
protected char	**GlobalState** Global State of the Component ConnexionMngt
Int	**NbConnectionSupported** Global number of connections
Int	**NbLocalConnection** Number of connections opened by the local Client to a remote Server
Int	**NbRemoteConnection** Number of connections opened by a remote Client to the local Server

Constructor Summary	
CconnexionMngt(TConfigureObject * lnkConfigureObject) Constructor : Create internal object , initialize to 0 variables.	

Method Summary	
int	**m_EventOnSocket**(void) wake-up
bool	**m_IsConnectionAuthorized**(unsigned long IPAddress) Return true if new connection is authorized
int	**m_ReceiveData**(TItemConnexion * lnkConnexion) Interface with CTCPConnexion::write method for reading data from network return negative value if problem
bool	**m_Reset**(void) Method for Resetting ConnectionMngt component return true if Reset
int	**m_SendData**(TItemConnexion * lnkConnexion) Interface with CTCPConnexion::read method for sending data to the network Return negative value if problem
bool	**m_Start**(void) Method for Starting ConnectionMngt component return true if Started
bool	**m_Stop**(void) Method for Stopping component return true if Stopped

MODBUS over Serial Line

Specification and Implementation Guide

V1.02

Contents

Document modifications		
	Month-Year	Modifications
1.0	Nov 02	Creation. This document comprises a description of Master / slave protocol and of the two different transmission modes (RTU, ASCII). The main features of the physical layer (RS485, RS232) and some recommendations are provided. Implementation classes are proposed to guide the implementation.
1.01	Aug 30, 2006	Minor clarifications and correction of typos.
1.02	Dec 20, 2006	Minor clarifications and correction of typos.

1 Introduction

1.1 Scope of this document

The MODBUS standard defines an application layer messaging protocol, positioned at level 7 of the OSI model that provides "client/server" communications between devices connected on different types of buses or networks. It standardizes also a specific protocol on serial line to exchange MODBUS request between a master and one or several slaves.

The objective of this document is to present the MODBUS protocol over serial line, in order to be used by all system designers when they want to implement MODBUS protocol on their serial line products. Thus, this document will facilitate interoperability between devices using the MODBUS protocol.

This document comes in complement to the document called "MODBUS Application Protocol Specification".

In chapter 5 different implementation classes are defined for "MODBUS Serial Line". Specification of a class is the sum of requirements that a device must respect in order to belong to that class.

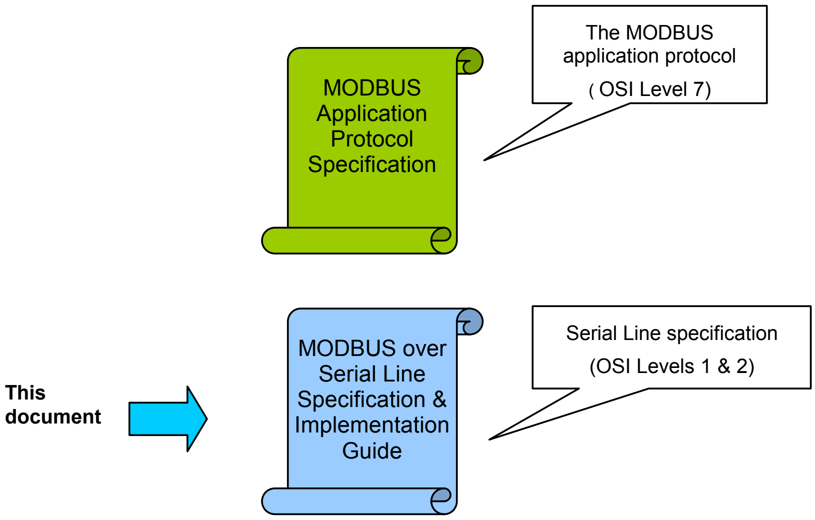


Figure 1: General overview of MODBUS documents

1.2 Protocol overview

This document describes the MODBUS over Serial Line protocol. **MODBUS Serial Line protocol is a Master-Slave protocol.** This protocol takes place at level 2 of the OSI model.

A master-slave type system has one node (the master node) that issues explicit commands to one of the "slave" nodes and processes responses. Slave nodes will not typically transmit data without a request from the master node, and do not communicate with other slaves.

At the physical level, MODBUS over Serial Line systems may use different physical interfaces (RS485, RS232). TIA/EIA-485 (RS485) Two-Wire interface is the most common. As an add-on option, RS485 Four-Wire interface may also be implemented. A TIA/EIA-232-E (RS232) serial interface may also be used as an interface, when only short point to point communication is required. (see chapter "Physical Layer")

The following figure gives a general representation of MODBUS serial communication stack compared to the 7 layers of the OSI model.

Layer	ISO/OSI Model	
7	Application	MODBUS Application Protocol
6	Presentation	Empty
5	Session	Empty
4	Transport	Empty
3	Network	Empty
2	Data Link	MODBUS Serial Line Protocol
1	Physical	EIA/TIA-485 (or EIA/TIA-232)

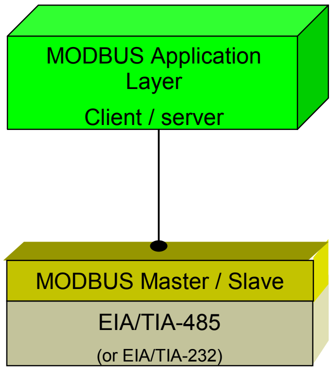


Figure 2: MODBUS Protocols and ISO/OSI Model

MODBUS application layer messaging protocol, positioned at level 7 of the OSI model, provides client/server communication between devices connected on buses or networks. On MODBUS serial line the client role is provided by the Master of the serial bus and the Slaves nodes act as servers.

1.3 Conventions

In this document, the following words are used to define the significance of each particular **requirement**.

- **"MUST" / "REQUIRED"**

All requirements containing the word "**MUST**" are mandatory. The word **MUST**, or the adjective "**REQUIRED**", means that the item is an absolute requirement of the implementation. These words are underlined.

- **"SHOULD" / "RECOMMENDED"**

All recommendations containing the word "**SHOULD**", or the adjective "**RECOMMENDED**", are considered desired behavior. These recommendations should be used as a guideline when choosing between different options to implement functionality. There may be valid reasons in particular circumstances to ignore this item, but the full implications should be understood and the case carefully weighed before choosing a different course. These words are underlined.

- **"MAY" / "OPTIONAL"**

The word "**MAY**", or the adjective "**OPTIONAL**", means that this item is truly optional. One designer may choose to include the item because a particular marketplace requires it or because it enhances the product, for example; another designer may omit the same item.

1.4 Compliance

An implementation is **not in conformity** if it fails to satisfy one or more of the **MUST** requirements from its implementation class.

An implementation that satisfies all the MUST requirements and all the SHOULD recommendations is said to be "**unconditionally compliant**".

One that satisfies all the MUST requirements but not all the SHOULD recommendations is said to be "**conditionally compliant**".

1.5 Glossary

Definition of particular words, symbols, and abbreviations used in this document.

2W	The Two-Wire configuration defined in the “Electrical Interface” chapter, or one of its interfaces.
4W	The Four-Wire configuration defined in the “Electrical Interface” chapter, or one of its interfaces.
AUI	Attachment Unit Interface
AWG	American Wire Gauge, a standard method denoting wire diameter; please see Appendix E - References.
Common	The Signal Common in EIA/TIA Standards. In a 2W-or 4W-RS485 MODBUS Network, Signal and optional Power Supply Common
DCE	a MODBUS Device, for example a programmable controller adapter, which implements an RS232 Data Circuit-terminating Equipment, also named Data Communication Equipment.
Device	or “MODBUS device” : see this definition.
Driver	Generator, or Transmitter.
DTE	a MODBUS Device, for example a programming panel or a PC, which implements an RS232 Data Terminal Equipment.
ITr	Physical bus Interface on Trunk side.
IDv	Physical bus Interface on Derivation (or tap or device drop) side.
LT	Line Termination.
MODBUS Device	a Device that implements MODBUS over Serial Line and respects this Technical Note.
RS232	EIA/ TIA -232 Standard.
RS485	EIA/ TIA -485 Standard.
RS485-MODBUS	A 2W-or 4W-Network in accordance with this Technical Note.
Transceiver	a **Trans**mitter and a Re**ceiver** (or Driver and Receiver).

2 MODBUS Data Link Layer

2.1 MODBUS Master / Slaves protocol principle

The MODBUS Serial Line protocol is a Master-Slaves protocol. Only one master (at the same time) is connected to the bus, and one or several (247 maximum number) slaves nodes are also connected to the same serial bus. A MODBUS communication is always initiated by the master. The slave nodes will never transmit data without receiving a request from the master node. The slave nodes will never communicate with each other. The master node initiates only one MODBUS transaction at the same time.

The master node issues a MODBUS request to the slave nodes in two modes :

➔ In **unicast mode**, the master addresses an individual slave. After receiving and processing the request, the slave returns a message (a 'reply') to the master .

In that mode, a MODBUS transaction consists of 2 messages : a request from the master, and a reply from the slave.

Each slave must have an unique address (from 1 to 247) so that it can be addressed independently from other nodes.

➔ In **broadcast mode**, the master can send a request to all slaves.

No response is returned to broadcast requests sent by the master. The broadcast requests are necessarily writing commands. **All devices must accept the broadcast for writing function**. The address 0 is reserved to identify a broadcast exchange.

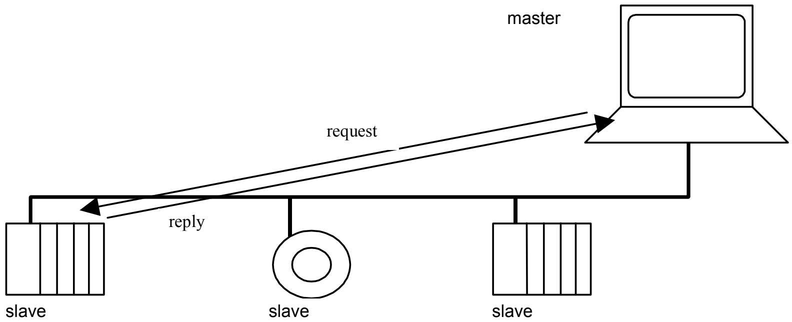


Figure 3: Unicast mode

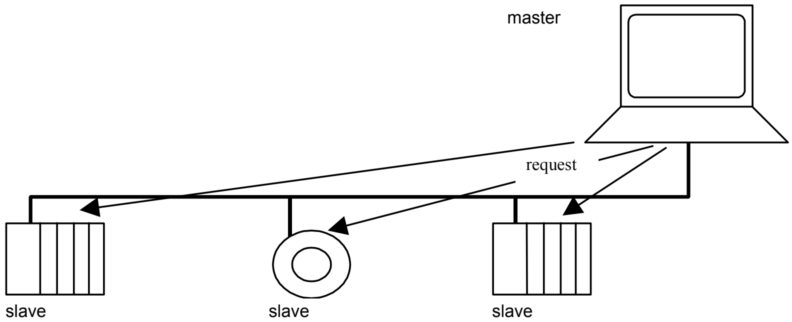


Figure 4: Broadcast mode

2.2 MODBUS Addressing rules

The MODBUS addressing space comprises 256 different addresses.

0	From 1 to 247	From 248 to 255
Broadcast address	**Slave individual addresses**	**Reserved**

The Address 0 is reserved as the broadcast address. All slave nodes must recognise the broadcast address.

The MODBUS Master node has no specific address, only the slave nodes must have an address. This address must be unique on a MODBUS serial bus.

2.3 MODBUS frame description

The MODBUS application protocol [1] defines a simple **P**rotocol **D**ata **U**nit (**PDU**) independent of the underlying communication layers:

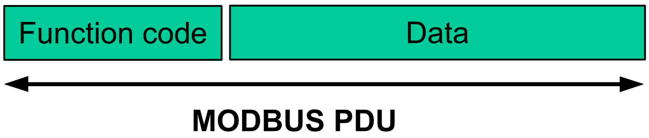


Figure 5: MODBUS Protocol Data Unit

The mapping of MODBUS protocol on a specific bus or network introduces some additional fields on the **P**rotocol **D**ata **U**nit. The client that initiates a MODBUS transaction builds the MODBUS PDU, and then adds fields in order to build the appropriate communication PDU.

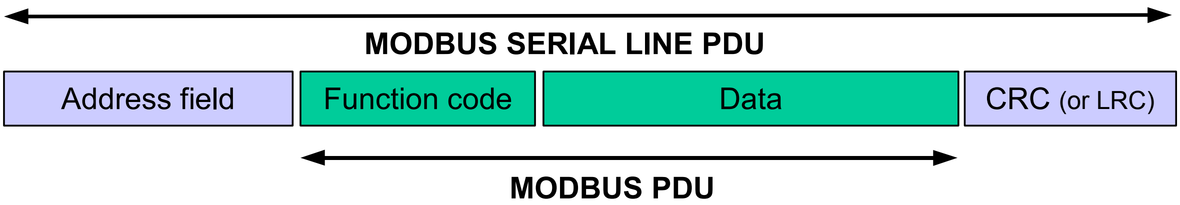


Figure 6: MODBUS frame over Serial Line

- On MODBUS Serial Line, the Address field only contains the slave address.

As described in the previous section the valid slave nodes addresses are in the range of 0 – 247 decimal. The individual slave devices are assigned addresses in the range of 1 – 247. A master addresses a slave by placing the slave address in the address field of the message. When the slave returns its response, it places its own address in the response address field to let the master know which slave is responding.

- The function code indicates to the server what kind of action to perform. The function code can be followed by a data field that contains request and response parameters.

- Error checking field is the result of a "Redundancy Checking" calculation that is performed on the message contents. Two kinds of calculation methods are used depending on the transmission mode that is being used (RTU or ASCII). (see 2.5 section, "*The two serial Transmission Modes*")

2.4 Master / Slaves State Diagrams

The MODBUS data link layer comprises two separate sub layers :

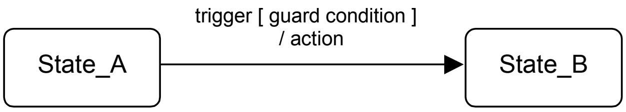
- The Master / slave protocol
- The transmission mode (RTU vs ASCII modes)

The following sections describes the state diagrams of a master and a slave that are independent of transmission modes used.

The RTU and ASCII transmission modes are specified in next chapters using two state diagrams. The reception and the sending of a frame are described.

Syntax of state diagram :

The following state diagrams are drawn in compliance with UML standard notations. The notation is briefly recalled below :



When a "trigger" event occurs in a system being in "State_A", system is going into "State_B", only if "guard condition" is true. An action "action" is then performed.

2.4.1 Master State diagram

The following drawing explains the Master behavior :

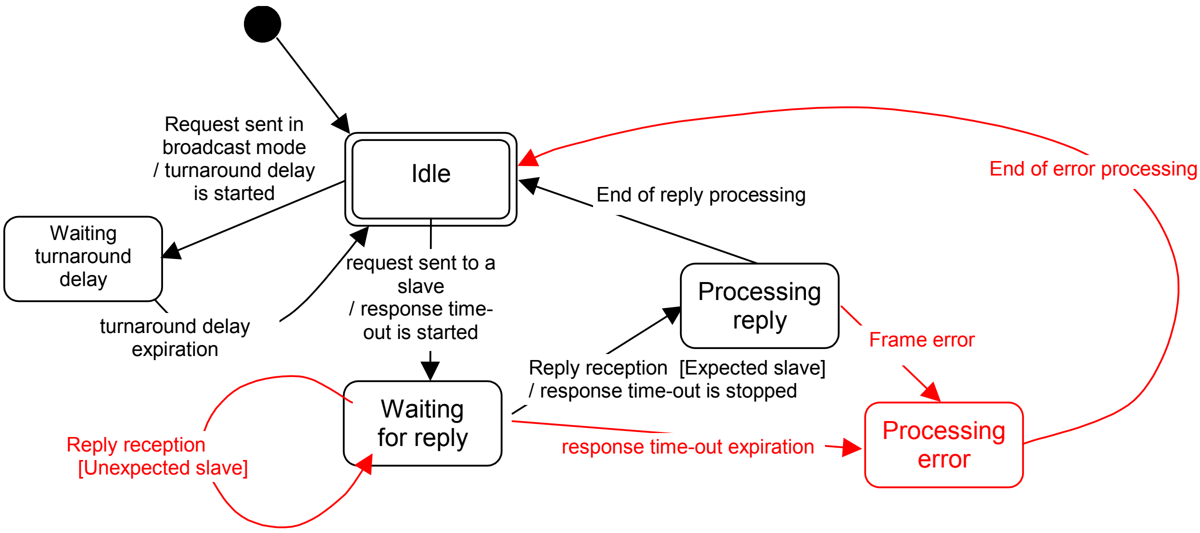


Figure 7: Master state diagram

<u>Some explanations about the state diagram above</u> :

- State "Idle" = no pending request. This is the initial state after power-up. A request can only be sent in "Idle" state. After sending a request, the Master leaves the "Idle" state, and cannot send a second request at the same time
- When a unicast request is sent to a slave, the master goes into "Waiting for reply" state, and a "Response Time-out" is started. It prevents the Master from staying indefinitely in "Waiting for reply" state. Value of the Response time-out is application dependant.
- When a reply is received, the Master checks the reply before starting the data processing. The checking may result in an error, for example a reply from an unexpected slave, or an error in the received frame. In case of a reply received from an unexpected slave, the Response time-out is kept running. In case of an error detected on the frame, a retry <u>may</u> be performed.
- If no reply is received, the Response time-out expires, and an error is generated. Then the Master goes into "Idle" state, enabling a retry of the request. The maximum number of retries depends on the master set-up.

- When a broadcast request is sent on the serial bus, no response is returned from the slaves. Nevertheless a delay is respected by the Master in order to allow any slave to process the current request before sending a new one. This delay is called "Turnaround delay". Therefore the master goes into "Waiting Turnaround delay" state before going back in "idle" state and before being able to send another request.
- In unicast the Response time out must be set long enough for any slave to process the request and return the response, in broadcast the Turnaround delay must be long enough for any slave to process only the request and be able to receive a new one. Therefore the Turnaround delay should be shorter than the Response time-out. Typically the Response time-out is from 1s to several second at 9600 bps; and the Turnaround delay is from 100 ms to 200ms.
- Frame error consists of : 1) Parity checking applied to each character; 2) Redundancy checking applied to the entire frame. See §2.6 "*Error Checking Methods*" for more explanations.

The state diagram is intentionally very simple. It does not take into account access to the line, message framing, or retry following transmission error, etc … For more details about frame transmission, please refer to 2.5 paragraph, "*The two serial Transmission Modes*".

2.4.2 Slave State Diagram

The following drawing explains the Slave behavior :

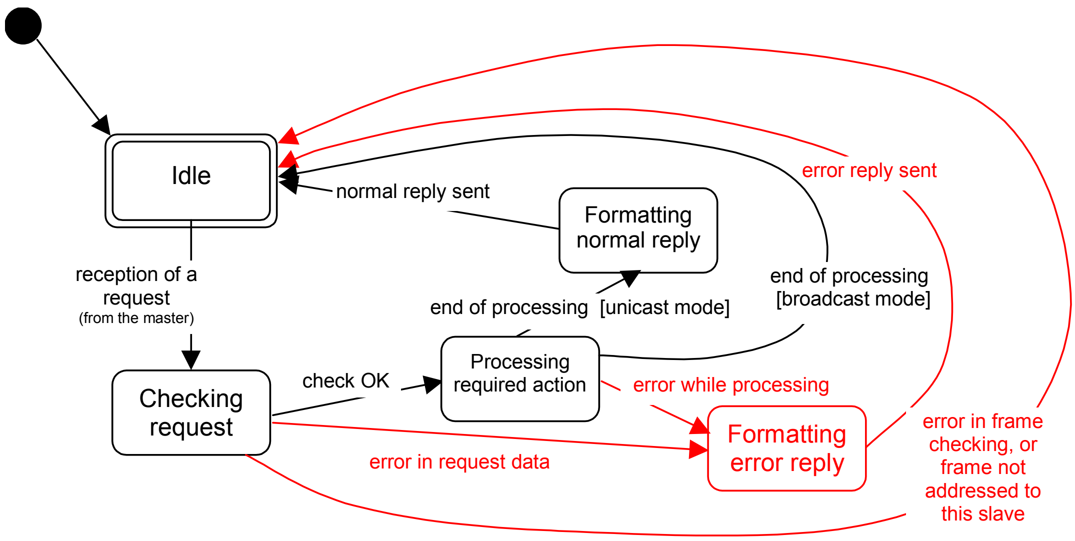


Figure 8: Slave state diagram

Some explanations about the above state diagram :

- State "Idle" = no pending request. This is the initial state after power-up.
- When a request is received, the slave checks the packet before performing the action requested in the packet. Different errors may occur : format error in the request, invalid action, … In case of error, a reply must be sent to the master.
- Once the required action has been completed, a unicast message requires that a reply must be formatted and sent to the master.
- If the slave detects an error in the received frame, no respond is returned to the master.
- MODBUS diagnostics counters are defined and should be managed by any slave in order to provide diagnostic information. These counters can be get using the Diagnostic MODBUS function (see Appendix A, and the MODBUS application protocol specification [1]).

2.4.3 Master / Slave communication time diagram

This following figure shows the time diagram of 3 typical scenarios of Master / Slave communications.

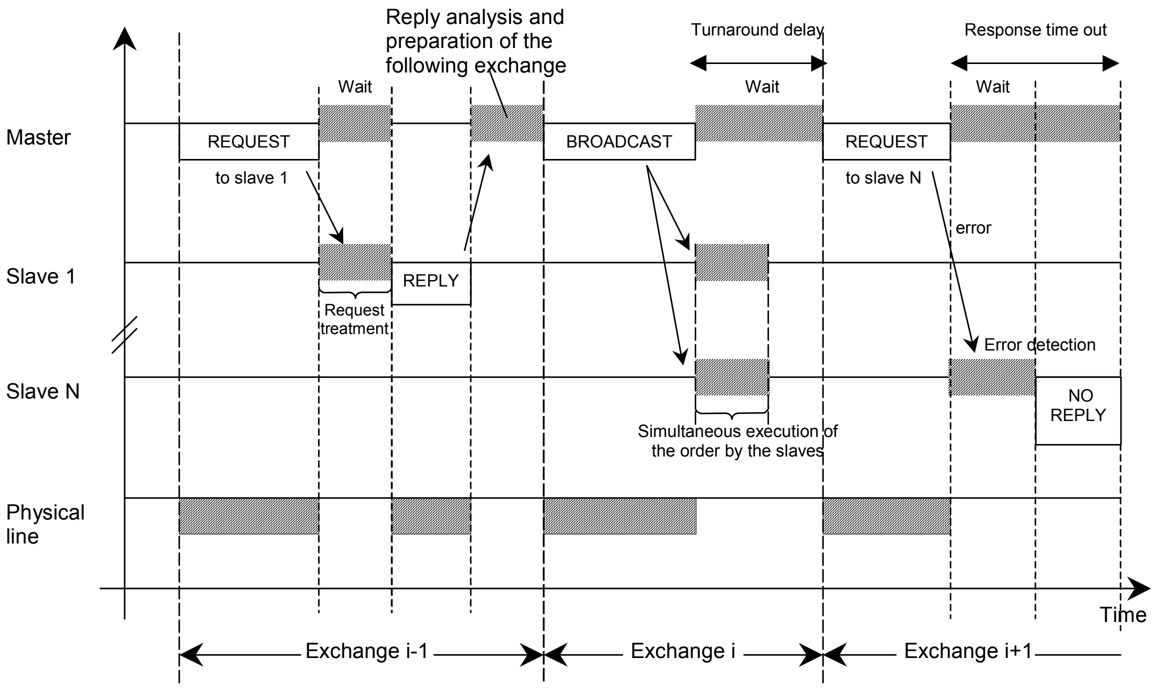


Figure 9: Master / Slave scenario time diagram

Remarks :

- the duration of the REQUEST, REPLY, BROACAST phases depends on the communication features (frame length and throughput).
- the duration of the WAIT and TREATMENT phases depends on the request processing time needed for the slave application.

2.5 The two serial Transmission Modes

Two different serial transmission modes are defined : The RTU mode and the ASCII mode.

It defines the bit contents of message fields transmitted serially on the line. It determines how information is packed into the message fields and decoded.

The transmission mode (and serial port parameters) must be the same for all devices on a MODBUS Serial Line.

Although the ASCII mode is required in some specific applications, interoperability between MODBUS devices can be reached only if each device has the same transmission mode : **All devices must implement the RTU Mode.** The ASCII transmission mode is an option.

Devices should be set up by the users to the desired transmission mode, RTU or ASCII. Default setup must be the RTU mode.

2.5.1 RTU Transmission Mode

When devices communicate on a MODBUS serial line using the RTU (Remote Terminal Unit) mode, each 8–bit byte in a message contains two 4–bit hexadecimal characters. The main advantage of this mode is that its greater character density allows better data throughput than ASCII mode for the same baud rate. Each message must be transmitted in a continuous stream of characters.

The format (11 bits) for each byte in RTU mode is :

Coding System: 8–bit binary

Bits per Byte: 1 start bit
8 data bits, least significant bit sent first
1 bit for parity completion
1 stop bit

Even parity is required, other modes (odd parity, no parity) may also be used. In order to ensure a maximum compatibility with other products, it is recommended to support also No parity mode. The default parity mode must be even parity.

Remark : the use of no parity requires 2 stop bits.

How Characters are Transmitted Serially :

Each character or byte is sent in this order (left to right):

Least Significant Bit (LSB) . . . Most Significant Bit (MSB)

With Parity Checking

Start	1	2	3	4	5	6	7	8	Par	Stop

Figure 10: Bit Sequence in RTU mode

Devices may accept by configuration either Even, Odd, or No Parity checking. If No Parity is implemented, an additional stop bit is transmitted to fill out the character frame to a full 11-bit asynchronous character :

Without Parity Checking

Start	1	2	3	4	5	6	7	8	Stop	Stop

Figure 11: Bit Sequence in RTU mode (specific case of No Parity)

Frame Checking Field : Cyclical Redundancy Checking (CRC)

<u>Frame description</u> :

Slave Address	Function Code	Data	CRC
1 byte	1 byte	0 up to 252 byte(s)	2 bytes CRC Low CRC Hi

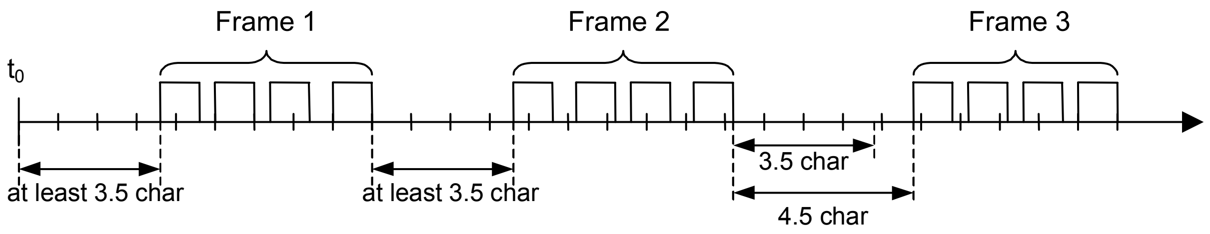
Figure 12: RTU Message Frame

➔ The maximum size of a MODBUS RTU frame is 256 bytes.

2.5.1.1 MODBUS Message RTU Framing

A MODBUS message is placed by the transmitting device into a frame that has a known beginning and ending point. This allows devices that receive a new frame to begin at the start of the message, and to know when the message is completed. Partial messages <u>must</u> be detected and errors <u>must</u> be set as a result.

In RTU mode, message frames are separated by a silent interval of <u>at least</u> 3.5 character times. In the following sections, this time interval is called t3,5.

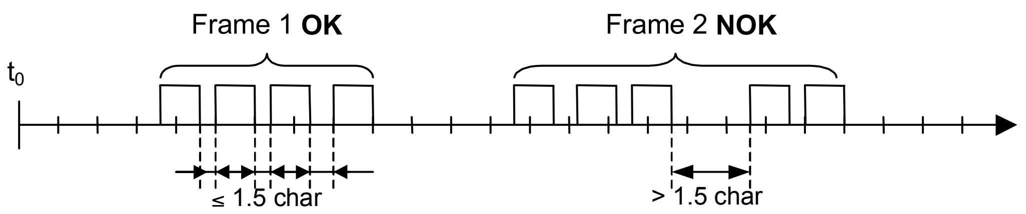


Start	MODBUS message				End
Start	Address	Function	Data	CRC Check	End
≥ 3.5 char	8 bits	8 bits	N x 8 bits	16 bits	≥ 3.5 char

Figure 13: RTU Message Frame

The entire message frame <u>must</u> be transmitted as a continuous stream of characters.

If a silent interval of more than 1.5 character times occurs between two characters, the message frame is declared incomplete and should be discarded by the receiver.



<u>Remark :</u>

The implementation of RTU reception driver may imply the management of a lot of interruptions due to the $t_{1.5}$ and $t_{3.5}$ timers. With high communication baud rates, this leads to a heavy CPU load. Consequently these two timers <u>must</u> be strictly respected when the baud rate is equal or lower than 19200 Bps. For baud rates greater than 19200 Bps, fixed values for the 2 timers should be used: it is recommended to use a value of 750µs for the inter-character time-out ($t_{1.5}$) and a value of 1.750ms for inter-frame delay ($t_{3.5}$).

The following drawing provides a description of the RTU transmission mode state diagram. Both "master" and "slave" points of view are expressed in the same drawing :

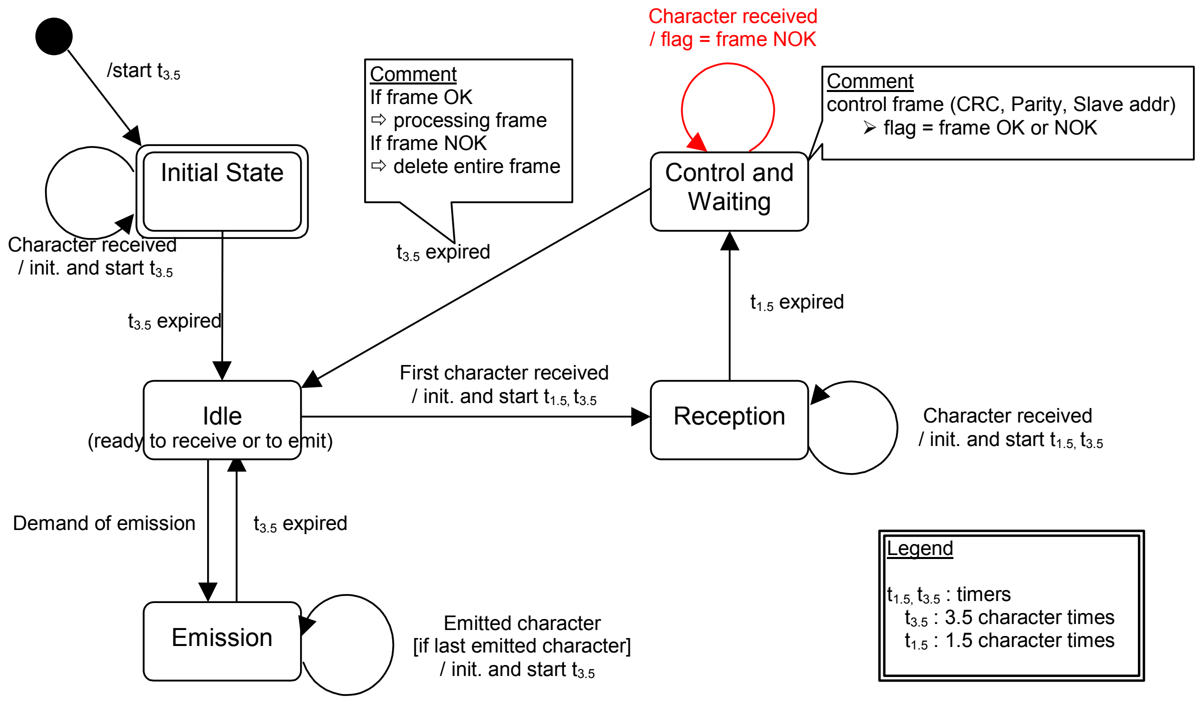


Figure 14: RTU transmission mode state diagram

Some explanations about the above state diagram:

- Transition from "Initial State" to "Idle" state needs $t_{3.5}$ time-out expiration : that insures inter-frame delay
- "Idle" state is the normal state when neither emission nor reception is active.
- In RTU mode, the communication link is declared in "idle" state when there is no transmission activity after a time interval equal to at least 3,5 characters.
- When the link is in idle state, each transmitted character detected on the link is identified as the **start of a frame**. The link goes to the "active" state. Then, the **end of frame** is identified when no more character is transmitted on the link after the time interval t3,5.
- After detection of the end of frame, the CRC calculation and checking is completed. Afterwards the address field is analysed to determine if the frame is for the device. If not the frame is discarded. In order to reduce the reception processing time the address field can be analysed as soon as it is received without waiting the end of frame. In this case the CRC will be calculated and checked only if the frame is addressed to the slave (broadcast frame included).

2.5.1.2 CRC Checking

The RTU mode includes an error–checking field that is based on a Cyclical Redundancy Checking (**CRC**) method performed on the message contents.

The CRC field checks the contents of the entire message. It is applied regardless of any parity checking method used for the individual characters of the message.

The CRC field contains a 16–bit value implemented as two 8–bit bytes.

The CRC field is appended to the message as the last field in the message. When this is done, the low–order byte of the field is appended first, followed by the high–order byte. The CRC high–order byte is the last byte to be sent in the message.

The CRC value is calculated by the sending device, which appends the CRC to the message. The receiving device recalculates a CRC during receipt of the message, and compares the calculated value to the actual value it received in the CRC field. If the two values are not equal, an error results.

The CRC calculation is started by first pre-loading a 16–bit register to all 1's. Then a process begins of applying successive 8–bit bytes of the message to the current contents of the register. Only the eight bits of data in each character are used for generating the CRC. Start and stop bits and the parity bit, do not apply to the CRC.

During generation of the CRC, each 8–bit character is exclusive ORed with the register contents. Then the result is shifted in the direction of the least significant bit (LSB), with a zero filled into the most significant bit (MSB) position. The LSB is extracted and examined. If the LSB was a 1, the register is then exclusive ORed with a preset, fixed value. If the LSB was a 0, no exclusive OR takes place.

This process is repeated until eight shifts have been performed. After the last (eight) shift, the next 8–bit byte is exclusive ORed with the register's current value, and the process repeats for eight more shifts as described above. The final content of the register, after all the bytes of the message have been applied, is the CRC value.

When the CRC is appended to the message, the low-order byte is appended first, followed by the high-order byte. A detailed example of CRC generation is contained in Appendix B.

2.5.2 The ASCII Transmission Mode

When devices are setup to communicate on a MODBUS serial line using ASCII (American Standard Code for Information Interchange) mode, each 8–bit byte in a message is sent as two ASCII characters. This mode is used when the physical communication link or the capabilities of the device does not allow the conformance with RTU mode requirements regarding timers management.

Remark : this mode is less efficient than RTU since each byte needs two characters.

➔ Example : The byte 0X5B is encoded as two characters : 0x35 and 0x42 (0x35 ="5", and 0x42 ="B" in ASCII).

The format (10 bits) for each byte in ASCII mode is :

Coding System: Hexadecimal, ASCII characters 0–9, A–F
One hexadecimal character contains 4-bits of data within each ASCII character of the message

Bits per Byte: 1 start bit
7 data bits, least significant bit sent first
1 bit for parity completion;
1 stop bit

Even parity is required, other modes (odd parity, no parity) may also be used. In order to ensure a maximum compatibility with other products, it is recommended to support also No parity mode. The default parity mode must be Even parity.

Remark : the use of no parity requires 2 stop bits.

How Characters are Transmitted Serially :

Each character or byte is sent in this order (left to right):

Least Significant Bit (LSB) . . . Most Significant Bit (MSB)

With Parity Checking

Start	1	2	3	4	5	6	7	Par	Stop

Figure 15: Bit Sequence in ASCII mode

Devices may accept by configuration either Even, Odd, or No Parity checking. If No Parity is implemented, an additional stop bit is transmitted to fill out the character frame :

Without Parity Checking

Start	1	2	3	4	5	6	7	Stop	Stop

Figure 16: Bit Sequence in ASCII mode (specific case of No Parity)

Frame Checking Field: Longitudinal Redundancy Checking (LRC)

2.5.2.1 MODBUS Message ASCII Framing

A MODBUS message is placed by the transmitting device into a frame that has a known beginning and ending point. This allows devices that receive a new frame to begin at the start of the message, and to know when the message is completed. Partial messages must be detected and errors must be set as a result.

The address field of a message frame contains two characters.

In ASCII mode, a message is delimited by specific characters as Start-of-frames and End-of-frames. A message must start with **a 'colon' (:)** character (ASCII 3A hex), and end with a **'carriage return – line feed'** (CRLF) pair (ASCII 0D and 0A hex).

Remark : The LF character can be changed using a specific MODBUS application command (see MODBUS application protocol specification).

The allowable characters transmitted for all other fields are hexadecimal 0–9, A–F (ASCII coded). The devices monitor the bus continuously for the 'colon' character. When this character is received, each device decodes the next character until it detects the End-Of-Frame.

Intervals of up to one second may elapse between characters within the message. Unless the user has configured a longer timeout, an interval greater than 1 second means an error has occurred. Some Wide-Area-Network application may require a timeout in the 4 to 5 second range.

A typical message frame is shown below.

Start	Address	Function	Data	LRC	End
1 char :	2 chars	2 chars	0 up to 2x252 char(s)	2 chars	2 chars CR,LF

Figure 17: ASCII Message Frame

Remark : Each data byte needs two characters for encoding. Thus, to ensure compatibility at MODBUS application level between ASCII mode and RTU mode, the maximum data size for ASCII data field (2x252) is the double the maximum data size for RTU data field (252). Consequently, the maximum size of a MODBUS ASCII frame is 513 characters.

The ASCII framing requirements are synthesized in the following state diagram. Both "master" and "slave" points of view are expressed in the same drawing :

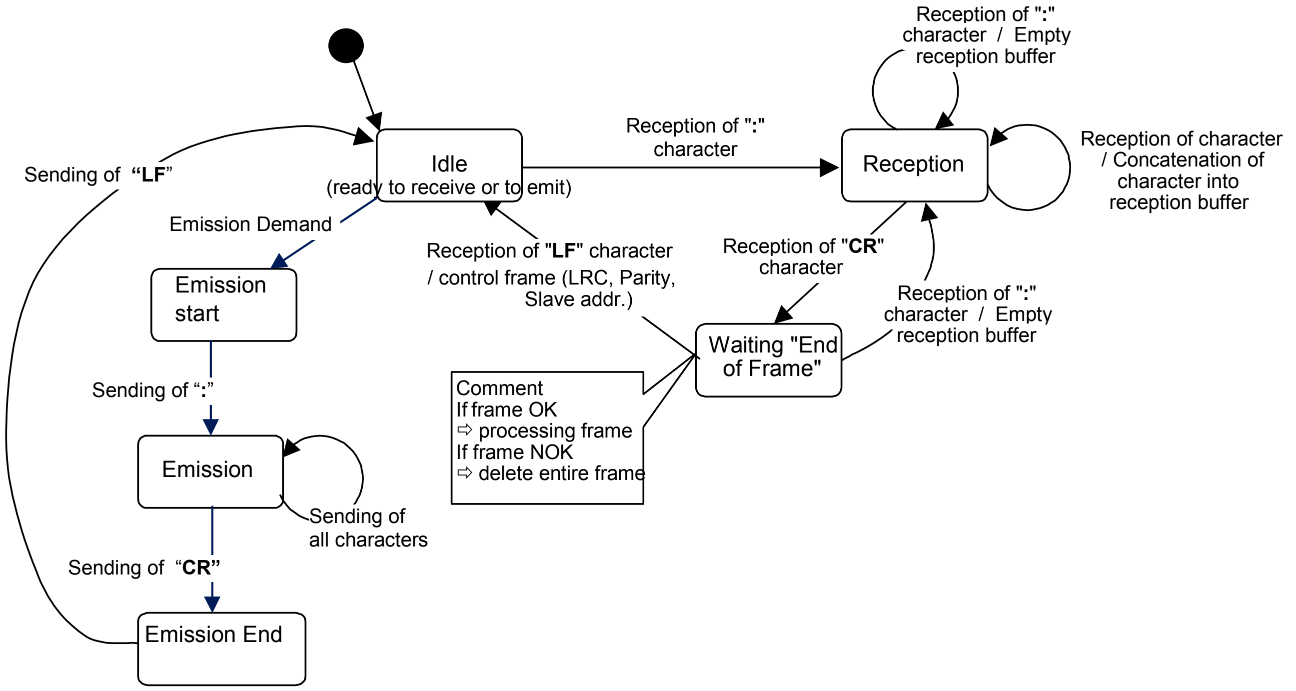


Figure 18: ASCII Transmission mode State diagram

Some explanations about the above state diagram :

- "Idle" state is the normal state when neither emission nor reception is active.
- Each reception of a ":" character means a beginning of a new message. If a message was in process of reception while receiving such a character, the current message is declared incomplete and it is discarded. A new reception buffer is then allocated.
- After detection of the end of frame, the LRC calculation and checking is completed. Afterwards the address field is analyzed to determine if the frame is for the device. If not the frame is discarded. In order to reduce the reception processing time the address field can be analyzed as soon as it is reserved without waiting the end of frame.

2.5.2.2 LRC Checking

In ASCII mode, messages include an error–checking field that is based on a Longitudinal Redundancy Checking (**LRC**) calculation that is performed on the message contents, exclusive of the beginning 'colon' and terminating CRLF pair characters. It is applied regardless of any parity checking method used for the individual characters of the message.

The LRC field is one byte, containing an 8–bit binary value. The LRC value is calculated by the device that emits, which appends the LRC to the message. The device that receives calculates an LRC during receipt of the message, and compares the calculated value to the actual value it received in the LRC field. If the two values are not equal, an error results.

The LRC is calculated by adding together successive 8–bit bytes of the message, discarding any carries, and then two's complementing the result. It is performed on the bytes of the message, before the encoding of each byte in the two ASCII characters corresponding to the hexadecimal representation of each nibble. The computation does not include the 'colon' character that begins the message, and does not include the CRLF pair at the end of the message.
The resulting LRC is ASCII encoded into two bytes and placed at the end of the ASCII mode frame before the CRLF.

A detailed example of LRC generation is contained in Appendix B.

2.6 Error Checking Methods

The security of standard MODBUS Serial Line is based on two kinds of error checking :

- Parity checking (even or odd) should be applied to each character.
- Frame checking (LRC or CRC) must be applied to the entire message.

Both the character checking and message frame checking are generated in the device (master or slave) that emits and applied to the message contents before transmission. The device (slave or master) checks each character and the entire message frame during receipt.

The master is configured by the user to wait for a predetermined timeout interval (Response time-out) before aborting the transaction. This interval is set to be long enough for any slave to respond normally (unicast request). If the slave detects a transmission error, the message will not be acted upon. The slave will not construct a response to the master. Thus the timeout will expire and allow the master's program to handle the error. Note that a message addressed to a nonexistent slave device will also cause a timeout.

2.6.1 Parity Checking

Users may configure devices for Even (required) or Odd Parity checking, or for No Parity checking (recommended). This will determine how the parity bit will be set in each character.

If either Even or Odd Parity is specified, the quantity of 1 bits will be counted in the data portion of each character (seven data bits for ASCII mode, or eight for RTU). The parity bit will then be set to a 0 or 1 to result in an Even or Odd total of 1 bits.

For example, these eight data bits are contained in an RTU character frame:

1100 0101

The total quantity of 1 bits in the frame is four. If Even Parity is used, the frame's parity bit will be a 0, making the total quantity of 1 bits still an even number (four). If Odd Parity is used, the parity bit will be a 1, making an odd quantity (five).

When the message is transmitted, the parity bit is calculated and applied to the frame of each character. The device that receives counts the quantity of 1 bits and sets an error if they are not the same as configured for that device (all devices on the MODBUS Serial Line must be configured to use the same parity checking method).

Note that parity checking can only detect an error if an odd number of bits are picked up or dropped in a character frame during transmission. For example, if Odd Parity checking is employed, and two 1 bits are dropped from a character containing three 1 bits, the result is still an odd count of 1 bits.

If No Parity checking is specified, no parity bit is transmitted and no parity checking can be made. An additional stop bit is transmitted to fill out the character frame.

2.6.2 Frame Checking

Two kinds of frame checking is used depending on the transmission mode, RTU or ASCII.

- In RTU mode, messages include an error–checking field that is based on a Cyclical Redundancy Checking (CRC) method. The CRC field checks the contents of the entire message. It is applied regardless of any parity checking method used for the individual characters of the message.
- In ASCII mode, messages include an error–checking field that is based on a Longitudinal Redundancy Checking (LRC) method. The LRC field checks the contents of the message, exclusive of the beginning 'colon' and ending CRLF pair. It is applied regardless of any parity checking method used for the individual characters of the message.

The detailed information about error checking methods is contained in the previous sections.

3 Physical Layer

3.1 Preamble

A new MODBUS solution over serial line should implement an electrical interface in accordance with EIA/TIA-485 standard (also known as RS485 standard). This standard allows point to point and multipoint systems, in a "two-wire configuration". In addition, some devices may implement a "Four-Wire" RS485-Interface.

A device may also implement an RS232-Interface.

In such a MODBUS system, a Master Device and one or several Slave Devices communicate on a passive serial line.

On standard MODBUS system, all the devices are connected (in parallel) on a trunk cable constituted by 3 conductors. Two of those conductors (the "Two-Wire" configuration) form a balanced twisted pair, on which bi-directional data are transmitted, typically at the bit rate of 9600 bits per second.

Each device may be connected (see figure 19):

- either directly on the trunk cable, forming a daisy-chain,
- either on a passive Tap with a derivation cable,
- either on an active Tap with a specific cable.

Screw Terminals, RJ45, or D-shell 9 connectors may be used on devices to connect cables (see the chapter "Mechanical Interfaces").

3.2 Data Signaling Rates

9600 bps and 19.2 Kbps are required and 19.2 is the required default

Other baud rates may optionally be implemented : 1200, 2400, 4800, … 38400 bps, 56 Kbps, 115 Kbps, …

Every implemented baud rate must be respected better than 1% in transmission situation, and must accept an error of 2% in reception situation.

3.3 Electrical Interfaces

3.3.1 Multipoint Serial Bus Infrastructure

Figure 19 gives a general overview of the serial bus infrastructure in a MODBUS multipoint Serial Line system.

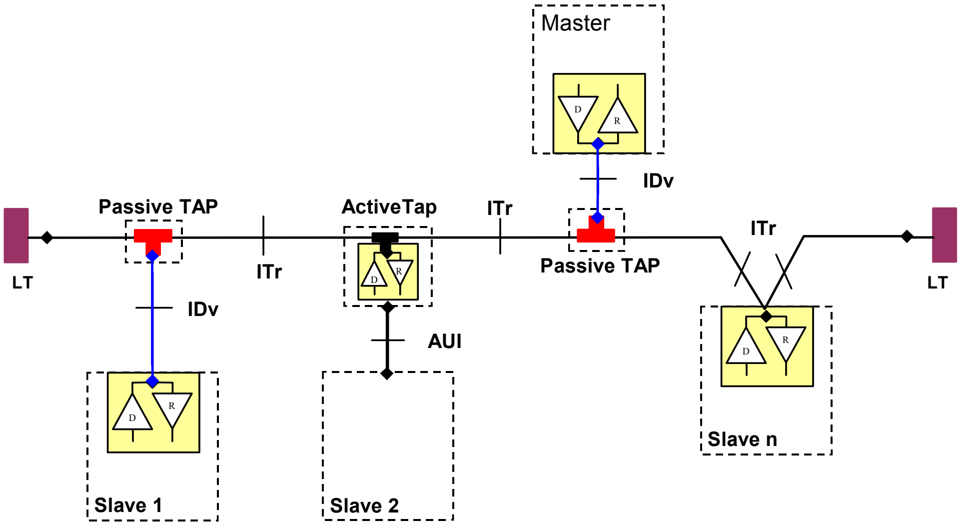


Figure 19 : Serial bus infrastructure

A multipoint MODBUS Serial Line bus is made of a principal cable (**the Trunk**), and possibly some **derivation** cables.
Line terminations are necessary at each extremity of the trunk cable for impedance adaptation (see § "Two-Wire MODBUS Definition" & "Optional Four-Wire MODBUS Definition" for details).

As shown in figure 19, different implementations may operate in the same MODBUS Serial Line system :

- the device integrates the communication transceiver and is connected to the trunk using a **Passive Tap** and a derivation cable (case of Slave 1 and Master) ;
- the device doesn't integrate the communication transceiver and is connected to the trunk using an **Active Tap** and a derivation cable (the active TAP integrates the transceiver) (case of Slave 2) ;
- the device is connected directly to the trunk cable, in a **Daisy-Chain** (case of Slave n)

The following conventions are adopted :

- The interface with the **trunk** is named **ITr** (Trunk Interface)
- The interface between the device and the **Passive Tap** is named **IDv** (Derivation Interface)
- The interface between the device and the **Active Tap** is named **AUI** (Attachment Unit Interface)

Remarks :

1. In some cases, the Tap may be connected directly to the IDv-socket or the AUI-socket of the device, without using a derivation cable.
2. A Tap may have several IDv sockets to connect several devices. Such a Tap is named **Distributor** when it is a passive one.
3. When using an active Tap, power supply of the Tap may be provided either via its AUI or ITr interface.

ITr and **IDv** interfaces are described in the following chapters (see § "Two-Wire MODBUS DEFINITION" & "Four-Wire MODBUS DEFINITION").

3.3.2 Two-Wire MODBUS Definition

A MODBUS solution over serial line should implement a “Two-Wire” electrical interface in accordance with EIA/TIA-485 standard.

On such a 2W-bus, at any time one driver only has the right for transmitting.

In fact a third conductor must also interconnect all the devices of the bus : the common.

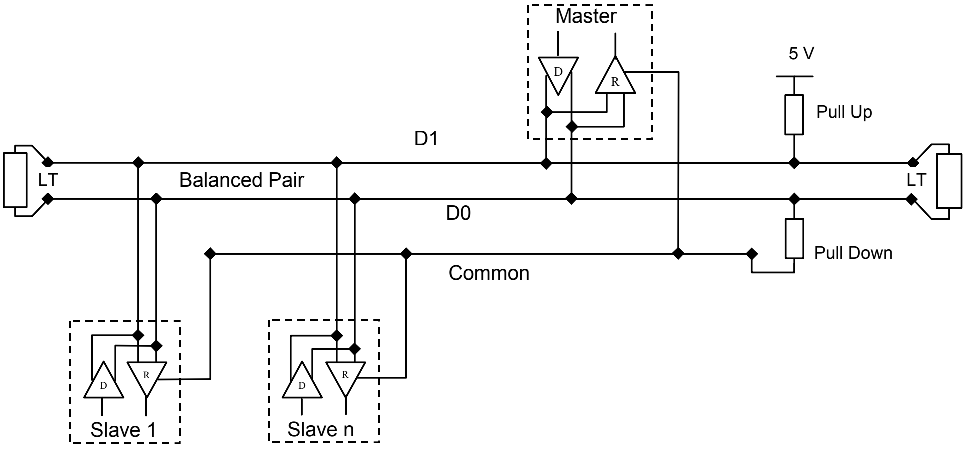


Figure 20: General 2-Wire Topology

2W-MODBUS Circuits Definition

Required Circuits		**For device**	**Required on device**	**EIA/TIA-485 name**	**Description**
on ITr	**on IDv**				
D1	D1	I/O	**X**	B/B’	**Transceiver terminal 1, V1 Voltage** (V1 > V0 for binary 1 [OFF] state)
D0	D0	I/O	**X**	A/A’	**Transceiver terminal 0, V0 Voltage** (V0 > V1 for binary 0 [ON] state)
Common	Common	--	**X**	C/C’	Signal and optional Power Supply Common

Notes :

- For Line Termination (LT), Pull Up and Pull Down resistors, please refer to section “Multipoint System requirements".
- D0, D1, and Common circuit names must be used in the documentation related to the device and the Tap (User Guide, Cabling Guide, …) to facilitate interoperability.
- Optional electrical interfaces may be added, for example :
  - **Power Supply :** 5..24 V D.C.
  - **Port mode control :** PMC circuit (TTL compatible). When needed, port mode may be controlled either by this external circuit and/or by another way (a switch on the device for example). In the first case while an open circuit PMC will ask for the 2W-MODBUS mode, a Low level on PMC will switch the port into 4W-MODBUS or RS232-MODBUS Mode, depending on the implementation.

3.3.3 Optional Four-Wire MODBUS Definition

Optionally, such MODBUS devices also permit to implement a **2-pair** bus (4 wires) of mono directional data. The data on the **master pair** (RXD1-RXD0) are only received by the slaves ; the data on the **slave pair** (TXD1-TXD0) are only received by the only master.

In fact a fifth conductor must also interconnect all the devices of the 4W-bus : the common.

In the same way as on a 2W-MODBUS, at any time one driver only has the right for emitting.

Such a device must implement, for each balanced pair, a driver and a transceiver **in accordance with EIA/ TIA-485**. (Sometimes this solution has been named "RS422", which is not correct : the RS422 standard does not support several drivers on one balanced pair.)

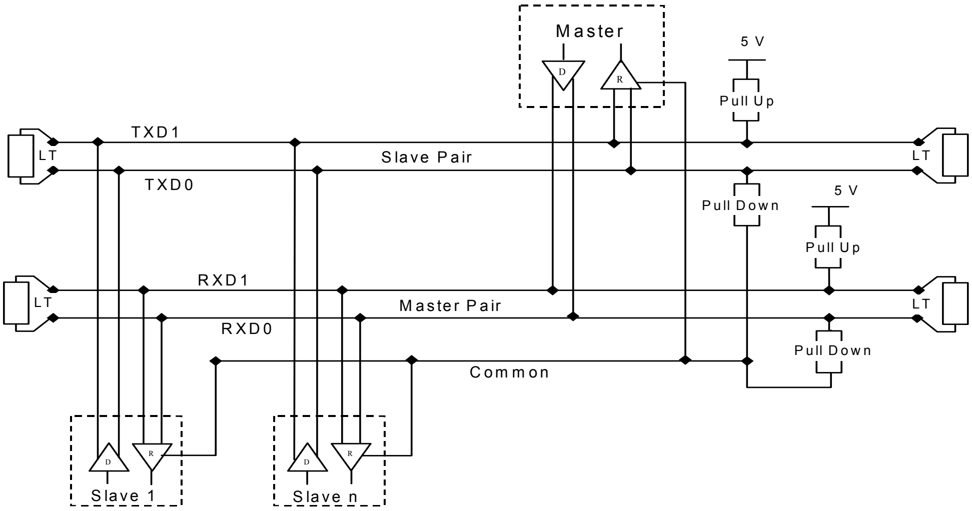


Figure 21: General 4-wire topology

Optional 4W-MODBUS Circuits Definition

Required Circuits on ITr	**Required Circuits on IDv**	**For device**	**Required on device**	**EIA/TIA-485 name**	**Description for IDv**
TXD1	TXD1	Out	**X**	B	**Generator terminal 1,** Vb Voltage (Vb > Va for binary 1 [OFF] state)
TXD0	TXD0	Out	**X**	A	**Generator terminal 0**, Va Voltage (Va > Vb for binary 0 [ON] state)
RXD1	RXD1	In	**(1)**	B'	**Receiver terminal 1**, Vb' Voltage (Vb' > Va' for binary 1 [OFF] state)
RXD0	RXD0	In	**(1)**	A'	**Receiver terminal 0**, Va' Voltage (Va' > Vb' for binary 0 [ON] state)
Common	Common	--	**X**	C/C'	Signal and optional Power Supply **Common**

Notes :

- For Line Termination (LT), Pull Up and Pull Down resistors, please refer to section "Multipoint System requirements".
- Those circuits **(1)** are required only if an 4W-MODBUS option is implemented.
- The name of the 5 required circuits must be used in the documentation related to the device and the Tap (User Guide, Cabling Guide, …) to facilitate interoperability.
- Optional electrical interfaces may be added, for example :
  - **Power Supply :** 5..24 V D.C.
  - **PMC circuit :** See above (In 2W-MODBUS Circuits Definition) the note about this optional circuit.

3.3.3.1 4W-Cabling System Important Topic

In such a 4W-MODBUS, Master Device and Slave Devices have IDv interfaces with the same 5 required circuits.
As the master has to :

- receive from the slave the data on the slave pair (TXD1-TXD0),
- and transmit on the master pair (RXD1-RXD0 , received by the slaves) ,

the 4W-cabling system <u>must</u> cross the two pairs of the bus between ITr and the IDv of the master :

	Signal on Master IDv		EIA/TIA-485 Name	**Circuit on ITr**
	Name	Type		
Slave Pair	RXD1	In	B'	TXD1
	RXD0	In	A'	TXD0
Master Pair	TXD1	Out	B	RXD1
	TXD0	Out	A	RXD0
	Common	--	C/C'	Common

This crossing may be implemented by crossed cables, but the connection of such crossed cables in a 2-wire system may cause damages. To connect a 4W master device (which have a MODBUS connector) a better solution is to use a Tap which includes the crossing function.

3.3.3.2 Compatibility between 4-Wire and 2-Wire cabling

In order to connect devices implementing a 2-Wire physical interface to an already existing 4-Wire system, the 4-Wire cabling system can be modified as described below :

- TxD0 signal shall be wired with the RxD0 signal, turning them to the D0 signal
- TxD1 signal shall be wired with the RxD1 signal, turning them to the D1 signal.
- Pull-up, Pull-down and line terminations resistors shall be re-arranged to correctly adapt the D0, D1 signals.

The figure hereafter gives an example where slaves 2 and 3 which use a 2-Wire interface can operate with the Master and the slave 1 which use a 4-Wire interface.

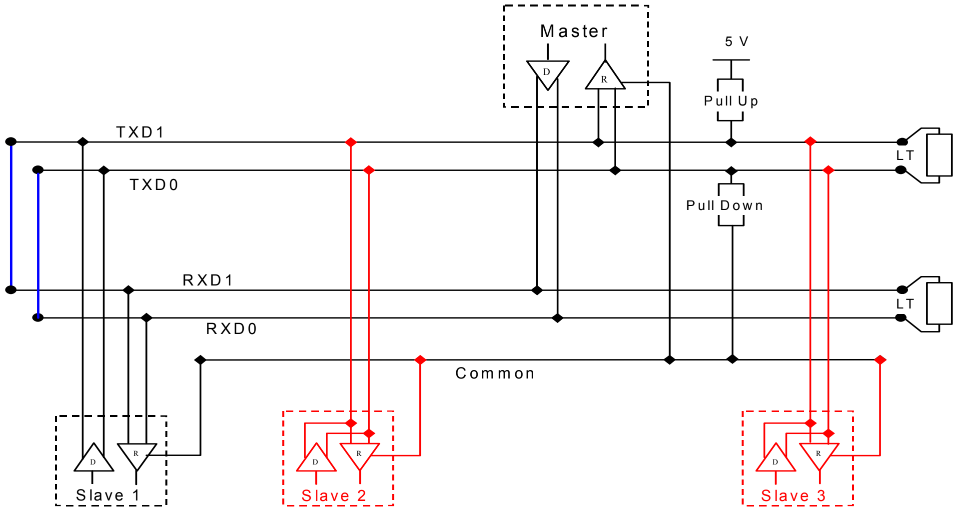


Figure 22 : Changing a 4-Wire cabling system into a 2-Wire cabling system

In order to connect devices implementing a 4-Wire physical interface to an already existing 2-Wire system, the 4-Wire interface of the new coming devices can be arranged as describe below :

On each 4-Wire device interface :

- TxD0 signal shall be wired with the RxD0 signal and then connected to the D0 signal of the trunk ;
- TxD1 signal shall be wired with the RxD1 signal and then connected to the D1 signal of the trunk.

The figure hereafter gives an example where slaves 2 and 3 which use a 4-Wire interface can operate with the Master and the slave 1 which use a 2-Wire interface.

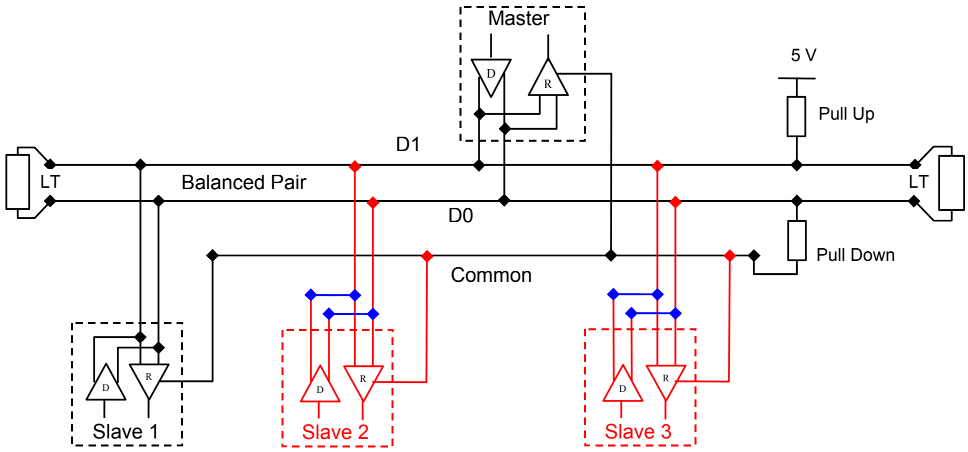


Figure 23 : Connecting devices with 4-Wire interface to a 2-Wire cabling system

3.3.4 RS232-MODBUS Definition

Some devices may implement an RS232-Interface between a DCE and a DTE.

Optional RS232-MODBUS Circuits Definition

Signal	For DCE	Required on DCE (1)	Required on DTE (1)	Description
Common	--	**X**	**X**	**Signal Common**
CTS	In			Clear to Send
DCD	--			Data Carrier Detected (from DCE to DTE)
DSR	In			Data Set Ready
DTR	Out			Data Terminal Ready
RTS	Out			Request to Send
RXD	In	**X**	**X**	**Received Data**
TXD	Out	**X**	**X**	**Transmitted Data**

Notes :

- "X" marked signals are required only if an RS232-MODBUS option is implemented.
- Signals are in accordance with EIA/ TIA-232.
- Each TXD must be wired with RXD of the other device ;
- RTS may be wired with CTS of the other device,
- DTR may be wired with DSR of the other device.
- Optional electrical interfaces may be added, for example :
  - **Power Supply :** 5..24 V D.C.
  - **PMC circuit :** See above (In 2W-MODBUS Circuits Definition) the note about this optional circuit.

3.3.5 RS232-MODBUS requirements

This optional MODBUS on Serial Line system should only be used for short length (typically less than 20m) point to point inter-connection.

Then, the EIA/TIA-232 standard must be respected :

⇒ circuits definition,

⇒ maximum wire capacitance to ground (2500 pF, then 25 m for a 100 pF/m cable).

Please refer to chapter "Cables" for the shield, and for the possibility to use Category 5 Cables.

Documentation of the device must indicate :

⇒ if the device must be considered as a DCE either as a DTE,

⇒ how optional circuits must work if such is the case.

3.4 Multipoint System requirements

For any EIA/ TIA-485 multipoint system, in either 2-wire or 4-wire configuration, the following requirements all apply.

3.4.1 Maximum number of devices without repeater

A figure of **32 devices** is always authorized on any RS485-MODBUS system without repeater.

Depending of :

- all the possible addresses,
- the figure of RS485 Unit Load used by the devices,
- and the line polarization in need be,

A RS485 system may implement a larger number of devices. Some devices allow the implementation of a RS485-MODBUS serial line with more than 32 devices, without repeater.

In this case these MODBUS devices must be documented to say how many of such devices are authorized without repeater.

The use of a **repeater** between two heavy loaded RS485-MODBUS is also possible.

3.4.2 Topology

An RS485-MODBUS configuration without repeater has one trunk cable, along which devices are connected, directly (daisy chaining) or by short derivation cables.

The trunk cable, also named “Bus”, can be long (see hereafter). Its two ends must be connected on Line Terminations.

The use of repeaters between several RS485-MODBUS is also possible.

3.4.3 Length

The end to end length of the **trunk cable** must be limited. The maximum length depends on the baud rate, the cable (Gauge, Capacitance or Characteristic Impedance), the number of loads on the daisy chain, and the network configuration *(2-wire or 4-wire)*.

For a maximum 9600 Baud Rate and AWG26 (or wider) gauge, the maximum length is 1000m. In the specific case shown in the figure 22 (4 Wire cabling used as a 2 Wire cabling system) the maximum length must be divided by two.

The **derivations** must be short, never more than 20m. If a multi-port tap is used with n derivations, each one must respect a maximum length of 40m divided by n.

3.4.4 Grounding Arrangements

The « Common » circuit (Signal and optional Power Supply Common) must be connected directly to protective ground, preferably at **one point only** for the entire bus. Generally this point is to choose on the master device or on its Tap.

3.4.5 Line Termination

A reflection in a transmission line is the result of an impedance discontinuity that a travelling wave sees as it propagates down the line. To minimize the reflections from the end of the RS485-cable it is required to place a Line Termination **near each of the 2 Ends** of the Bus.

It is important that the line be terminated at **both** ends since the propagation is bi-directional, but it is not allowed to place more than 2 LT on one passive D0-D1 balanced pair . Never place any LT on a derivation cable.

Each line termination must be connected between the two conductors of the balanced line : D0 and D1.

Line termination may be a 150 ohms value (0.5 W) resistor.

A serial capacitor (1 nF, 10 V minimum) with a 120 Ohms (0.25 W) resistor is a better choice when a polarization of the pair must be implemented (see here after).

In a 4W-system, each pair must be terminated at each end of the bus.

In an RS232 interconnections, no termination should be wired.

3.4.6 Line Polarization

When there is no data activity on an RS-485 balanced pair, the lines are not driven and, thus susceptible to external noise or interference. To insure that its receiver stays in a constant state, when no data signal is present, some devices need to bias the network.

Each MODBUS device must be documented to say :

- if the device needs a line polarization,
- if the device implements, or can implement, such a line polarization.

If one or several devices need polarization, **one** pair of resistors must be connected on the RS-485 balanced pair :

- a Pull-Up Resistor to a 5V Voltage on D1 circuit,
- a Pull-Down Resistor to the common circuit on D0 circuit.

The value of those resistors must be between 450 Ohms and 650 Ohms. 650 Ohms resistors value may allow a higher number of devices on the serial line bus.

In this case, a polarization of the pair must be implemented **at one location for the whole Serial Bus**. Generally this point is to choose on the master device or on its Tap. Other devices must not implement any polarization.

The maximum number of devices authorized on such a MODBUS Serial Line is reduced by 4 from a MODBUS without polarization.

3.5 Mechanical Interfaces

Screw Terminals may be used for both IDv and ITr connections. All information must be provided to the users about the exact location of each signal, with names in accordance with the previous chapter "Electrical Interface".

If a RJ45 (or a mini-DIN or a D-Shell) **connector** is used on an equipment for a MODBUS mechanical interface, a **shielded female connector** must be chosen. Then the cable-end must have a shielded male connector.

3.5.1 Connectors pin-out for 2W-MODBUS

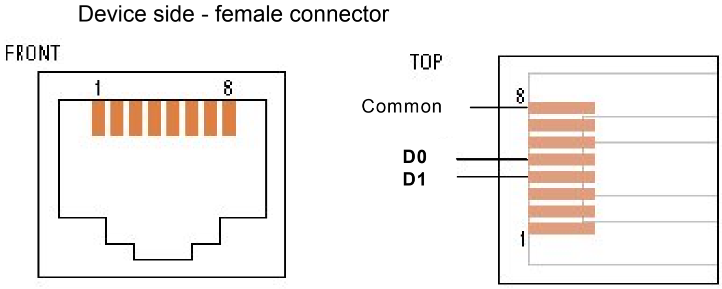


Figure 24: 2W- MODBUS on RJ45 connector (required pin-out)

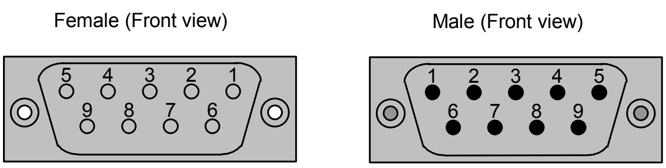


Figure 25: D-shell 9-pin connector

Screw type connectors can also be used.

If an RJ45 or a 9-pin D-shell connector is used for a standard MODBUS device, the pinouts hereafter must be respected for every implemented circuit.

2W-MODBUS RJ45 and 9-pin D-shell Pinouts

Pin on RJ45	Pin on D9-shell	Level of requirement	IDv Circuit	ITr Circuit	EIA/TIA-485 name	Description for IDv
3	3	optional	PMC	--	--	Port Mode Control
4	5	**required**	D1	D1	B/B'	**Transceiver terminal 1, V1 Voltage** (V1 > V0 for binary 1 [OFF] state)
5	9	**required**	D0	D0	A/A'	**Transceiver terminal 0, V0 Voltage** (V0 > V1 for binary 0 [ON] state)
7	2	recommended	VP	--	--	Positive 5...24 V D.C. Power Supply
8	1	**required**	Common	Common	C/C'	**Signal and Power Supply Common**

3.5.2 Connectors pin-out for optional 4W-MODBUS

Device side - female connector

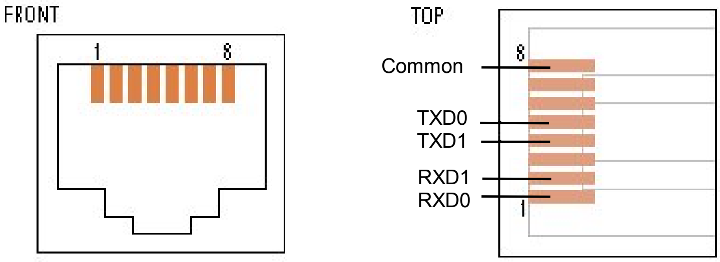


Figure 26: 4W- MODBUS on RJ45 connector (required pin-out)

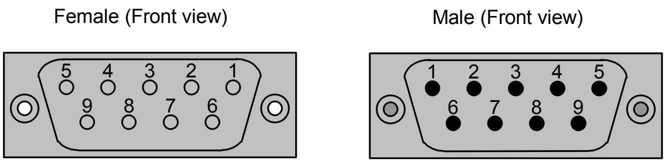


Figure 27: D-shell 9-pin connector

Screw type connectors can also be used.

If an RJ45 or a 9-pin D-shell connector is used for a 4W-MODBUS device, the pinouts hereafter must be respected for every implemented circuit.

Optional 4W-MODBUS RJ45 and 9-pin D-shell Pinouts

Pin on RJ45	Pin on D9-shell	Level of requirement	IDv Signal	ITr Signal	EIA/TIA-485 name	Description for IDv
1	8	**required**	RXD0	RXD0	A'	**Receiver terminal 0, Va' Voltage** (Va' > Vb' for binary 0 [ON] state)
2	4	**required**	RXD1	RXD1	B'	**Receiver terminal 1, Vb' Voltage** (Vb' > Va' for binary 1 [OFF] state)
3	3	optional	PMC	--	--	Port Mode Control
4	5	**required**	TXD1	TXD1	B	**Generator terminal 1, Vb Voltage** (Vb > Va for binary 1 [OFF] state)
5	9	**required**	TXD0	TXD0	A	**Generator terminal 0, Va Voltage** (Va > Vb for binary 0 [ON] state)
7	2	recommended	VP	--	--	Positive 5...24 V DC Power Supply
8	1	**required**	Common	Common	C/C'	**Signal and Power Supply Common**

Note : When both 2 and 4-Wire configurations are implemented on the same port, the **4W notations must be used.**

3.5.3 RJ45 and 9-pin D-shell Pinouts for optional RS232-MODBUS

If an RJ45 or a 9-pin D-shell connector is used for a RS232-MODBUS device, the pinouts hereafter <u>must</u> be respected for every implemented circuit.

DCE **Underlined** pins can be output			Circuit			DTE **Underlined** pins can be output		
Pin on RJ45	**Pin on D9-shell**	**Level of requirement**	**Name**	**Description**	**RS232 Source**	**Level of requirement**	**Pin on RJ45**	**Pin on D9-shell**
<u>1</u>	**<u>2</u>**	**required**	**TXD**	**Transmitted Data**	**DTE**	**required**	**<u>2</u>**	**<u>3</u>**
2	**3**	**required**	**RXD**	**Received Data**	**DCE**	**required**	**1**	**2**
3	7	optional	CTS	Clear to Send	DCE	optional	6	8
<u>6</u>	**<u>8</u>**	optional	RTS	Request to Send	DTE	optional	**<u>3</u>**	**<u>7</u>**
8	**5**	**required**	**Common**	**Signal Common**	**--**	**required**	**8**	**5**

Important Note : Some DCE Pinouts are crossed with DTE Pinouts with the same name :

A directly pin to pin wired cable (without any crossing) <u>must</u> be used between one DTE (a PC for example) and a DCE (a PLC for example).

3.6 Cables

A MODBUS over Serial Line Cable **must be shielded**. At one end of each cable its shield must be connected to protective ground. If a connector is used at this end, the shell of the connector is connected to the shield of the cable.

An RS485-MODBUS must use **a balanced pair** (for D0-D1) **and a third wire** (for the Common). In addition to that a second balanced pair must be used in a 4W-MODBUS system (for RXD0-RXD1).

If a connectorized 4 pairs Category 5 Cable is used, please remember to the user in the User Guides :

"Connection of a crossed cable in a 2-wire MODBUS system may cause damages".

To minimize errors in cabling, a Color Code is recommended for the wires in the RS485-MODBUS Cables :

	Signal Names	Recommended Color
	D1-*TXD1*	**yellow**
	D0-*TXD0*	**brown**
	Common	**grey**
4W (Optional)	*RXD0*	*white*
4W (Optional)	*RXD1*	*blue*

Figure 28: Color code for RS485-MODBUS wires

Note : Category 5 Cables use other colors.

For RS485-MODBUS, Wire Gauge must be chosen sufficiently wide to permit the maximum length (1000 m). AWG 24 is always sufficient for the MODBUS Data.

Category 5 cables may operate for RS485-MODBUS, to a maximum length of 600m.

For the balanced pairs used in an RS485-system, a **Characteristic Impedance** with a value higher than 100 Ohms may be preferred, especially for 19200 and higher baud rates.

3.7 Visual Diagnosis

For a visual diagnosis, communication status and device status must be indicated by LEDs :

LED	Level of requirement	State	Recommended colour
Communication	**required**	Switched ON during frame reception or sending. (2 LEDs for frame reception and frame sending, or 1 LED for both purposes.)	Yellow
Error	**recommended**	Switched ON : internal fault Flashing : Other faults (Communication fault or configuration error)	Red
Device status	**optional**	Switched ON : device powered	Green

4 Installation and Documentation

4.1 Installation

Product vendor should pay attention to give to the user of a MODBUS System or MODBUS Devices all useful information to prevent them **from any error in cabling** or bad utilization of cabling accessories :

- Some other Fieldbuses, CANopen for example, use the same connector types (D-shell, RJ45…) .
- Studies are conducted on Ethernet, with power supply on the same Balanced Pairs Cable.
- Some Products use for I/O circuits the same connector types (D-shell, RJ45…).

On these connectors, for the most part, **no foolproofing** is **available** (polarizing notch or other implementation) .

4.2 User Guide

The User Guide of any MODBUS Device or Cabling System Component must include in a non exhaustive manner one or two types of information:

4.2.1 For any MODBUS Product :

The following information should be documented :

- All the implemented requests.
- The operating modes.
- The visual diagnostics.
- The reachable registers and supported function codes.
- Installation rules.

- The required information in the following sections should also be documented :

⇒ "Two-Wire MODBUS Definition" (to mention the Required Circuits) ;
⇒ "Optional Four-Wire MODBUS Definition" (to mention the Required Circuits) ;
⇒ "Line Polarization" (to mention a possible Need or an Implementation) ;
⇒ "Cables" (special care of crossed cables).

- **A specific indication** relating to the devices addresses, is to be written in the form of an important warning :

"*It is of great importance to ensure at the time of the procedure of devices addressing, that there is not two devices with the same address. In such a case, an abnormal behavior of the whole serial bus can occur, the Master being then in the impossibility to communicate with all present slaves on the bus.*"

- A "**Getting Started**" chapter is highly recommended, with the documented description of a typical application example, for an easy start.

4.2.2 For a MODBUS Product with implemented Options :

The different optional parameters must be clearly detailed :

⇒ Optional serial Transmission mode ;
⇒ Optional Parity Checking ;
⇒ Optional Baud Rates ;
⇒ Optional Circuit(s) : Power Supply, Port Configuration ;
⇒ Optional Interface(s) ;
⇒ Maximum number of devices (without repeater) if greater than 32.

5 Implementation Classes

Each device on a MODBUS Serial Line must respect all the **mandatory** requirements of a same implementation class.

The following parameters are used to classify the MODBUS Serial Line devices :

- Addressing
- Broadcasting
- Transmission mode
- Baud rate
- Character format
- Electrical interface parameter

Two implementation classes are proposed, the Basic and the Regular classes.

The regular class must provide configuration capabilities.

	BASIC		REGULAR	Default value
Addressing	Slave : configurable address from 1 to 247	Master : to be able to address a slave from address 1 to 247	Same as Basic	-
Broadcast	Yes		Yes	-
Baud Rate	9600 (19200 is also recommended)		9600, 19200 + additional configurable baud rates	19200 (if implemented, else 9600)
Parity	EVEN		EVEN + possibility to configure NO and ODD parity	EVEN
Mode	RTU		RTU + ASCII	RTU
Electrical Interface	RS485 2W-cabling or RS232		RS485 2W-cabling (and 4W-cabling as an additional option) or RS232	RS485 2W-cabling
Connector Type	RJ 45 (recommended)			-

6 Appendix

6.1 Appendix A - Management of Serial Line Diagnostic Counters

6.1.1 General description

MODBUS Serial Line defines a list of diagnostic counters to allow performance and error management.

These counters are accessible using the MODBUS application protocol and its Diagnostic function (function code 08).

Each counter can be get by a sub-function code bound to the counter number. All counters can be cleared using the sub-function code 0x0A.

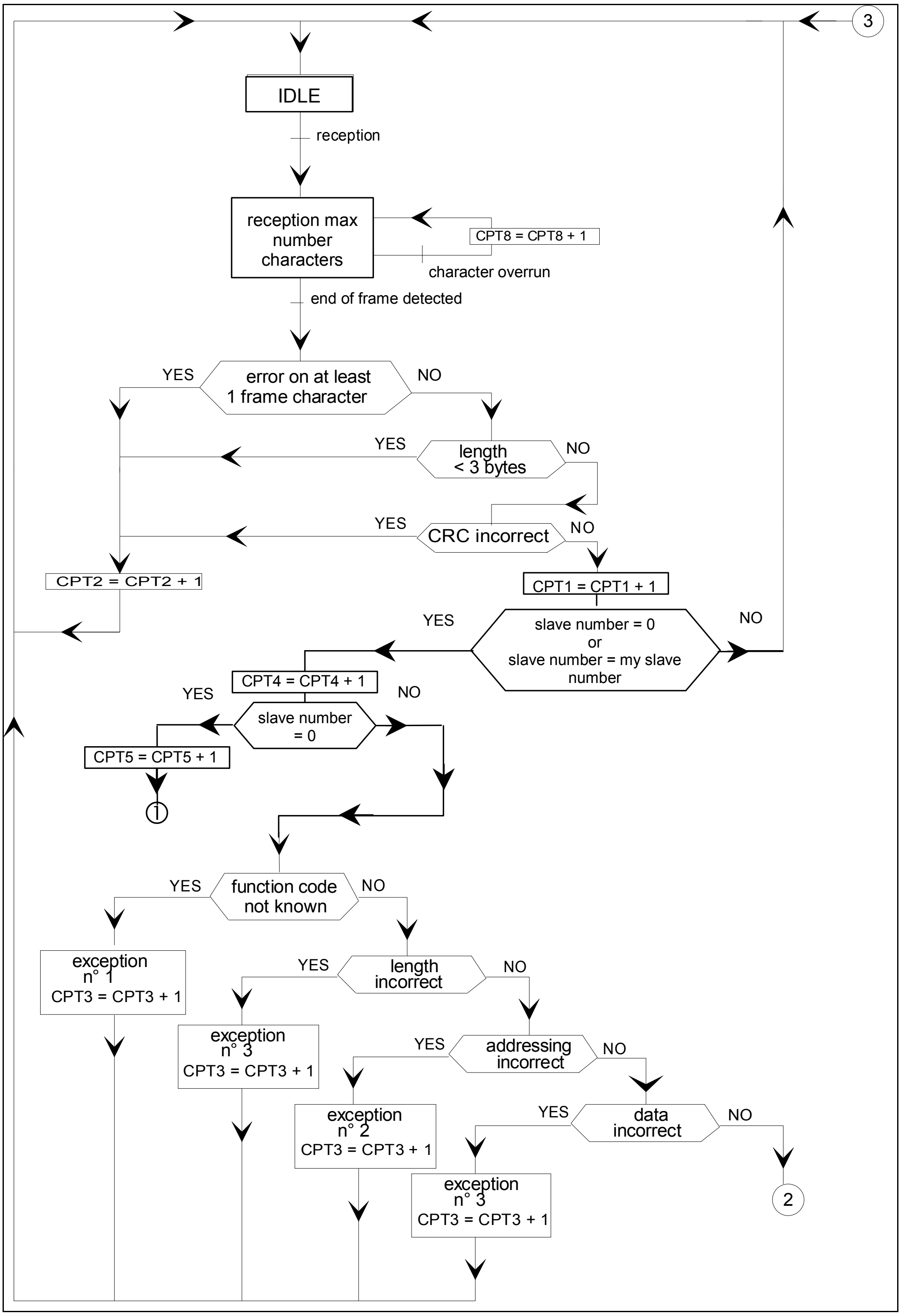
The format of the Diagnostic function is described in the MODBUS application protocol specification.

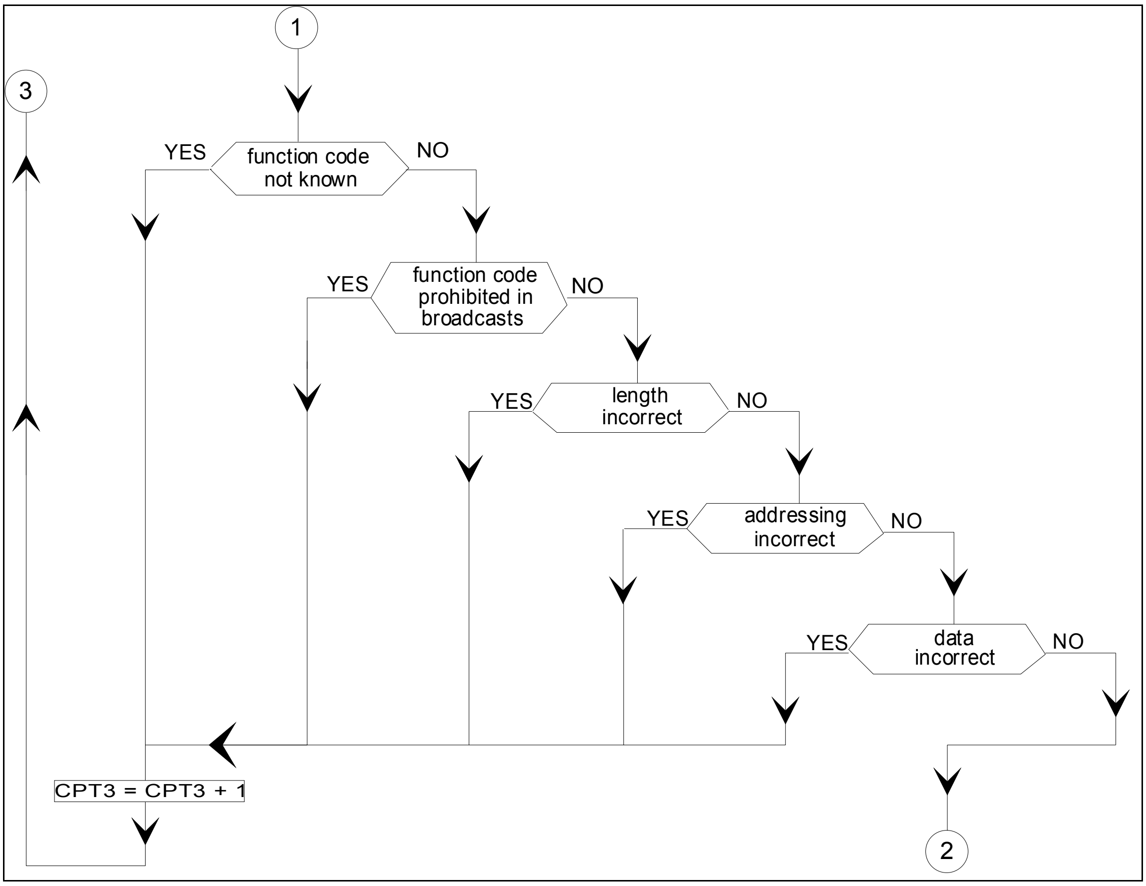
Herein is the list of diagnostics and associated sub-function codes supported by a serial line device.

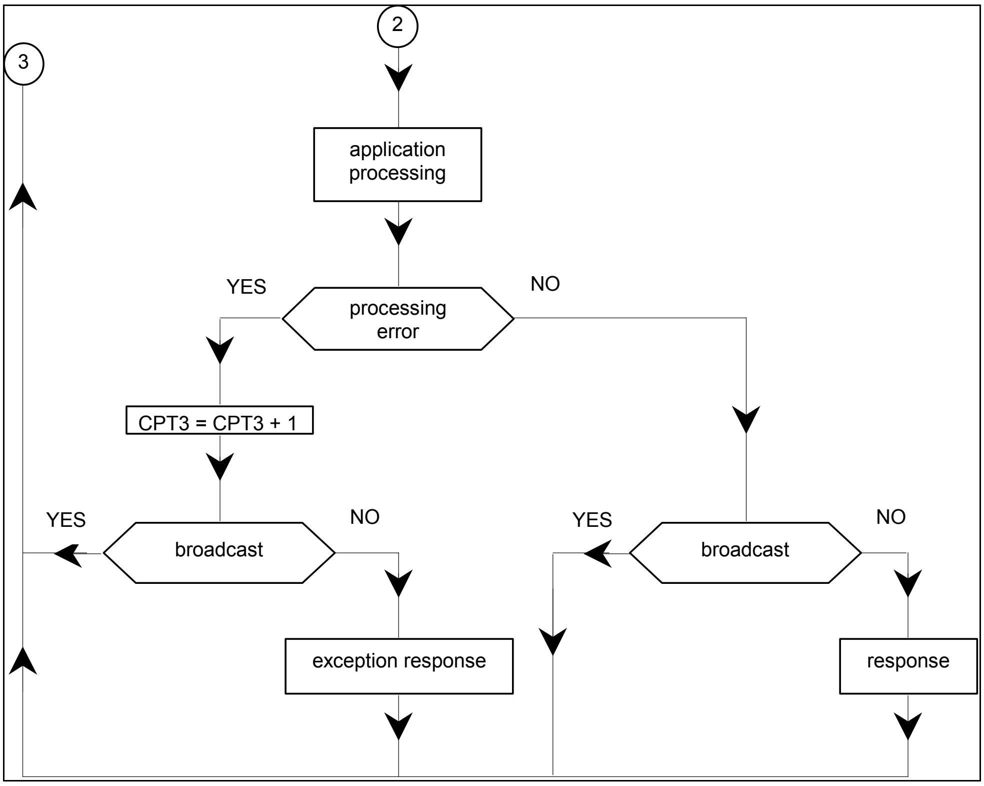
Sub-function code	Counter number	Counters Name	Comments (for diagram below)
Hex	Dec		
0x0B	1	Return Bus Message Count	Quantity of messages that the remote device has detected on the communications system since its last restart, clear counters operation, or power–up. Messages with bad CRC are not taken into account.
0x0C	2	Return Bus Communication Error Count	Quantity of CRC errors encountered by the remote device since its last restart, clear counters operation, or power–up. In case of an error detected on the character level, (overrun, parity error), or in case of a message length < 3 bytes, the receiving device is not able to calculate the CRC. In such cases, this counter is also incremented.
0x0D	3	Return Slave Exception Error Count	Quantity of MODBUS exception error detected by the remote device since its last restart, clear counters operation, or power–up. It comprises also the error detected in broadcast messages even if an exception message is not returned in this case. Exception errors are described and listed in "MODBUS Application Protocol Specification" document.
0xOE	4	Return Slave Message Count	Quantity of messages addressed to the remote device, including broadcast messages, that the remote device has processed since its last restart, clear counters operation, or power–up.
0x0F	5	Return Slave No Response Count	Quantity of messages received by the remote device for which it returned no response (neither a normal response nor an exception response), since its last restart, clear counters operation, or power–up. Then, this counter counts the number of broadcast messages it has received.
0x10	6	Return Slave NAK Count	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 "MODBUS Application Protocol Specification" document.
0x11	7	Return Slave Busy Count	Quantity of messages addressed to the remote device for which it returned a Slave Device Busy exception response, since its last restart, clear counters operation, or power–up. Exception responses are described and listed in "MODBUS Application Protocol Specification" document
0x12	8	Return Bus Character Overrun Count	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.

6.1.2 Counters Management Diagram

The following diagrams describe when each previous counters must be incremented.




1
3
YES
function code not known
NO
YES
function code prohibited in broadcasts
NO
YES
length incorrect
NO
YES
addressing incorrect
NO
YES
data incorrect
NO
CPT3 = CPT3 + 1
2


2
3
application processing
YES
processing error
NO
CPT3 = CPT3 + 1
YES
broadcast
NO
YES
broadcast
NO
exception response
response

6.2 Appendix B - LRC/CRC Generation

6.2.1 LRC Generation

The Longitudinal Redundancy Checking (LRC) field is one byte, containing an 8–bit binary value. The LRC value is calculated by the transmitting device, which appends the LRC to the message. The device that receives recalculates an LRC during receipt of the message, and compares the calculated value to the actual value it received in the LRC field. If the two values are not equal, an error results.

The LRC is calculated by adding together successive 8–bit bytes in the message, discarding any carries, and then two's complementing the result. The LRC is an 8–bit field, therefore each new addition of a character that would result in a value higher than 255 decimal simply 'rolls over' the field's value through zero. Because there is no ninth bit, the carry is discarded automatically.

A procedure for generating an LRC is:

1. Add all bytes in the message, excluding the starting 'colon' and ending CRLF. Add them into an 8–bit field, so that carries will be discarded.
2. Subtract the final field value from FF hex (all 1's), to produce the ones–complement.
3. Add 1 to produce the twos–complement.

Placing the LRC into the Message

When the 8–bit LRC (2 ASCII characters) is transmitted in the message, the high–order character will be transmitted first, followed by the low–order character. For example, if the LRC value is 61 hex (0110 0001):

Colon	Addr	Func	Data Count	Data	Data	Data	Data	LRC Hi	LRC Lo	CR	LF
								"6" 0x36	"1" 0x31		

Figure 29: LRC Character Sequence

Example: an example of a C language function performing LRC generation is shown below.

The function takes two arguments:

unsigned char *auchMsg; A pointer to the message buffer containing binary data to be used for generating the LRC,

unsigned short usDataLen; The quantity of bytes in the message buffer.

LRC Generation Function

```
static unsigned char LRC(auchMsg, usDataLen)     /* the function returns the LRC as a type unsigned char */
unsigned char *auchMsg ;                         /* message to calculate LRC upon */
unsigned short usDataLen ;                       /* quantity of bytes in message */
{
  unsigned char uchLRC = 0 ;                     /* LRC char initialized */
  while (usDataLen—)                             /* pass through message buffer */
  uchLRC += *auchMsg++ ;                         /* add buffer byte without carry */
  return ((unsigned char)(–((char)uchLRC))) ;    /* return twos complement */
}
```

6.2.2 CRC Generation

The Cyclical Redundancy Checking (CRC) field is two bytes, containing a 16–bit binary value. The CRC value is calculated by the transmitting device, which appends the CRC to the message. The device that receives recalculates a CRC during receipt of the message, and compares the calculated value to the actual value it received in the CRC field. If the two values are not equal, an error results.

The CRC is started by first preloading a 16–bit register to all 1's. Then a process begins of applying successive 8–bit bytes of the message to the current contents of the register. Only the eight bits of data in each character are used for generating the CRC. Start and stop bits and the parity bit, do not apply to the CRC.

During generation of the CRC, each 8–bit character is exclusive ORed with the register contents. Then the result is shifted in the direction of the least significant bit (LSB), with a zero filled into the most significant bit (MSB) position. The LSB is extracted and examined. If the LSB was a 1, the register is then exclusive ORed with a preset, fixed value. If the LSB was a 0, no exclusive OR takes place.

This process is repeated until eight shifts have been performed. After the last (eighth) shift, the next 8–bit character is exclusive ORed with the register's current value, and the process repeats for eight more shifts as described above. The final content of the register, after all the characters of the message have been applied, is the CRC value.

A procedure for generating a CRC is:

1. Load a 16–bit register with FFFF hex (all 1's). Call this the CRC register.
2. Exclusive OR the first 8–bit byte of the message with the low–order byte of the 16–bit CRC register, putting the result in the CRC register.
3. Shift the CRC register one bit to the right (toward the LSB), zero–filling the MSB. Extract and examine the LSB.
4. (If the LSB was 0): Repeat Step 3 (another shift).
   (If the LSB was 1): Exclusive OR the CRC register with the polynomial value 0xA001 (1010 0000 0000 0001).
5. Repeat Steps 3 and 4 until 8 shifts have been performed. When this is done, a complete 8–bit byte will have been processed.
6. Repeat Steps 2 through 5 for the next 8–bit byte of the message. Continue doing this until all bytes have been processed.
7. The final content of the CRC register is the CRC value.
8. When the CRC is placed into the message, its upper and lower bytes must be swapped as described below.

Placing the CRC into the Message

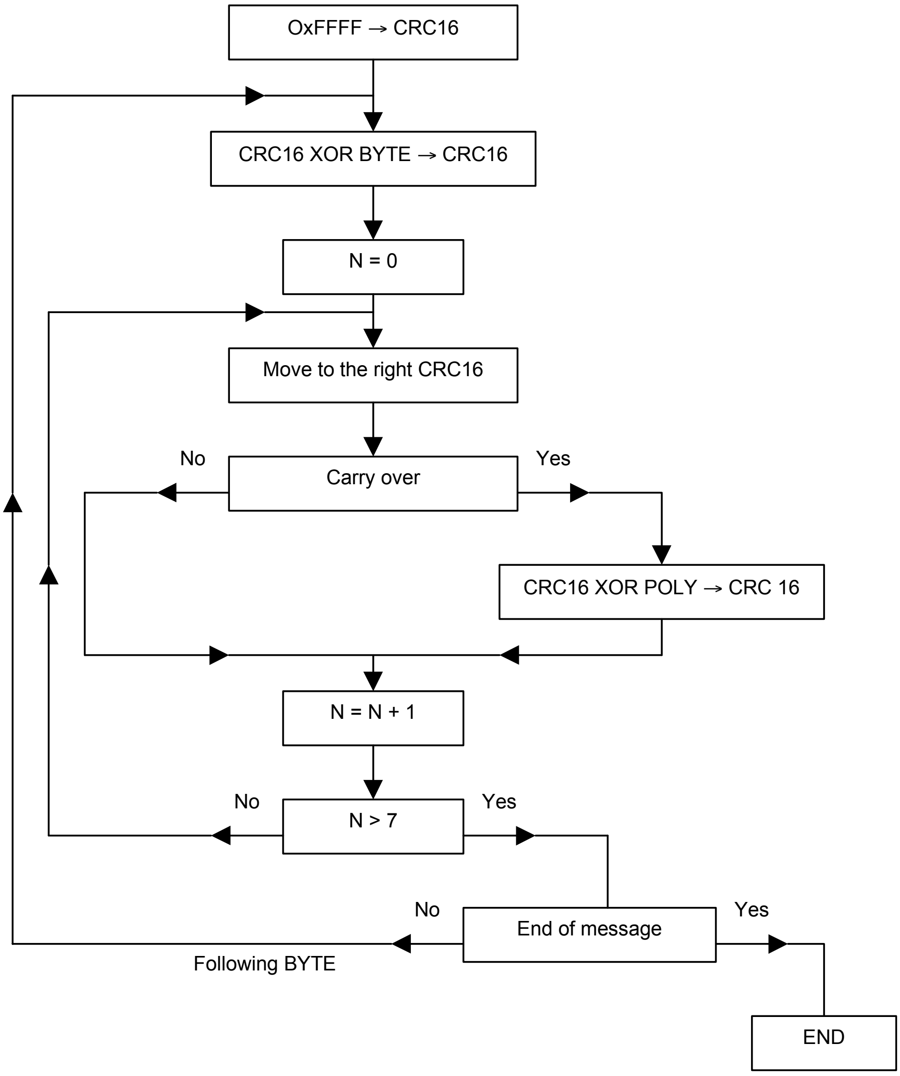
When the 16–bit CRC (two 8–bit bytes) is transmitted in the message, the low-order byte will be transmitted first, followed by the high-order byte.

For example, if the CRC value is 1241 hex (0001 0010 0100 0001):

Addr	Func	Data Count	Data	Data	Data	Data	CRC Lo	CRC Hi
							0x41	0x12

Figure 30: CRC Byte Sequence

Calculation algorithm of the CRC 16



XOR = exclusive or

N = number of information bits

POLY = calculation polynomial of the CRC 16 = 1010 0000 0000 0001

(Generating polynomial = $1 + x_2 + x_{15} + x_{16}$)

In the CRC 16, the 1st byte transmitted is the least significant one.

Example of CRC calculation (frame 02 07)

CRC register initialization		1111	1111	1111	1111
XOR 1st character		0000	0000	0000	0010
		1111	1111	1111	1101
	Move 1	0111	1111	1111	1110|1
Flag to 1, XOR polynomial		1010	0000	0000	0001
		1101	1111	1111	1111
	Move 2	0110	1111	1111	1111|1
Flag to 1, XOR polynomial		1010	0000	0000	0001
		1100	1111	1111	1110
	Move 3	0110	0111	1111	1111|0
	Move 4	0011	0011	1111	1111|1
		1010	0000	0000	0001
		1001	0011	1111	1110
	Move 5	0100	1001	1111	1111|0
	Move 6	0010	0100	1111	1111|1
		1010	0000	0000	0001
		1000	0100	1111	1110
	Move 7	0100	0010	0111	1111|0
	Move 8	0010	0001	0011	1111|1
		1010	0000	0000	0001
		1000	0001	0011	1110
XOR 2nd character		0000	0000	0000	0111
		1000	0001	0011	1001
	Move 1	0100	0000	1001	1100|1
		1010	0000	0000	0001
		1110	0000	1001	1101
	Move 2	0111	0000	0100	1110|1
		1010	0000	0000	0001
		1101	0000	0100	1111
	Move 3	0110	1000	0010	0111|1
		1010	0000	0000	0001
		1100	1000	0010	0110
	Move 4	0110	0100	0001	0011|0
	Move 5	0011	0010	0000	1001|1
		1010	0000	0000	0001
		1001	0010	0000	1000
	Move 6	0100	1001	0000	0100|0
	Move 7	0010	0100	1000	0010|0
	Move 8	0001	0010	0100	0001|0
		Most significant		least significant	

The CRC 16 of the frame is then: 4112

Example

An example of a C language function performing CRC generation is shown on the following pages. All of the possible CRC values are preloaded into two arrays, which are simply indexed as the function increments through the message buffer. One array contains all of the 256 possible CRC values for the high byte of the 16–bit CRC field, and the other array contains all of the values for the low byte.

Indexing the CRC in this way provides faster execution than would be achieved by calculating a new CRC value with each new character from the message buffer.

Note: This function performs the swapping of the high/low CRC bytes internally. The bytes are already swapped in the CRC value that is returned from the function.

Therefore the CRC value returned from the function can be directly placed into the message for transmission.

The function takes two arguments:

unsigned char *puchMsg;	A pointer to the message buffer containing binary data to be used for generating the CRC
unsigned short usDataLen;	The quantity of bytes in the message buffer.

CRC Generation Function

```
unsigned short CRC16 ( puchMsg, usDataLen )          /* The function returns the CRC as a unsigned short type  */
unsigned char *puchMsg ;                             /* message to calculate CRC upon                          */
unsigned short usDataLen ;                           /* quantity of bytes in message                           */
{
        unsigned char uchCRCHi = 0xFF ;              /* high byte of CRC initialized                           */
        unsigned char uchCRCLo = 0xFF ;              /* low byte of CRC initialized                            */
        unsigned uIndex ;                            /* will index into CRC lookup table                       */

        while (usDataLen--)                          /* pass through message buffer                            */
        {
                uIndex = uchCRCLo ^ *puchMsg++ ;     /* calculate the CRC                                      */
                uchCRCLo = uchCRCHi ^ auchCRCHi[uIndex] ;
                uchCRCHi = auchCRCLo[uIndex] ;

        }
        return (uchCRCHi << 8 | uchCRCLo) ;
}
```

High-Order Byte Table

```
/* Table of CRC values for high–order byte */
static unsigned char auchCRCHi[] = {
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
    0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
    0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
    0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81,
    0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
    0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
    0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
    0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
    0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
    0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
    0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
    0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
    0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
    0x40
} ;
```

Low-Order Byte Table

```
/* Table of CRC values for low–order byte */
static char auchCRCLo[] = {
    0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06, 0x07, 0xC7, 0x05, 0xC5, 0xC4,
    0x04, 0xCC, 0x0C, 0x0D, 0xCD, 0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09,
    0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A, 0x1E, 0xDE, 0xDF, 0x1F, 0xDD,
    0x1D, 0x1C, 0xDC, 0x14, 0xD4, 0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13, 0xD3,
    0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3, 0xF2, 0x32, 0x36, 0xF6, 0xF7,
    0x37, 0xF5, 0x35, 0x34, 0xF4, 0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A,
    0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29, 0xEB, 0x2B, 0x2A, 0xEA, 0xEE,
    0x2E, 0x2F, 0xEF, 0x2D, 0xED, 0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26,
    0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60, 0x61, 0xA1, 0x63, 0xA3, 0xA2,
    0x62, 0x66, 0xA6, 0xA7, 0x67, 0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC, 0xAD, 0x6D, 0xAF, 0x6F,
    0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68, 0x78, 0xB8, 0xB9, 0x79, 0xBB,
    0x7B, 0x7A, 0xBA, 0xBE, 0x7E, 0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C, 0xB4, 0x74, 0x75, 0xB5,
    0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71, 0x70, 0xB0, 0x50, 0x90, 0x91,
    0x51, 0x93, 0x53, 0x52, 0x92, 0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54, 0x9C, 0x5C,
    0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B, 0x99, 0x59, 0x58, 0x98, 0x88,
    0x48, 0x49, 0x89, 0x4B, 0x8B, 0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C, 0x8C,
    0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42, 0x43, 0x83, 0x41, 0x81, 0x80,
    0x40
};
```

6.3 Appendix E - References

ANSI/ TIA/ EIA-232-F-1997	**Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange.**
ANSI/ TIA/ EIA-485-A-1998	**Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems.**
AWG	**"American Wire Gauge" is a standard method denoting wire diameter, it is used in the USA and in other countries; increasing gauge numbers give decreasing wire parameters. See for example D.G. Fink and H.W. Beaty, Standard Handbook for Electrical Engineers, 13th Edition, McGraw-Hill, 1993.**
MODBUS.org	**MODBUS application protocol specification**