Project Acronym: MEDIS

Project Title: A Methodology for the Formation of Highly Qualified Engineers at Masters Level in the Design and Development of Advanced Industrial Informatics Systems

Contract Number: 544490-TEMPUS-1-2013-1-ES-TEMPUS-JPCR

Starting date: 01/12/2013 **Ending date:** 30/11/2016

Deliverable Number: 2.4.2

Title of the Deliverable: AIISM teaching resources - Industrial Networks and Fieldbuses – Modbus/TCP

Task/WP related to the Deliverable: Development of the AIISM teaching resources - Industrial Networks and Fieldbuses

Type (Internal or Restricted or Public): Internal

Author(s): Paulo Portugal

Partner(s) Contributing: FEUP

Contractual Date of Delivery to the CEC: 30/09/2014

Actual Date of Delivery to the CEC: 30/09/2014

Project Co-ordinator

Company name :	Universitat Politècnica de València (UPV)
Name of representative :	Houcine Hassan
Address :	Camino de Vera, s/n. 46022 - Valencia (Spain)
Phone number :	+34 96 387 7578
Fax number :	+34 96 387 7579
E-mail :	husein@disca.upv.es
Project WEB site address :	http://medis.upv.es/

Context

WP 2	Design of the AIISM-PBL methodology
WPLeader	Universitat Politècnica deValència (UPV)
Task 2.4	Development of the AIISM teaching resources - – Industrial Networks and Fieldbuses – Modbus/TCP
Task Leader	UP
Dependencies	UPV, MDU, TUSofia, USTUTT, UP

Author(s)	Paulo Portugal
Contributor(s)	
Reviewers	Mário de Sousa

History

Version	**Date**	**Author**	**Comments**
1.0	01/04/2014	Paulo Portugal	Lecture
2.0	15/04/2014	Paulo Portugal	Seminar
3.0	01/05/2014	Paulo Portugal	Lab
4.0	15/05/2014	Paulo Portugal	Mini-project
5.0	19/09/2015	Paulo Portugal	Revision

Table of Contents

1 Executive summary

WP 2.4 details the learning materials of the Advanced Industrial Informatics Specialization Modules (AIISM) related to the Industrial Networks and Fieldbuses.

The contents of this package follows the guidelines presented in the Partner's documentation of the WP 1 (Industrial Networks and Fieldbuses)

1. The PBL methodology was presented in WP 1.1
2. The list of the module's chapters and the temporal scheduling in WP 1.2
3. The required human and material resources in WP 1.3
4. The evaluation in WP 1.4

During the development of this WP a separate document has been created for each of the chapters of the Industrial Networks and Fieldbuses Module (list of chapters in WP1.1). This document is for the second chapter – Modbus/TCP.

In this document, section 2 defines the lecture, sections 3 and 4 describe the laboratory work for weeks 3 and 4, sections 5 and 6 explain the seminar topics for weeks 3 and 4, and sections 7 and 8 define the requirements to fulfill for the mini-project during weeks 3 and 4. Section 9 lists the bibliography and the references.

2 Lecture

This lecture aims to present the Modbus protocol in general and its implementation over the TCP/IP stack in particular.

2.1 Context

In 1979, a company called Modicon took the initiative to develop a new communication protocol for their Programmable Logic Controllers (PLCs), which they called *Modbus*. Later, they made the decision to make available to the whole community its specification, as well as their use without paying royalties. The simplicity of the protocol combined with the previous decisions encouraged many companies over the last four decades to include Modbus in their products, thus leading to a wide industry acceptance as a *de facto* communication standard. Since 2004, Modbus specification and future evolution is controlled by the *Modbus Organization*, a community of independent users, entities and equipment suppliers which are associated to the use of this protocol [1].
Currently, Modbus can be found in a large variety of devices, ranging from small embedded systems to high level control and supervision applications like MES (Manufacturing Execution Systems). Moreover, their inclusion in new products is facilitated by the fact that there are many implementations, using a large variety of programming languages, whose source code is freely open.

2.2 Modbus

Data Model

The Modbus protocol [2] assumes that communications are established between *devices*. A device can be any type equipment (e.g. embedded system, PLC, etc.) equipped with a suitable communication interface (e.g. serial port, Ethernet). The protocol doesn't pose any restrictions regarding the type or complexity of the device. By assuming this the protocol abstracts from real behavior of the device (i.e. how it interacts with its environment) and focuses on how device' internal data is organized and can be accessed.

Communication between devices is performed according to a *Client-Server* model (Figure 1). The client makes requests to server, which replies with the required data. This process is called a *transaction*. A client can make requests to several servers, while a server can reply to requests coming from different clients.

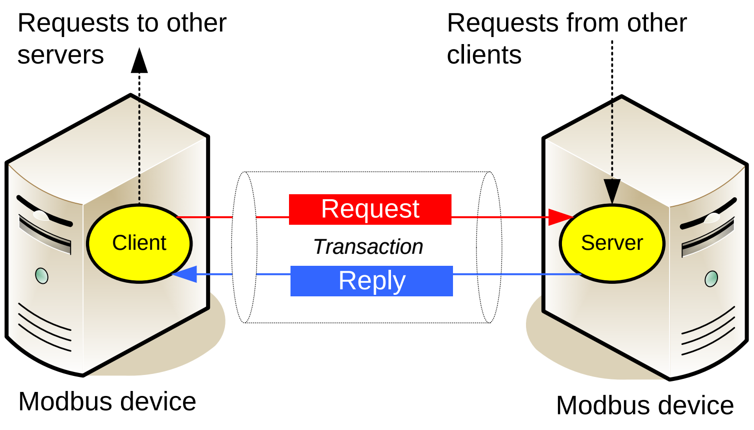
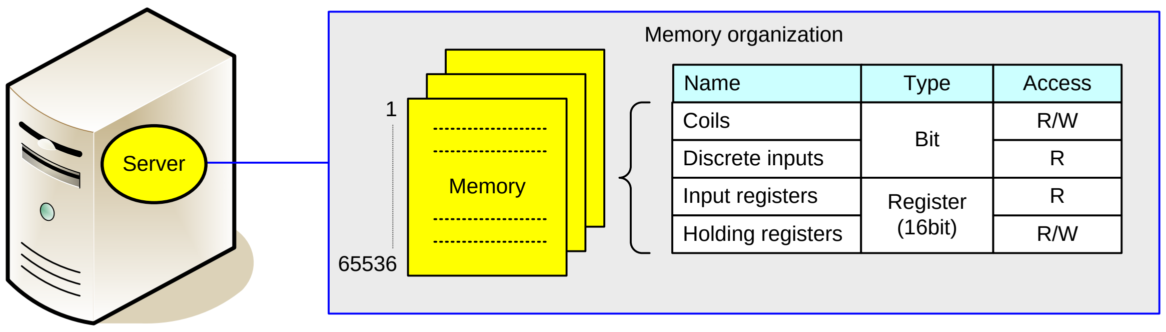


Figure 1 - Interaction model.

A Modbus server is organized as a block of memory, which can be written or read. There are four distinct memory areas (Figure 2). Two of them are organized as 16 bit registers, whereas the other two are composed of arrays of single bits. Likewise, two of the memory areas provide read and write access permissions, while the other two may only be read from. Every data item (bit or register) has an address in the range from 1 to 65536.



Name	Type	Access
Coils	Bit	R/W
Discrete inputs		R
Input registers	Register (16bit)	R
Holding registers		R/W

Figure 2 – Memory areas.

Modbus doesn't impose any semantics to these memory areas, being the device manufacturer the responsible for its definition. That is, the protocol doesn't define what will happen on the server when a specific value is written is a specific memory area. Neither defines the meaning of the data that is read from a specific memory area. The way memory areas are organized is also under the responsibility of manufacturer. These areas can be overlapped or splitted over multiple addresses. Nevertheless and for historical reasons, is common to associate *Coils* and *Discrete inputs* to digital outputs and inputs, respectively, whereas *Input* and *Holding registers* are associated to analog inputs and outputs, respectively.

Figure 3 shows an example how the memory of a Programmable Logic Controller (PLC) can be organized. The manufacturer decided to map the PLC digital outputs (#1…512) at the *Coils* memory area, while digital inputs (#1K…2K) are at the *Discrete inputs* area. PLC analog inputs (#1…8) and outputs (#1…16) are respectively mapped at the *Input registers* and *Holding registers* areas. The manufacturer also decided to use distinct areas (ie. non overlapped) because they have different meanings (eg. a discrete input has different characteristics from an analog input).

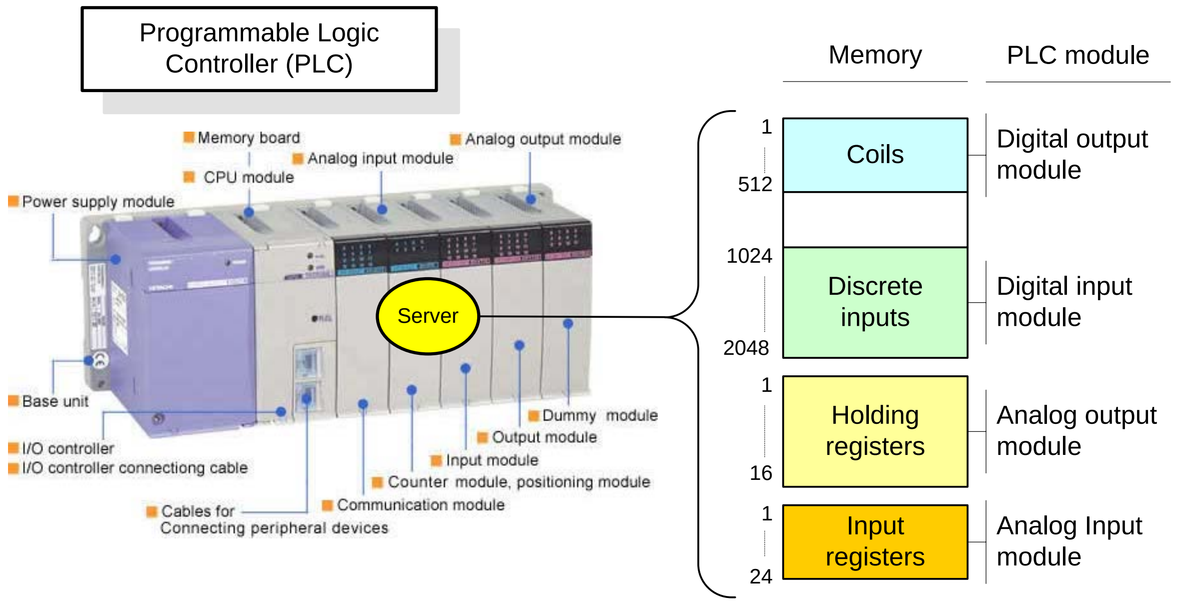


Figure 3 - Example: PLC memory organization.

Figure 4 shows a different example regarding a variable speed controller. In this case the manufacturer decided to use a register at the *Input registers* area to store the motor speed drive (#1). The status of the device can be accessed as an *Input register* (#2), and simultaneously each bit of that register can also be accessed as a *Discrete input* (#1…16).

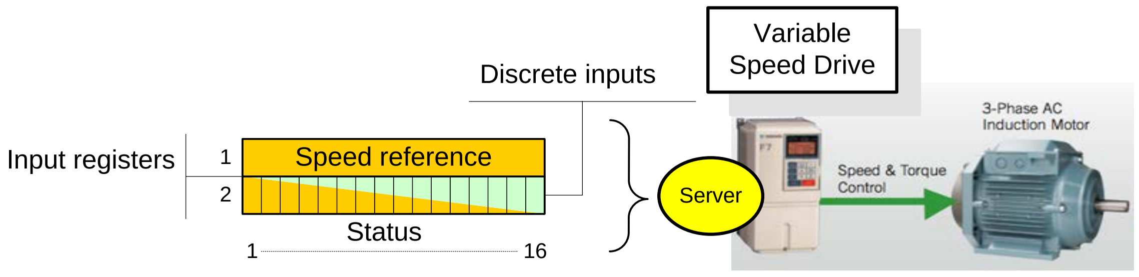


Figure 4 - Example: variable speed drive memory organization.

Protocol Architecture

Modbus is an application layer protocol which internally is structured in two sublayers (Figure 5).

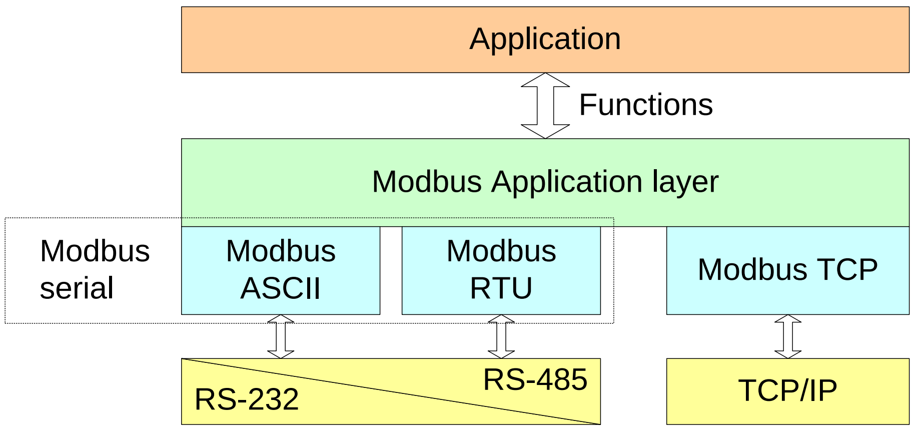


Figure 5 - Organization of Modbus protocols.

The upper layer, called *Modbus Application Layer* [2], defines the communication services that are available to the application, their semantics and how they operate. Modbus calls to these services *functions*. These services are independent of the underlying communication layers.

The lower layer defines how the upper layer data is encapsulated and encoded into frames to be sent over different physical layers. There are three distinct versions of this layer. Two of them, Modbus RTU (*Remote Terminal Unit*) [4] and Modbus ASCII [4] (*American Standard Code for Information Interchange*) were proposed to transmit Modbus data over a serial line using EIA/TIA-232 or EIA/TIA-485 interfaces. The TCP version [3], as the name implies, is used to send Modbus data using TCP/IP (*Transmit Control Protocol/Internet Protocol*) connections.

Modbus also includes the concept of *Modbus gateway device*, which is used to interconnect Modbus devices placed on different networks (Figure 6).

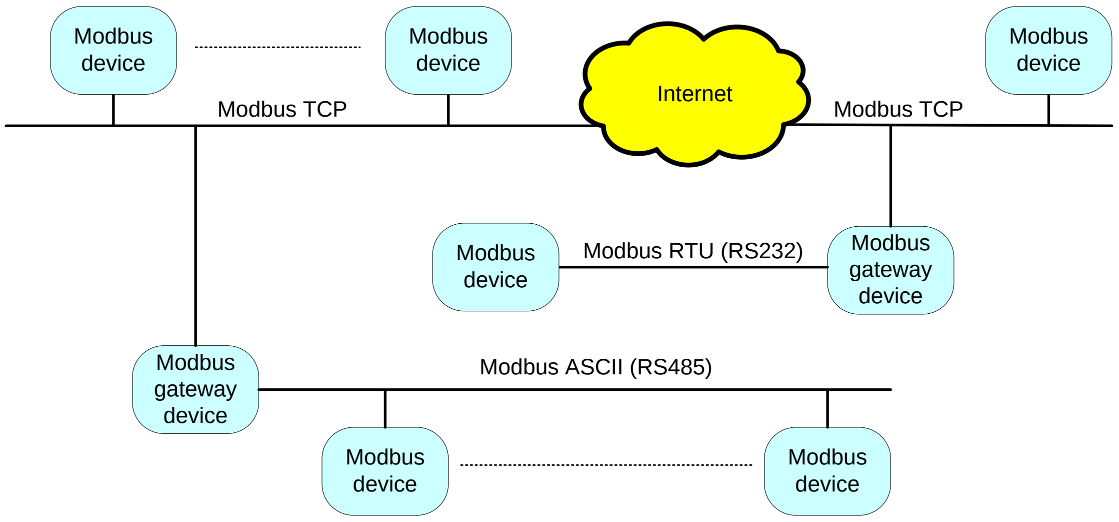


Figure 6 - Example: interconnection of different Modbus networks.

Modbus Application Layer

Modbus application layer [2] defines functions (services) that are used by the client application to invoke services at the server. Each function is identified by a *function code*, an integer number ranging from #1 to #127. Function codes are organized in three categories:

- **Public -** functions defined by Modbus Organization, i.e. included in the Modbus specification.
- **User-defined -** functions which aren't included in the specification and that are defined and implemented by the users.
- **Reserved**: functions that defined by the device manufacturer and which are not available to the users.

Some functions also support additional *sub-function codes* that are used to perform multiple actions.

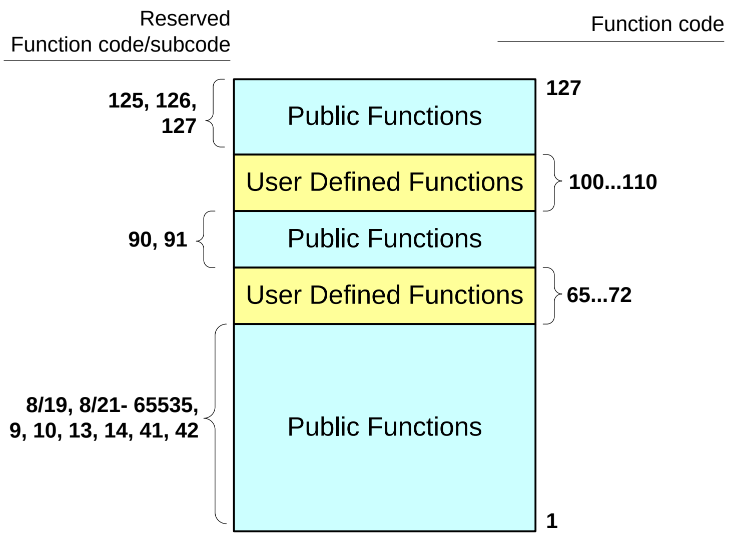


Figure 7 - Function codes.

Public functions are organized in three main groups:

- Data access
- Diagnostic
- Other

Data Access Functions

Modbus defines 13 *Data Access* functions which provide different ways to access to the server memory area (Figure 8). All functions belonging to this group, with the exception of functions #20, #21, #22 and #24, simply request that some data item (bits or registers) be read from or write to a specific memory area. Functions #5 and #6 are used to write to a single data item, while the remaining functions allow the client to read from or write to multiple contiguous data items the same memory area.

Function #22 (*Mask Write Register*) is used to change the value of a register by applying two masks (AND and OR masks) to their current value.

Function #24 (*Read FIFO Queue*) allows the client to read a FIFO queue from the server. This queue is stored as an array at the Holding Register memory area. The first register of the array contains the number of elements of the queue, while the remaining elements of the array contain the values of each element of the queue.

Modbus also includes two functions to access to data structures which are similar to a *file* (#20 *Read File Record* and #21 *Write File Record*). A file is composed by group of contiguous records (up to 10000), where each record is a register (16 bits). Files are stored in a memory area that is independent from the memory areas defined previously. Each file is referenced by its number, ranging from 0 to 65535.

Name	Code	Subcode	Parameters
Read Coils	1		Starting address, Quantity of colis
Write Single Coil	5		Address, Value
Write Multiple Coils	15		Starting address, Quantity of coils, Values
Read Discrete Inputs	2		Starting address, Quantity of inputs
Read Input Registers	4		Starting address, Quantity of registers
Read Holding Registers	3		Starting address, Quantity of registers
Write Single Register	6		Address, Value
Write Multiple Registers	16		Starting address, Quantity of registers, Values
Mask Write Register	22		Address, AND Mask, OR Mask
Read/Write Multiple Registers	23		Read starting address, Quantity to read, Write starting address, Quantity to write, Values
Read FIFO Queue	24		FIFO pointer address
Read File Record	20		Reference type, File number, Record number, Record length
Write File Record	21		Reference type, File number, Record number, Record length, Record data

Figure 8 - Data access functions.

Diagnostic Functions

Diagnostic functions allow a client to obtain diagnostic information from the server (Figure 9) These functions, with exception of function #43 (subcode 14, *Read Device Identification*), are only available on Modbus RTU/ASCII versions.

Name	Code	Subcode	Parameters
Read Exception Status	7		
Diagnostics	8	0-18, 20	Sub function, Data
Get Comm Event Counter	11		
Get Comm Event Log	12		
Report Slave ID	17		
Read Device Identification	43	14	MEI type, Device ID Code, Object ID

Figure 9 - Diagnostic functions

Function #07 (*Read Exception Status*) is used to read information about the exception status of the server.

Function #08 (*Diagnostic*) allows a client to request diagnostic information from the server or to request that a server executes diagnostic routines and to report their result.

Functions #11 (*Get Com Event Counter*) and #12 (*Get Com Event Log*) are closely related. The former allows getting information regarding the event counter. This counter is incremented by the server once for each successful transaction. The latter is used to get a log of events related with this counter.

Functions #17 (*Report Slave ID*) and #43 subcode #14 (*Read Device Identification*) are also related. The former allows getting the run status of the server (i.e. run / stop), a device specific identification, and some additional device specific data. The latter allows reading additional identification information.

Other Functions

Functions belonging to this group are used for tunneling other communication protocols inside a Modbus transaction (Figure 10). It is assumed that the server supports also these protocols. A single function, #43, with two subcodes (13 and 14) is defined for this purpose. Function subcode #13 (*CANOpen General Reference*) is used to access to CANopen services, while subcode #14 (*Encapsulated Interface Transport*) is used to access to generic services of a transport protocol.

Name	**Code**	**Subcode**	**Parameters**
Encapsulated Interface Transport	43	13, 14	MEI type, MEI type specific data
CANOpen General Reference	43	13	MEI type, MEI type specific data

Figure 10 - Other functions.

Interested readers can find details regarding these functions on Modbus Specification document [2].

Device Classes

Modbus devices (i.e. client or servers) don't need to support all Public Access functions. Actually, Modbus groups devices into *conformance classes*, depending on which functions they implement (Figure 11). All devices must implement *Class 0* functions, while *Class 1* and *2* are optional.

- Class 0 includes the minimum useful set of functions.
- Class 1 defines a more complete set of the most commonly used functions.
- Class 2 devices support all data transfer functions.

Class 0
Read Holding Registers
Write Multiple Registers

Class 1
Read Coils
Read Discrete Inputs
Read Input Registers
Write Single Coil
Write Single Register
Read Exception Status

Class 2
Write Multiple Coils
Read File Record
Mask Write Register
Read/Write Multiple Registers
Read FIFO Queue

Figure 11 - Function classes.

Modbus APDU

Client requests and server replies are coded into messages called *Application layer Protocol Data Unit* (APDU). Modbus defines three distinct APDUs:

- Reply.
- Response.
- Exception.

All three APDUs share the same structure (Figure 12). They start with the *Function code* being requested or to which a reply is being made. Following this identifier comes all data and parameters which are specific of that function. Data and parameters have a variable number of bytes, depending on the function in question.

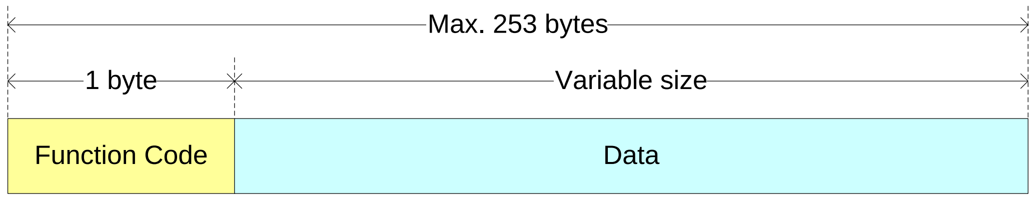


Figure 12 - Modbus APDU.

All APDUs are limited in size to a maximum of 253 bytes, due to limitations imposed by the underlying EIA/TIA-485 layer. Modbus also specifies that all 16 bit addresses and data items are encoded using big-endian representation.

The request APDU is sent by the client to the server. Upon successful completion of the desired function, the server replies with a response APDU (Figure 13).

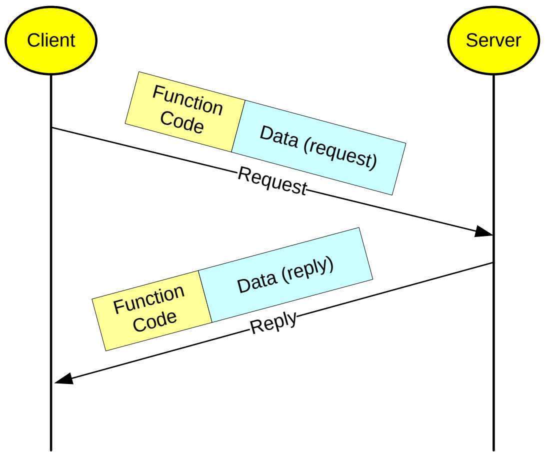


Figure 13 - Normal response.

If the server encounters a communication error (e.g. frame error, CRC/LCR error) it doesn't send any response to the client (Figure 14). The client is responsible to detect the absence of responses (eg. using a timeout).

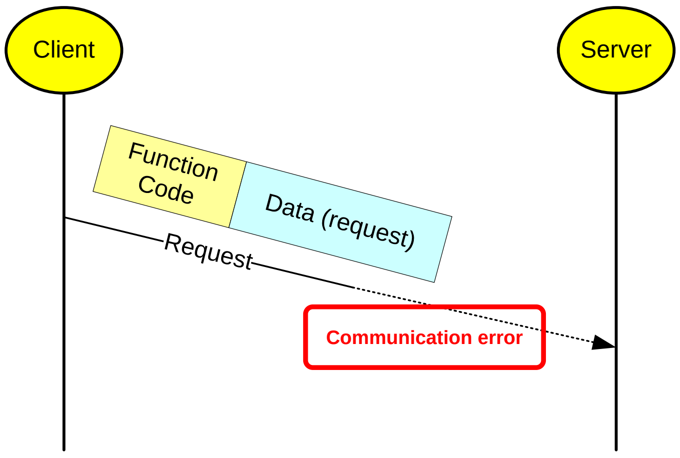


Figure 14 - Communication error.

If the server receives a request that cannot be executed it notifies the client with an *exception response APDU* (Figure 15).

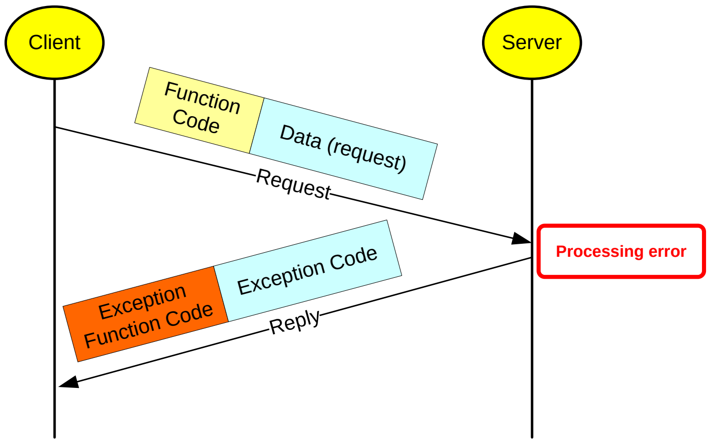


Figure 15 - Exception response.

Exception response APDU contains only 2 bytes. The first one contains the *function code* that caused the error but with the *Most Significant Bit* (MSB) set to 1. This allows the client to distinguish between regular and exception responses. The second byte contains a code value, called *exception code*, which indicates the cause of the error (Figure 16).

Exception Code	Name
1	Illegal Function
2	Illegal Data Address
3	Illegal Data Value
4	Slave Device Failure
5	Acknowledge
6	Slave Device Busy
9	Memory Parity Error
10	Gateway Path Unavailable
11	Gateway Target Device Failed to Respond

Figure 16 - Exception codes.

Modbus TCP

The Modbus TCP version [3] enables Modbus devices to communicate using the TCP/IP family of protocols. This allows that any Modbus devices connected to the Internet can communicate with each other.

The communication process follows the Client-Server model defined in the Application layer. Modbus APDU messages are encoded into Modbus TCP frames which are sent through a TCP connection previously established between the client and the server. Connection establishment and management is handled by the TCP/IP protocol, and occurs independently of the Modbus protocol. TCP port 502 is registered (i.e. reserved) for Modbus applications. A server listens on this port for requests from clients that wish to establish a new connection.

A client can establish multiple connections with different servers, while a server can maintain multiple connections with different clients. It is also allowed that a client to send consecutive requests to a server over the same connection, or using multiple connections (if the server support concurrency), even before the arrival of the replies to the previous requests. Any device can act as client and server at the same time.

The Modbus Messaging on TCP/IP Implementation Guide [3] suggests that client and server implementations should be based on the *sockets interface*.

Frames

A Modbus TCP frame has the same structure for a request or a reply. It consists of an MBAP header (*MODBUS Application Protocol header*) followed by the application layer APDU (Figure 17).

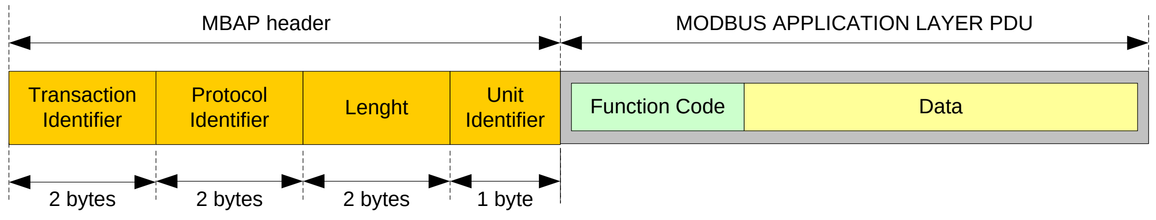


Figure 17 – ModbusTCP frame.

The MBAP header is composed by four fields:

– **Transaction Identifier**. When a client issues several concurrent transactions (i.e. requests) over the same connection, it is not guaranteed that the respective replies arrive in the same order that the requests were sent. To match each request with the correspondent reply it is necessary to identify at which transaction they belong. This field is initialized by the client (usually with the value 0) when it starts a new connection with the server, and is incremented by one every time a new request is made. The server echoes this value in the response frame.

– **Protocol Identifier**. Used to identify the protocol and to support future multiple versions of the protocol. Presently there is only one version of the protocol and consequently this field contains a 0 value.

– **Length**. This field indicates the size (in bytes) of the Unit Identifier field plus the APDU. Data transfer in a TCP connection is performed as a stream of bytes, which could lead to a situation where several frames are waiting to be read in the reception buffer (either clients or servers). By sending this field it enables the receiver to identify frame boundaries in these situations.

- **Unit Identifier**. This field is used to identify a destination device on a Modbus serial network (RTU or ASCII). A server can act as a *device gateway* between Modbus/TCP and Modbus serial networks (Figure 6). In this case if the client wants to send a request to a device on Modbus serial network it must indicate their address, which is sent on this field. Modbus TCP devices which are not gateways usually ignore this field.

Unlike Modbus serial, Modbus TCP frames don't have an error detection field. This was considered unnecessary as the TCP/IP stack already includes several error detection mechanisms. Further details about the implementation of Modbus TCP can be found in [3].

3 Lab – Week 3

The aim of these lab classes is to implement a Modbus TCP client and server running on a PC platform. These classes take place in two consecutive weeks, addressing the following topics:

- Week 3: Develop basic client and server applications using the sockets API.
- Week 4: Develop a basic Modbus TCP client and server.

Before starting this class the students should review the main concepts related with the TCP/IP *sockets* API. Reference [5, Chapter §5] provides an excellent introduction on this subject.

3.1 Equipment

- Personal computer (PC) with an operating system (OS) based on Linux or Windows.

3.2 Development environment

Although applications proposed in this lab task can be developed using any language, it is recommended to use the C language [6] due to its wide support across different operating systems.

The minimum development environment consists of the following elements:

- A text editor used to write the source code.
- TCP/IP stack, including the respective development code.
- A C compiler (e.g. gcc) to compile the source code into an executable application.
- A *bash shell*-like environment to invoke both the editor and compiler and to execute the applications.

A Linux-based operating system provides automatically all these features. In a Windows-based system is easy to support similar features by using Unix-like emulation environments such as Cygwin [7].

3.3 Development of a basic TCP client

To illustrate a basic interaction between a client and a server using the sockets API, it is proposed develop an ECHO client using the TCP protocol (*Transmission Control Protocol*).

This client connects to an ECHO server. The server receives data from the client and echoes it back (i.e. replies with received data). Reference [5, Chapters §5, §7] presents some examples how this client can be implemented.

The client application should go through the following steps:

1. Get data from the user (e.g. a string of characters received from the keyboard).
2. Establish a connection with the server (using their IP address and port).
3. Sent data to the server.
4. Wait for the server response.
5. Print the received data.
6. Close the connection with the server.

This client can be tested against an ECHO server. This server can be either local or remote. For testing purposes a local server (i.e. running at the same machine as the client) is sufficient. All operating systems offer such service through a local server available at TCP *port* number 7 [8]. In most of the cases this server is already active at the computer and no configuration is necessary. Otherwise, the system administrator can easily configure it and activate it. To use this server the client must send their requests to the *localhost* IP address, i.e. 127.0.0.1.

3.4 Development of a basic TCP server

The goal of this assignment is to develop a TCP ECHO server. To simplify the development it is assumed that the server supports a single connection. I.e., no more than one client is allowed to connect to the server at the same time. Reference [5, Chapters §8, §10] presents some examples how this server can be implemented.

The server application should go through the following steps:

1. Get the port number from the user (e.g. using the keyboard)..
2. Binding the server to that port.
3. Listen for client connections.
4. Accept a connection from the client.
5. Receive data from the client.
6. Echoes data to the client.
7. Close the connection.

This server can be tested against the client developed in the previous task.

3.5 Development of a TCP concurrent server

This is an advanced and optional task that should accomplished out of classes.

The main goal is to develop a server that accepts several requests from different clients at the same time, i.e., a concurrent server, Reference [5, Chapters §11, §12, §13] presents some examples how this server should be implemented.

Basically there are two alternatives how to implement this server:

a) Using a single process to process all client requests. This can be accomplished in two ways:
   i. Using the *select()* socket function, inside a loop, to know which clients are active (i.e. have data to be echoed).
   ii. Using *threads* within a process. Each thread is responsible for answering to a request from a single client.

b) Employing one process per client. A parent process is used to receive connections requests from all the clients. After receiving the request, this process dispatches a *child* process which is responsible for answering to that client. This approach requires the use OS primitives such as *fork()* to support concurrency among processes.

Alternative a.i) is the most simple and straightforward implementation. Therefore it would be a good starting point for the server implementation.

Although this server can be tested against the client developed previously (e.g. by using several clients at the same time), it would be difficult to assess concurrency aspects. Since a transaction (send-receive data) has a very short duration and the respective connection is only established during this period, it would be very difficult to have two, or more, connections active at the same time. This problem can be circumvented by making a small modification on the client and server structure: the connection should be closed only by the client upon user initiative (e.g. CTRL-C keystroke).

Regarding the client, it would be interesting to implement the following aspects:

1. The server address and port should be supplied by the user through the keyboard.
2. Include a *timeout* mechanism to detect the lack of response from the server (e.g. the server crashes).
3. Print information regarding error messages (eg. “Connection cannot be established”).

4 Lab – Week 3

4.1 Implementing a basic Modbus TCP Client and Server

The goal of this assignment is to develop a basic Modbus TCP client and server. Both applications should be developed according to the layered structure presented in Figure 5:

- The **User layer** implements the part of the application responsible by the user interface (e.g. requests from the user, print messages on the screen, receive keyboard input, etc.)
- The **Modbus application layer** implements the required Modbus functions.
- The **Modbus/TCP layer** is responsible to encode/decode APDUs received/sent from the application layer to the TCP/IP stack.

Since the main goal is to build a basic client and server it is only necessary to implement a subset of the Modbus functions available. Taking into account the requirements imposed by the Mini Project the functions that must be implemented are:

- Read Discrete Inputs (#2)
- Write Multiple Coils (#15)

To simplify the interaction between adjacent layers, the client should adopt a synchronous approach when invoking functions across layers. I.e., the program flow is blocked at the function calling and continues when the function returns. For illustrative purposes Figure 18 shows an example of the code structure for the client.

```
main()
{
        /* USER LAYER */
        …
        /* Read user requests */
        scanf(..);
        …
        /* Read Discrete Inputs request */
        /* Sends the request to the lower layer and waits for the results */
        modbus_function_02(..)
        …
        /* print results */
        printf(..)
        …
}

modbus_function_02(..)
{
        /* MODBUS APPLICATION LAYER */
        …
        /* Build Modbus APDU */
        …
        /* Send the APDU to the lower layer and waits for the results */
        modbus_tcp(..)
        …
        /* return Modbus function results to the upper layer*/
}

modbus_TCP(..)
{
        /* MODBUS TCP LAYER */
        …
        /* Build ModbusTCP PDU */
        …
        /* Send the PDU to the socket interface */
        write(..)
        …
        /* Wait for the response at the socket interface*/
        read(..)
        …
        /* Returns Modbus APU to the upper layer*/
}
```

Figure 18 – Modbus client code structure example.

The *ModbusTCP* layer can be built reusing the code developed for the TCP ECHO client, namely the establishment and closing of TCP connections. Modbus TCP [3] recommends to use a single connection between the client and the server, and which should be active for as long as possible (ie. do not open and close it for each transaction).

The *User layer* is strongly dependent from the application requirements. For this particular client there aren't specific requirements, so we recommend to build a very simple interface that asks to the user which function to invoke (functions #2 and #15) and the respective parameters.

The server code can be developed in a similar way, i.e. using a layered architecture. It is suggested to reuse the code previously developed for the TCP ECHO server. The Modbus TCP document [3] presents a detailed discussion regarding the server implementation.

Client and server can be tested against third party applications (servers and clients), such as [9], [10], [11] and [12].

5 Seminar – Week 3

Students should perform a study of the TCP communication protocol, and the sockets API. This survey should focus on the following aspects:

- Semantics of the TCP protocol – connection oriented
- Relationship to the IP protocol
- Addressing in the IP protocol (IP addresses, and ports)
- socket API

6 Seminar – Week 4

Students must perform a survey regarding the availability of Modbus TCP implementations (client and server) using the C language. This survey should focus on the following aspects:

- Structure of the code (client and server).
- Functions implementation.
- Server concurrency.
- Source code licensing.

7 Mini-project – Week 3

The goal of this assignment is to develop an application to control a factory scenario composed by 3 conveyers (CVi, i=1,2,3) as presented in Figure 19.

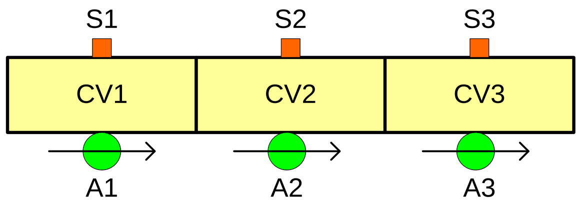


Figure 19 – Factory scenario example.

This scenario is emulated by the Manufacturing Cell Model which was described on Chapter §1 of this module. Conveyor sensors and actuators can be accessed through the Modbus server embedded at the simulator. Sensors (Si, i=1..3) are mapped as *Discrete Inputs* and actuators (Ai, i=1..3) as *Coils* (Figure 20). By default, this server listens for client connections at TCP port 5502.

Name	Address
S1	0
S2	1
S3	2
A1	1
A2	3
A3	5

Figure 20 – Conveyors's Modbus addresses.

Control requirements are as follows:

5. Wait for a part to be placed (manually) at conveyor CV1.
6. Move this part to conveyor CV3, through conveyer CV2.
7. Wait for this part to be removed (manually) from conveyer CV3.
8. Go to step 1.

The control application must include a Modbus client to access to the sensors/actuators at the server. The Modbus client developed previously should be used as a basis for this application.

To take the correct decisions the client must have the most recent data regarding the image of the process (i.e. sensor outputs). The simplest way to obtain this data is to use a polling approach. That is, the client is continually making requests to the server about sensor outputs (e.g. sent a new request after receiving the response to the last one). These requests can be easily implemented inside a *while loop*, that can also be used to send commands to the process (i.e. turn on/off actuators).

During week 3 the students should implement a control program that fulfills the above requirements. This control program does not need to connect to the plant floor simulator – we suggest you write the headers of functions to read the sensors and write to the actuators. These functions will be written in the following week. Test your application by forcing the variables.

8 Mini-project – Week 4

During week 4 the students should complete the requirements described in the previous section (week 3), including the functions that use Modbus/TCP to read and write to the plant floor simulator. Test your application by using the plant floor simulator.

9 References

[1] http://www.modbus.org/
[2] Modbus Application Protocol Specification V1.1b3, April 26, 2012, available from http://www.modbus.org/
[3] Modbus Messaging on TCP/IP Implementation Guide V1.0b, Modbus-IDA, October 24 2006, available from http://www.modbus.org/
[4] Modbus over Serial Line Specification and Implementation Guide V1.02, December 20, 2006, available from http://www.modbus.org/

[5] D. Comer, L. Stevens, Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications, Linux/Posix Sockets Version, 2000.
[6] B. Kerninghan, D. Ritchie, The C Programming Language, 2nd edition,1998.
[7] https://www.cygwin.com/
[8] http://tools.ietf.org/html/rfc862
[9] Digslave Modbus Slave Simulator, available at http://www.modbusdriver.com
[10] Modpoll Modbus Master Simulator, available at http://www.modbusdriver.com
[11] Modbus Slave, available at http://www.modbustools.com/
[12] Modbus Poll, available at http://www.modbustools.com/