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: 2.4.4

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

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 Politecnica de Valencia (UPV)
Name of representative :	Houcine Hassan
Address :	Camino de Vera, s/n. 46022-Valencia (Spain)
Phone number :	+34 96 387 7578
Fax number :	
E-mail :	husein@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	19/09/2014	Paulo Portugal	Revision
6.0	19/09/2014	Mário de Sousa	Review

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)

- The PBL methodology was presented in WP 1.1
- The list of the module's chapters and the temporal scheduling in WP 1.2
- The required human and material resources in WP 1.3
- 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 fourth chapter – Modbus/Serial.

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

2 Lecture

This lecture aims to present the Modbus Serial protocol. Before starting reading this chapter, readers should review the material on the Modbus that was presented in chapter §2 of this deliverable.

2.1 Modbus Serial

The Modbus serial protocol deals with the transmission of Modbus messages (APDUs) [1] over serial lines, such as EIA/TIA-232 or EIA/TIA-485 networks [2].

The use of serial mediums raises an important problem: since the medium is shared between all devices, it is necessary to adopt a medium control mechanism that ensures that only one device at time is authorized to use the medium to initiate its transmissions. The use of a Client-Server interaction model between devices (as defined in the Modbus application layer) is not adequate on these circumstances since it does not guarantees a mutually exclusive access to the medium. For that reasons Modbus Serial adopts a more restricted interaction model, in particular it uses a *Master-Slave* model (Figure 1).

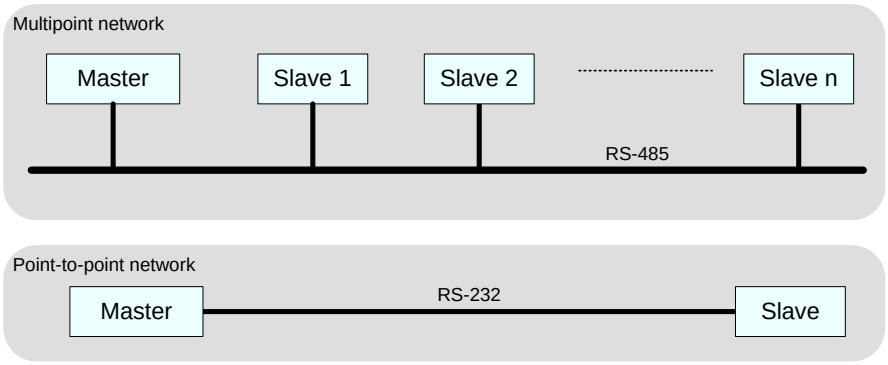

Figure 1 – Examples of Modbus serial networks.

According to this model, there is only one master in the network that coordinates the medium access of slaves. The master is responsible for initiating the communication by sending requests to the slaves. These in turn, reply to the master with the requested data. The request/reply exchange (i.e. a *transaction*) can be performed in two ways (Figure 2):

– **Unicast mode**. The master sends a request for a specific slave. The slave processes it and replies to the master.
– **Broadcast mode**. The master sends a request to all slaves. The slaves process it but don't reply to the master.

The master only initiates one transaction at the time, i.e. after the previous transaction has ended.

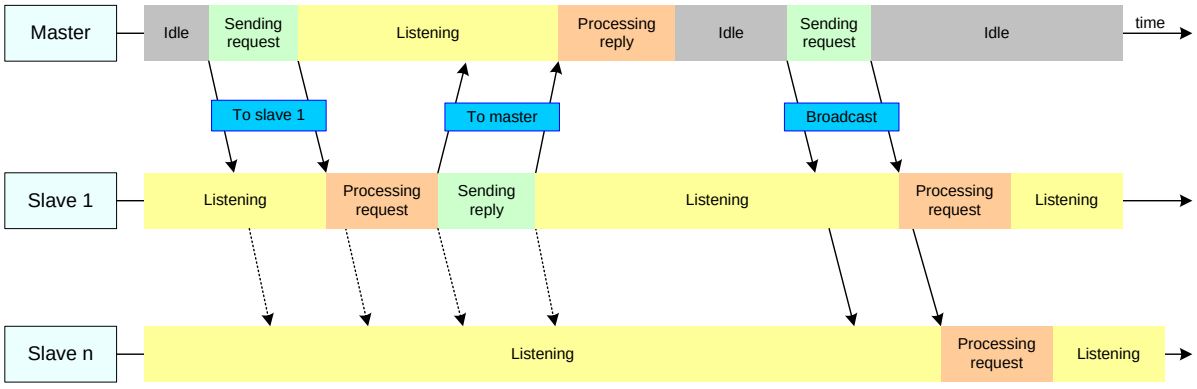

Figure 2 - Request/reply Exchange.

Each slave has a unique *address*. This address is used by the master to issue requests on unicast mode. A device can be both master and slave, but not at the same time. Therefore, a device can change its role whenever appropriate.

2.2 Frames

The request/reply exchange is performed using frames that share the same structure (Figure 3).

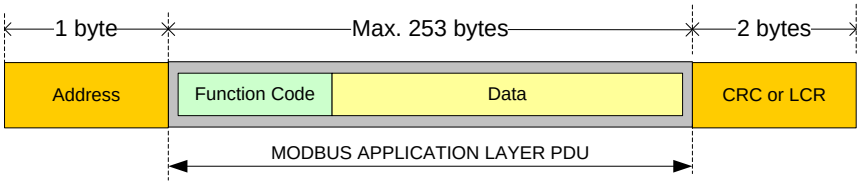

Figure 3 - Modbus serial frame.

A frame contains the application layer PDU (APDU) delimited by header and trailer fields, as following:

- Address. This field contains the slave address. In a request frame it identifies the destination device, while in a reply frame it identifies the sender. Each slave has a unique address in the range 1-247. Address 0 is used by the master for broadcasting requests. The range 248-255 is reserved.
- Modbus APDU. This field is the application layer PDU. Please refer to chapter §2 of this deliverable for details regarding their contents.
- CRC or LCR. This field is used for error detection purposes. The contents depends on the transmission mode (RTU or ASCII) being used.

Frames exchanged between master and slave devices can be transmitted in one of two modes: RTU (*Remote Terminal Unit*) or ASCII (*American Standard Code for Information Interchange*). Each mode defines different ways to code the frame contents for transmission.

The main observable difference between both modes is that RTU mode produces shorter frames than ASCII mode. Consequently, for the same transmission rate, the RTU mode achieves a higher throughput. However, this is achieved by requiring that RTU devices comply with strict timing requirements which imposes more complex software/hardware implementations.

All devices must implement the RTU mode, with the ASCII mode being optional. A device may support both modes, but never simultaneously. Within a network all devices must use the same transmission mode otherwise the communication cannot occur since both modes are incompatible.

2.3 RTU mode

The RTU (*Remote Terminal Unit*) mode assumes that each Modbus device is equipped with a UART (*Universal asynchronous receiver/transmitter*) able to support asynchronous transmissions. Each byte, within a frame, that is sent to the UART is transmitted in the physical medium using 11 bits (refereed as *UART character*) as following (Figure 4):

- 1 start bit (ST): used to guarantee the initial signal synchronization between the sender and the receivers.
- 8 bits (D0…D7): *data* being sent coded in binary with the least significant bit sent first (D0).
- 1 parity bit (PT): used for error checking purposes.
- 1 stop bit (SP): used to ensure that there is a minimum idle time between consecutive UART character transmissions.

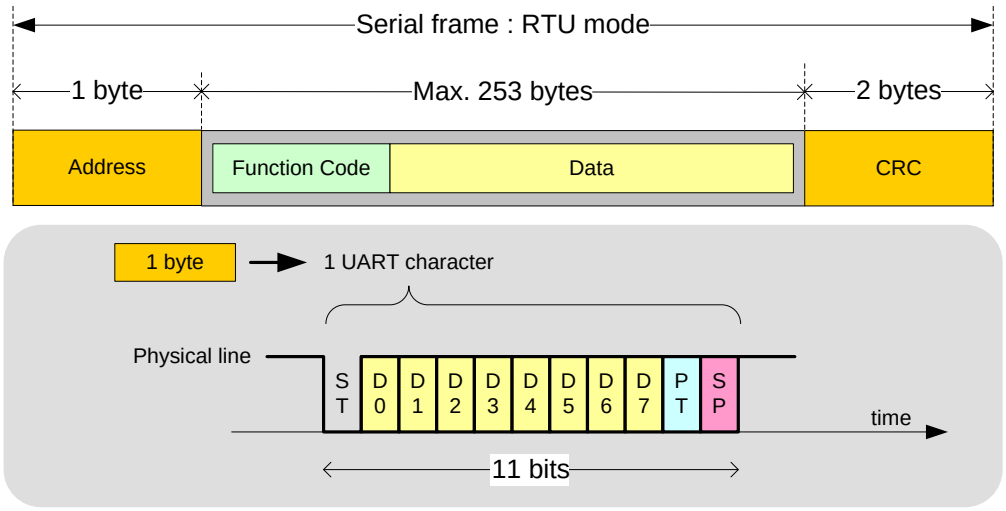

Figure 4 - RTU transmission.

Devices may choose any parity checking method available (e.g. Odd, Even, Mark, Space, None) for transmissions. However, all devices within the network must use the same method, otherwise communication errors will occur. To ensure that any two Modbus devices have the minimum communication requirements, all devices must support the *Even* parity method. When the *None* method is chosen, it is necessary to use 2 *Stop bits* to guarantee that the size of the UART character remains constant, i.e. 11 bits.

Since each byte within the frame is transmitted in an asynchronous way (i.e. one at time, separated by a minimum interval) is necessary for decoding purposes that the receiver identifies where a frame starts and ends. The absence of frame delimiters (special header and trailer characters) makes this task difficult.

To overcome this problem, Modbus requires the character transmissions compels with tight timing requirements. The process is conceptually simple: if two characters are transmitted close enough the receiver assumes that they belong to the same frame. Otherwise, they represent, respectively, the last character of a frame and the first character of the next frame. Modbus defines the following rules for the transmissions (Figure 5):

- The time interval between characters belonging to the same frame should not exceed 1.5 character times, i.e. the time necessary to transmit 1.5x11 bits. If this interval is exceeded, the receiver considers that the frame is incomplete and discards it.
- Consecutive frames must be separated by an idle interval with a minimum 3.5 character times.

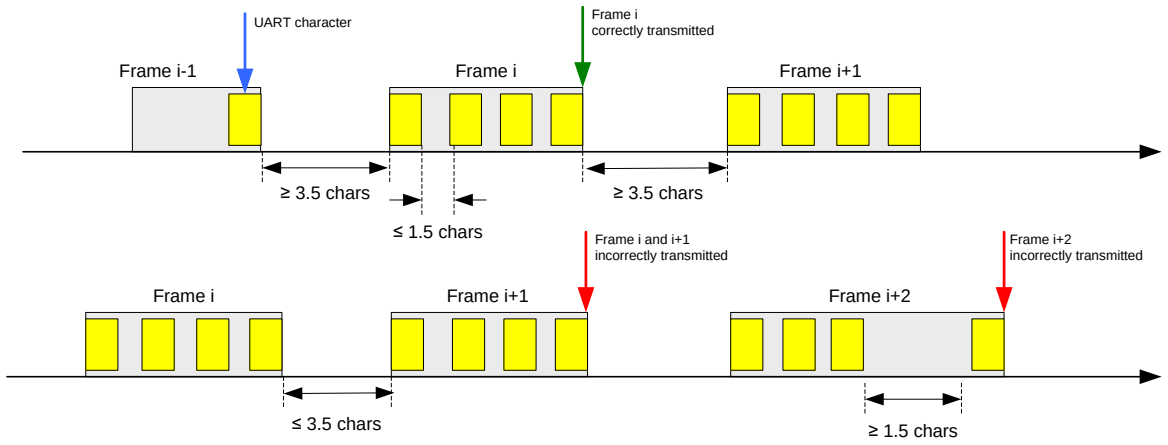

Figure 5 - RTU timing requirements.

From this description, it is easy to understand that some care must be taken when implementing an RTU driver, particularly with regard to the accuracy of the timers used. Although efficient, the RTU mode requires a tight control of timing.

2.4 ASCII mode

In this mode it is also assumed that each device uses a UART for transmitting. However, before starting a transmission the Modbus serial frame is coded using the ASCII (*American Standard Code for Information Interchange*) code. The coding process is simple: each byte is transmitted as 2 ASCII characters (Figure 5).

The coding process is as follows: each 8 bit byte is divided into 2 nibbles of 4 bits each. One nibble contains the 4 most significant bits in the byte, and the other the 4 least significant bits. Each nibble is then coded using its hexadecimal representation on the ASCII alphabet. For example, a byte with value 0xA1 (or 161 in decimal) is transmitted as the characters ‘A’ followed by ‘1’. The character 'A' is sent as the byte with value 0x41 which is the ASCII representation of the 'A' character. Likewise, the character '1' is transmitted as its ASCII value of 0x31.

Contrary to the RTU mode, a frame transmitted in the ASCII mode has a header and a trailer. The header identifies the beginning of a frame and is represented by the ‘:’ ASCII character (0x3A). The trailer identifies the end of the frame, and comprises 2 ASCII characters: CR (*Carriage Return* – 0x0D) and LF (*Line Feed* – 0A), transmitted in this order.

The use of headers and trailers helps the reception process, since the receiver has only to wait for the ‘:’ character to detect the beginning of a frame. Conversely, the end of a frame is detected when the sequence ‘CR’’LF’ occurs. Since this mode does not specify a maximum time interval between consecutive characters, this implies that this interval could theoretically assume any value. To prevent this situation, Modbus suggests using a *timeout mechanism*.

Each ASCII character is transmitted using 10 bits, as follows:

- 1 start bit (ST).
- 7 data bits (D0…D6): the ASCII character (‘0’-‘9’, ‘A’- ‘F’) with the least significant bit sent first.
- 1 Parity bit (PT).

- 1 Stop bit (SP).

The requirements for parity checking are the same as those of the RTU mode.

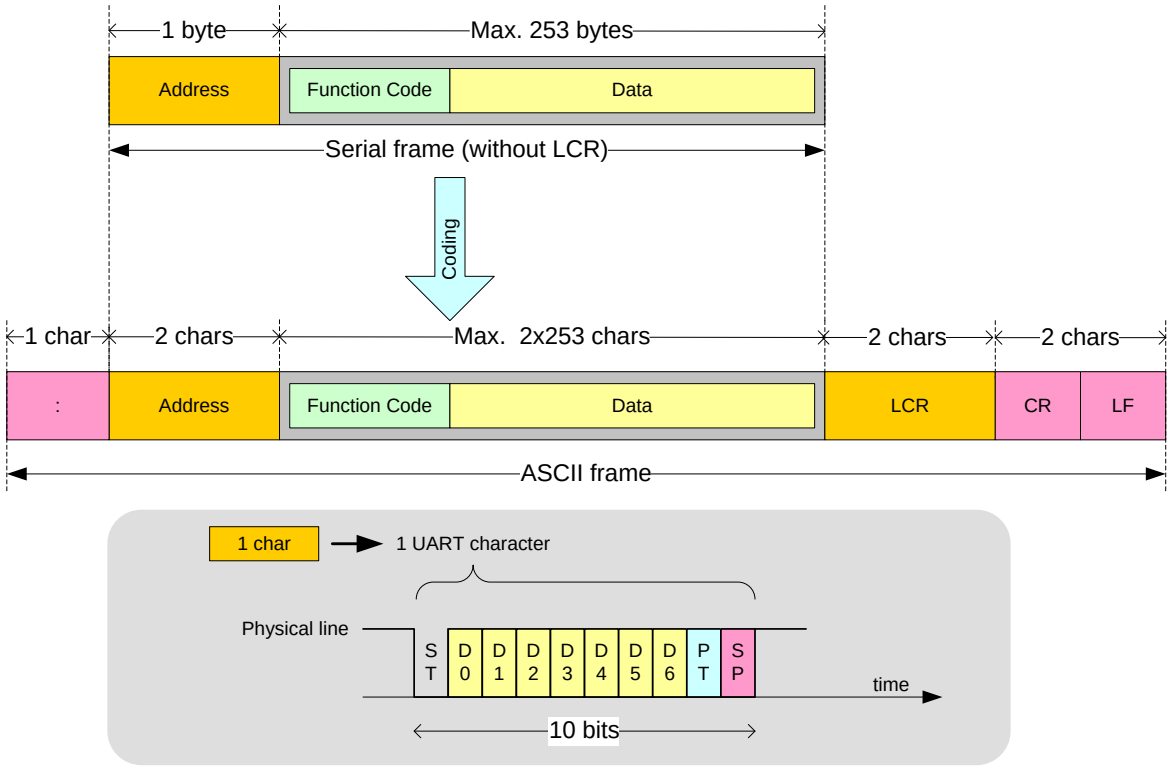

Figure 6 - ASCII transmission.

Each time it receives a character, the receiver device triggers a countdown timer. If a character is received and the timer has not expired (i.e. 0), the receiver accepts this character as valid and cancels the times. Otherwise, if the timer expires (i.e. a *timeout* occurs) the received data is discarded and the receiver waits for the next frame. Modbus doesn't define any specific value for this timer, but suggests that 1s value should be adequate. In practical situations, the value of this timer should be adjusted according the network transmission rate.

It is easy to understand that the ASCII mode is simple to implement and doesn't pose any particular timing requirements. However, the price to pay is almost the doubling of the frame size when compared to the RTU mode.

2.5 Error detection

Modbus serial uses two different error detection mechanisms that are located on different levels: character level and frame level.

At the character level, a parity checking method is used. The sender computes the parity of each character using the parity method chosen (Even, Odd, Mark, Space or None parity) and sets the parity bit accordingly. The receiver, using the same method of the sender, computes the expected parity bit and compares it with the received parity bit. If they are different, an error has occurred and the entire frame is discarded. This method has clear limitations, since any number of even errors (2, 4,..) is undetected.

Error checking at the frame level uses a CRC (*Cyclic Redundancy Check*) in the RTU mode, or LRC (*Longitudinal Redundancy Check*) in the ASCII mode. Although these methods are conceptually different, the approach is very similar. In simple terms, both methods use a mathematical algorithm that when applied to the frame contents produces a single value (called *signature*). This value is them appended to the frame before transmission. When the frame is received, the receiver removes the CRC/LCR field and applies the same algorithm to the contents received. If the computed value is equal to CRC/LCR field received, then is assumed that a frame doesn't have errors. Otherwise, it has errors and is discarded.

The CRC value is computed through a polynomial division of the frame using $X^{16} + X^{15} + X^2 + 1$ as the generator polynomial. It produces a 16-bit value, which is transmitted with the lower byte first. The LCR value is computed using a modulo-2 sum (XOR-sum) of all bytes of the frame, and produces an 8-bit value. A comparison between both methods shows that the CRC has a higher error detection capability. The CRC and LCR values are coded using the same approach (RTU, ASCII) used for the remaining characters.

2.6 Physical layer

Modbus serial supports 2 physical layers: EIA/TIA-485 (also known as RS-485 [5]) and EIA/TIA-232 (also known as RS-232 [6]).

RS-232 is a point-to-point solution that supports only 2 devices, 1 master and 1 slave, using a full-duplex connection (i.e. both can communicate at the same time). The standard defines both a mechanical and an electrical interface. The mechanical interface specifies the type of connectors used to connect a cable between devices (i.e. the plugs). The most common types are: RJ45, DB9 and DB25 connectors. The electrical interface defines how data is coded into electrical signals and the respective transmission. RS-232 uses non-differential voltage levels to transmit data (i.e. they use a common ground). A logical one is transmitted as a voltage in -3V to -25V range, while a logical 0 is transmitted as a voltage in +3V to +25V range. Signals received in a +3V to -3V range are considered invalid. The use of a non-differential transmission has two important limitations. First, it has a small robustness to external inferences. When electrical noise is superimposed on the signal is it difficult to establish their logical level. Second, the transmission distance is reduced due to the signal attenuation. This last aspect is also affected by the transmission rate. Higher rates impose short transmission distances. It's uncommon to use high transmission rates (e.g. 110Kbps), unless equipments are at close distances (e.g. <1m). Common figures are 9600bps for distances less than 10m. The electrical interface also defines several logical signals. The most important ones are used for data transmission (TX - Transmit, RX – Receive) and for handshaking purposes (DTE – Data Terminal Ready, DSR - Data Set Ready, RTS – Request to Send, CTS – Clear to Send). The latter signals have an important role since they can be used to regulate the data flow. In most cases, cables have a number of wires from 3 (TX, RX and ground) to 7 (TX, RX, GND, RTS, CTS, DTE, DSR) (Figure 7).

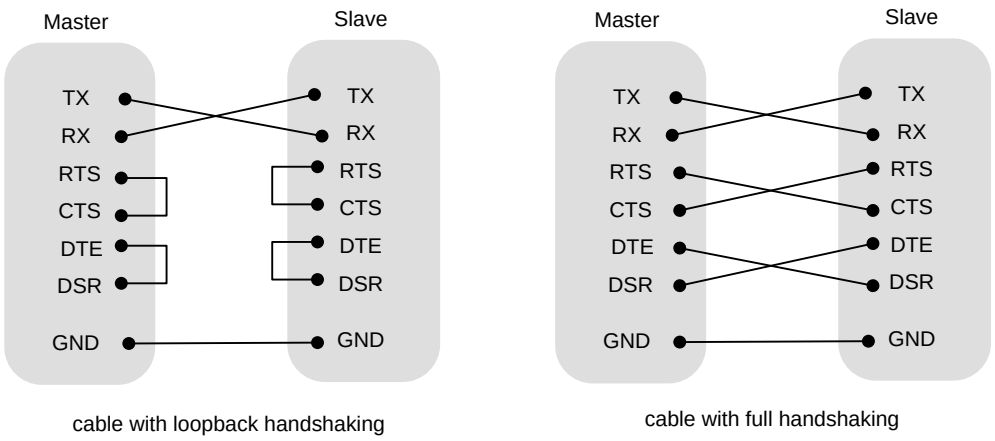

Figure 7 - RS-232 cable connections.

The RS-485 standard defines of a multi-drop network able to support multiple devices (typically 32 devices per network segment). Several segments can be connected by using repeaters, thus increasing the maximum number of devices in the network (Figure 8).

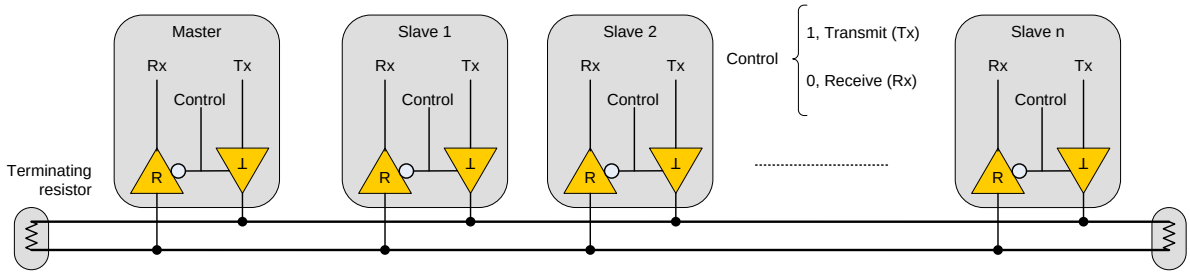

Figure 8 - RS-485 connections.

Unlike RS-232, it only defines an electrical interface. Data is transmitted using a differential voltage signal using a pair of wires. To avoid signal reflections, the network must be terminated with a 120 ohm resistor at each end. This approach enables high robustness to interferences (due to noise cancellation) and long transmission distances (due to attenuation cancellation). It's common to found transmission speeds up to few Mbps and distances up to few Km. These properties make it suitable to be used in industrial environments. However, it only supports half-duplex communications. That is, at any instant if one device is transmitting the remaining must listen. Is therefore necessary to use mechanisms at the upper layers to decide which device is authorized to transmit. Moreover, it also doesn't support any type of handshake mechanism, which, if necessary, should be implemented by the upper layers.

All Modbus devices must support an RS485 interface, while the RS232 is optional. Both transmission modes (RTU and ASCII) can be used on either interface. Moreover, all devices within a network must have the same configuration, comprising the following parameters: transmission mode (RTU or ASCII), parity type (Even, Odd or None parity) and transmission speed (bits/second). All devices must support 9.6 Kbps and 19.2 Kbps rates, with the latter being the default value [2]

3 Lab – Week 7

The aim of this lab class is to implement a Modbus RTU Master running on a PC platform.

3.1 Equipment

- Personal computer (PC) equipped with two RS-232 serial ports and with an operating system (OS) based on Linux or Windows.

3.2 Development environment

The development environment must have characteristics which are similar to that described in Chapter §2 Section §3.1.2 of this deliverable. There is however an important difference: the communication stack. For this lab is necessary to use a communication stack for a PC serial port. To simplify the development process it is provided a C library called *serial_util* that implements this type of stack (Figure 9). The main advantage of this stack lies in their simplicity and in the fact that can be used either on a Linux-based OS or in a Cygwin environment (running on Windows). This library provides an API (*Application Programming Interface*) with 5 functions:

- Serial_open() : opens the serial port.
- Serial_read(): receive characters from the serial port.
- Serial_write(): send characters to the serial port.
- Serial_close(): closes the serial port.
- Serial_config(): configures the serial port.

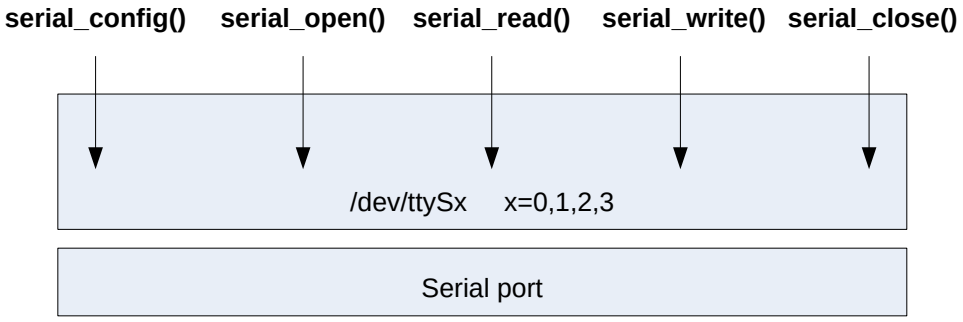

Figure 9 – Serial_util library API.

The source code of the library is available in two files:

- *serial_util.h* : definitions
- *serial_util.c* : source code of the functions

The outline of these functions is presented below.

```
int serial_open (const char* device);

// Opens the serial port /dev/ttySx, x=(0,1,2,3)
//
// Arguments:
//
// device: string with the identification of the serial port (e.g. "/dev/ttyS0")
//
// Returns:
//
// >0: success, serial port descriptor
// <0: error
//
// Note:
//
```

```
// PC serial ports have the following identification strings
// "/dev/ttyS0" : COM1
// "/dev/ttyS1" : COM2
// "/dev/ttyS2" : COM3
// "/dev/ttyS3" : COM4
//
```

```
int serial_read(int fd, uchar *buf, uint bufsize, uint timeout);

// Read characters from the serial port
//
// Arguments:
//
// fd: serial port descriptor obtained with serial_open()
// buf: buffer used to store received characters
// bufsize: buffer size (in bytes). Max. number of characters expected.
// timeout: maximum waiting time, in seconds, that the function waits for the characters
//
// Returns:
//
// >=0: success, number of characters received
// <0: error
//
// Notes:
//
// This function blocks (synchronous calling) until one the following events occurs:
// --- the timeout timer expires
// --- receives the number of characters specified in bufsize
//
```

```
int serial_write(int fd, uchar *buf, uint bufsize);

// Sends characters to the serial port
//
// Arguments:
//
// fd: serial port descriptor obtained with serial_open()
// buf: buffer with the characters to be sent
// bufsize: size (in bytes) of the buffer (number of characters to send)
//
// Returns:
//
// >=0: success, number of characters sent
// <0: error
//
// Notes:
//
// This function blocks (synchronous calling) until all characters are sent to the serial
// port device driver.
//
```

```
int  serial_close (int fd);

// Closes the serial port
//
// Arguments:
//
// fd: serial port descriptor obtained with serial_open()
//
// Returns:
//
// >0: success
// <0: error
//
```

```
int serial_config(int fd, tcflag_t baudrate, tcflag_t charsize, tcflag_t stopbits, tcflag_t
     parity);
```

```
// Configures serial port parameters
//
// Arguments:
//
// fd: serial port descriptor obtained with serial_open()
// baudrate: transmission rate
// charsize: number of data bits per UART character
// stopbits: number of stop bits
// parity: type of parity
//
// Returns:
//
// >0: success
// <0: error
//
// Notes:
//
// The file serial_util.h contains the definitions (#DEFINES) for baudrate, charsize, stopbits
// and parity
//
```

3.3 Modbus RTU Master

The goal of this assignment is to implement a Modbus RTU master. The development guidelines are similar as those defined for the implementation of the Modbus TCP client (see chapter §2). That is, the use of a layered architecture with synchronous calls of functions between layers. The main difference here is that the *sockets* library is replaced by the *serial_util* library.

Taking into account the requirements imposed by the Mini Project this master should implement the following Modbus functions:

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

Additionally, the *user layer* should implement functions that ask the user:

- The slave address
- Which function to request (#2 or #15) and the respective parameters.

To test this application Modbus slave devices can be emulated using third party applications such as [5] and [6]. The Modbus master should use one of the serial ports, while the other is used by the Modbus slave emulator. A serial cable, as described in Figure 7, is used to connect both ports. If the computer has a single serial port it is necessary to emulate additional ports using a serial port emulator such as [7] and [8] (notice that Linux does not need serial port emulators). These virtual ports can be connected using a virtual cable (implemented by the emulator), or by the 'socat' command line utility in Linux.

4 Lab – Week 8

The aim of this lab class is to implement a Modbus RTU Slave running on a Arduino platform.

4.1 Equipment

- Personal computer (PC) equipped with an operating system (OS) based on Linux or Windows. This computer should also have the Arduino development environment installed.
- Arduino board, connected to computer using USB cable
- Buttons and leds to connect to Arduino's physical I/O pins,

4.2 Modbus RTU Slave

Taking into account the requirements imposed by the Mini Project, this slave should be able to reply to requests of the following Modbus functions:

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

You are free to define a mapping between the Arduino's I/O pins and the memory bits that can be addressed by the modbus slave.

To access the serial port of the Arduino, you can use the Serial.open(), Serial.read() and Serial.write() functions. For example:

```
int Rx = 0;
void setup(){
  Serial.begin(9600);
}

void loop(){
  if (Serial.available() > 0) {
    Rx = Serial.read(); //read the incoming byte:
    Serial.write(42); // send one byte = value 42
    int bytesSent = Serial.write(“hello world!”); //send the string
  }
}
```

Test your slave against the master you wrote in the previous lab. To make sure your code is correct, test it also against the master written by another group.

5 Seminar – Week 7

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

- Structure of the code (Master and Slave).
- Functions implementation.
- Server concurrency.
- Source code licensing.

6 Seminar – Week 8

Students should design and discuss the architecture of an implementation of a Modbus slave on the Arduino. Special issues to consider are:

- The program must not block while waiting for a Modbus request frame.

- Determine the blocking properties of the Serial.write() function used to write on the serial port. Writing to a serial port may be very slow, it is not advisable to stop the remainder of the program while sending the reply frame

7 Mini-project – Week 7

The aim of this task is to integrate the control logic application developed in chapter §3, with the Modbus Serial master developed previously.

Consider the following additional requirements for the control logic application:

- The process starts processing parts only when the START button is pressed.
- The processing is stopped when the STOP button is pressed.
- A RED light is used to inform the user that the process is stopped.
- A GREEN light is used to inform the user that the process is running.
- While the MA machine is processing parts the YELLOW light should be on. Otherwise, it should be off.
- While the MB machine is processing parts the BLUE light should be on. Otherwise, it should be off.

Buttons and lights are physically available at Modbus RTU slave devices according to the specifications given in Figure 10. Buttons are mapped as *Discrete Inputs* and lights as *Coils*.

Name	Device	Address
START	1	0
STOP	1	1
RED	2	0
GREEN	2	1
YELLOW	2	2
BLUE	2	3

Figure 10 – Mapping of the buttons and lights on Modbus slave devices.

The control logic application should be modified to include these new requirements. In this context, this application must include the code for Modbus RTU master developed previously. As for the Modbus TCP client (see chapter §3), it is suggested to use a *polling* approach to get/send information from/to Modbus slave devices.

In the first week, you should focus on developing the Grafcet capable of handling the above requirements. Although it is not absolutely necessary, it is probably best if you adopt a hierarchical structure for your Grafcets.

8 Mini-project – Week 8

Integrate and test the extended control application developed in the previous week, with the Modbus slave running on the Arduino, and the Modbus master integrated into the control application.

9 References

[1] Modbus Application Protocol Specification V1.1b3, April 26, 2012, available from http://www.modbus.org/
[2] Modbus over Serial Line Specification and Implementation Guide V1.02, December 20, 2006, available from http://www.modbus.org/
[3] Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems, ANSI/ TIA/ EIA-485-A-1998.
[4] Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange, ANSI/ TIA/ EIA-232-F-1997.
[5] Digslave Modbus Slave Simulator, available at http://www.modbusdriver.com
[6] Modbus Slave, available at http://www.modbustools.com/
[7] Virtual Serial Port Driver, available at http://freevirtualserialports.com/
[8] Virtual Serial Port Driver for Linux (VSPDL) available at http://tibbo.com/soi/vspdl.html?swln=en