// teste 1 – prático –21/22 – 2022/10/14

1. In the Modbus implementation in **2 layers** (Modbus Application and Modbus TCP), the **Modbus Application** layer constructs the APDU packets corresponding to the Modbus functions, and the **Modbus TCP** layer sends and receives APDUs via TCP.

V

2. The data model of **Modbus** defines entities with lengths of **1 bit** and **16 bits**.

V

3. The **port 502** was one of the fields that the client had to insert in the **MBAP** header of **Modbus TCP** layer.

F

4. In a TCP/IP client, it is the function **accept()** that allows you to establish a communication channel to a remote server.

F

5. Consider the following snippet of code:
   \n
   uint16_t val[10];\n
   write(s, val, 10); //* *s* identifies an appropriate socket */
   \n
   Invoking the *write()* function allows you to write all 10 positions of the array *val* in the socket *s*.

F

6. The function *inet_aton()* that you used in the Modbus client was utilized to specify the ** IPv4 address** of the server as a **string** of four decimal numbers separated by dots.

V

7. When observing, with **Wireshark**, the transactions between ModbusSlave and ModbusPoll, you noticed that the **APDU** of the **ModbusAP** layer was inserted in the payload (SDU) of the ** ModbusTCP packet**.

V

8. When using **Wireshark** to observe the Modbus TCP transactions, you confirmed that the data transmited inside the respective TCP packets started with the **Function Code** of the respective function Modbus.

F

9. The **maximum** number of registers that can be **read** in a single **Modbus** read request is the **same** as the **maximum** number of registers that can be **written** in single write request.

F

10. The Modbus clients are able to read the **MBAPDU packets** with a variable length from the **socket TCP** because the TCP protocol signals where those packets start and end.

F

11. According to the specification, the value of the **Transaction Identifier (TI)** to use in the **MBAP header** must always be **0:0** (2 bytes).

F

12. If you would like to add the functions *write_multiple_coils()* and *read_coils()* to the **ModbusAP** layer that you developed, you could continue to use the same function *send_Modbus_request()* that you developed in the **ModbusTCP** layer.

V

13. If the function *read_h_regs()* is invoked by the client in the following form **read_h_regs("192.168.113.174", 502, 400, 150, val)**, then it should **return an error** and not send the Modbus request to the server at all.

V

14. Regarding the organization of the **protocol stack**, it is essential to use *header* files, such as **modbusAP.h** and **modbusTCP.h**.

F

15. When receiving a **MBAP header** in a TCP socket, you were able to learn the **length** of the respective APDU with the following operation:
\n
APDU_len = (uint16_t) MBAP[4]*256 + (uint16_t) MBAP[5] -1; \n
(Note: ***256** is equivalent to **<<8**)

V

16. A **Modbus** client sent the following APDU to the server: **10:00:02:00:01:02:00:10** (hex) and received from the server the following APDU: **90:01** (hex). This situation **does no comply** with the Modbus protocol.

F

17. Consider that you invoke the function *write_multiple_regs()* to write **two variables** of 16 bits in two registers with **Modbus** addresses 2 and 3. The first variable (lowest

address) has the value 256 and the second 1. The function should build the following APDU: **10:00:01:00:02:04:01:00:00:01** (hex)

V

18. Suppose you invoked the function *write_multiple_regs()* in a way that the function produced a **Modbus request** with the following APDU: **10:00:02:00:01:02:00:01** (hex). In the **response**, you received the following APDU: **10:00:02:00:01** (hex). The APDU of the response indicates that the write operation was **successful**.

V

19. The **MBAPDU**: **00:03:00:00:00:05:01:03:02:02:03** (hex) corresponds to a request to read 1 register.

F

20. If your client connects to a remote machine with an operational **TCP/IP protocol stack** and a **running** Modbus server, every time you send it a **correct** Modbus request, you will always get a **valid** Modbus response.

F