Exercise 9: Distributed Applications and the Modbus Protocol

Overview

In previous sessions, only a single microcontroller was used to complete a certain task. However, in many real world applications, the workload is distributed among multiple microcontrollers that interact with each other to achieve an ultimate goal.

Such systems are commonly known as distributed systems and the controllers are viewed as nodes in the system structure. Nodes are typically connected to a network and communicate with each other or a specific set of other controllers according to predefined protocols. A typical example is a car, where a huge number of microcontrollers communicate using various protocols such as LIN, CAN, MOST or FlexRay.

In this session, we will take a look at the simplified version of a protocol widely used in industrial automation named Modbus.

Note: The accompanying documentation should only serve as an in-depth reference on specific topics. Most information required for solving this exercise is already contained in the following description.

The Modbus Protocol

Modbus is an openly published serial communication standard for interconnection of an arbitrary number of devices over a bus. The Modbus specification defines the application layer messaging policy, which can be implemented over serial port or Ethernet (TCP/IP). Two flavors exist for serial connections, namely RTU mode (Remote Terminal Unit) and ASCII mode. The two modes have different representations of numerical data and slightly different protocol details. The RTU mode uses a compact, binary representation of the data, where as ASCII mode is human readable and more verbose. In this lab course, **we will use the RTU-based implementation over the serial line**.

Master, Slaves and Addressing

Modbus is a *Master-Slave based* protocol. This means that at any point in time at most one special device exists in the network – the *master* – that can initiate commands against all other devices – the *slaves* (compare figure 1). A slave can not initiate a transmission on its own. This simple principle restricts the communication in the network, but avoids collisions on the bus altogether.

For the master to be able to talk to a specific slave, each slave is assigned a unique number in the range 1..247, the so-called *slave ID.* Note that only the intended device should act on the command, even though other devices will also receive it.

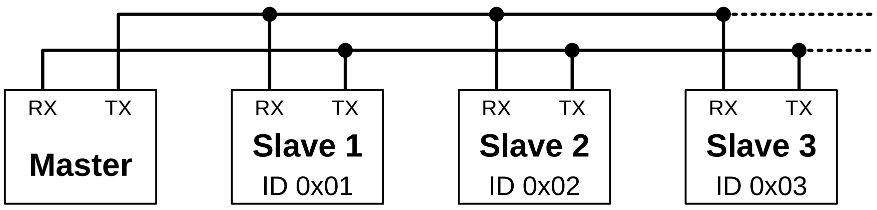


Figure 1: Modbus network with multiple slaves

Function Codes

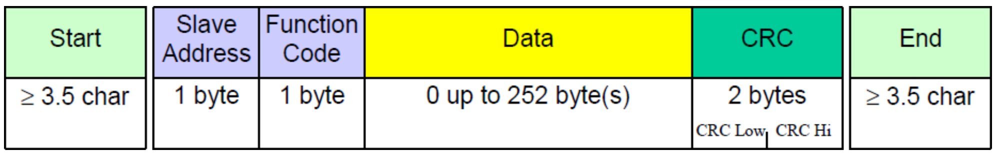
When the master issues a command, the *function code* in the command packet determines the action to be taken by the target slave device. A valid function code is a value between 1 and 255. The command packet also contains data that has to be interpreted in context of the function code.

In this exercise, we will implement a simplified version of Modbus that supports only two function codes, namely `0x03` (*Read Holding Register*) and `0x06` (*Write Single Register*). The details of these two function codes can be found in the Modbus specification.

Modbus Frames

The maximum length of a Modbus RTU frame is 256 bytes. In RTU mode of Modbus, a command PDU contains the information depicted in figure 2.

The 16-bit CRC value is the expected checksum of the frame calculated by the sender. The receiver should compute the CRC of the received frame and compare it with the expected value. Packets with non-matching CRC must be considered as errors. **A routine for calculating the CRC value is given on the network share `P:\`.** This implementation is equivalent to the lookup-table based approach described in the Modbus specification.



Start	Slave Address	Function Code	Data	CRC	End
≥ 3.5 char	1 byte	1 byte	0 up to 252 byte(s)	2 bytes CRC Low / CRC Hi	≥ 3.5 char

Figure 2: RTU frame format

Receiving an RTU Frame

The start and end points of a RTU frame are identified by the so-called inter-frame delays $t_{3.5}$ (compare figure 3). The idea is that RTU frames must be separated by at least 3.5 “byte times”. After receiving a byte, the receiver can start a timer which expires at the time that it takes to transmit 3.5 bytes. When next byte arrives within $t_{3.5}$, the timer is restarted. A timer expiration is considered as the end of the current frame.

The RTU frame must be transmitted in a continuous byte stream. To ensure this, the Modbus specification calls for a check for another interval: if a silence time of more than 1.5 “byte time” ($t_{1.5}$) occurs between two successive bytes, the frame should be considered as incomplete and discarded. However, in our version of the protocol, **we will skip the implementation of this concept** for reasons of simplicity.

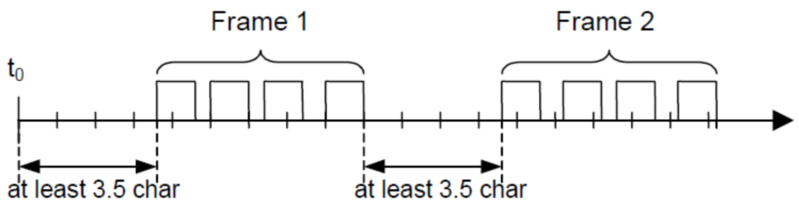


Figure 3: Separation of RTU frames

Data type	**Length**	**Access**	**Description**
Discrete inputs	1 bit	read-only	Boolean input values (e.g., from digital sensors).
Coils	1 bit	read-write	Boolean flags (e.g., program state, configuration).
Input registers	16 bit	read-only	Numeric input values (e.g., from analog sensors).
Holding registers	16 bit	read-write	Numeric values (e.g., program parameters).

Table 1: Modbus data model

Modbus Holding Registers

The Modbus data model defines four primary data types as listed in table 1. There are separate function codes for accessing commands with different data types. In this session, **we will only read from and write to holding registers**.

In our scenario, the slave devices can be viewed as external memories of the master device. The register content is stored in the respective slave devices and can be accessed by certain addresses in the range 0..65535. The value of a holding register can be updated by sending a *Write Single Register* command (`0x06`) or read by sending a *Read Holding Registers* command (`0x03`). Each of these commands receives the address of the register(s) to read or write. In our implementation, **we will restrict the *Read Holding Registers* to only read a single register**, i.e., *Quantity of Registers* = 1.

Modbus Slave Implementation

For the following exercises, connect the LC display to your controller and make sure you can use it for debugging purposes, since there will be no debug console available (the serial line will be used for Modbus communication).

To communicate with your slave device, you can reuse the host side software provided in the EasyLab installation (`C:/Program Files/EasyLab/bin/service_modbus_test.exe`). This software can be used to connect to a Modbus RTU device via serial port. Please ignore the “error retrieving program ID” message upon startup, it is related to the fact that EasyLab implements its own variant of the Modbus protocol. Self-explaining options are provided to read and write the holding registers in the slave device. In the host side software, the following configurations of the UART are hard-coded, please use *exactly the same* configuration so that the host and target side can communicate properly: 1 start bit, 8 data bits, no parity bit, 1 stop bit, baud rate 19,200. In addition, the slave ID of your device **must** be set to `0x01`.

Exercise 9.1

a) Implement a Modbus slave device that provides 4 holding register (with address 0..3) for reading/writing, each of which is 16 bit wide. The registers should be maintained in the local memory of the slave device and can be accessed via RTU commands from the master device.

Do this task step by step:

- Implement the inter-frame spacing timer and the CRC function.
- Print the command from the master on the LCD to verify that the RTU frame is correctly received and sent.
- Filter out messages that are not addressed to you (according to the slave ID).
- Add the CRC check to determine whether a message is valid.

- Add an error response according to the manual in case the CRC is incorrect.
- Parse the packet and extract the relevant data.
- Add an error response according to the manual in case anything else is wrong.
- Perform the requested action and build and send the reply packet.

The hints section below mentions software that may assist you in development.

b) Make the register with address 4 represent the current temperature value in $\frac{1}{10}$ degrees from the TSic^{TM} temperature sensor. For example, if the temperature is 24.3 °C, let the register contain the value 243. Write operations from the master to that specific holding register should simply be ignored. In addition, show the current temperature value on the LCD as usual.

Hints

- When communicating with the host software, you can use the *Free Serial Port Monitor* software to observe the traffic on the serial port to help you with debugging. The software is installed at `C:/Program Files/HHD Software/Free Serial Port Monitor`. Create a new session for your serial port (e.g., `COM1`) and use the *request view* as your visualizer. Now you can see the traffic on both directions of the respective port.
- To avoid a conflict on the serial port, always keep only the UART receiver or only the transmitter enabled. It is recommended that the transmitter is activated only before sending a response frame and it is deactivated again as soon as the response is successfully sent. For the rest of the time, enable the receiver and disable the transmitter. To keep listening on the serial port, you may want to use the receiver interrupt.

Modbus Master Implementation

We will now extend the Modbus implementation by adding a "master mode". This will allow us to form autonomous networks of multiple microcontrollers. This means that we do not need a host PC any more and our application will really be "embedded".

As this exercise heavily relies on the fact that the previous exercise has been finished, please do not start working on it before you have verified that your slave implementation works and conforms to the specification. Otherwise, you might negatively influence the other nodes on the network.

Exercise 9.2

For the remaining exercises, **set the slave ID of your device to your group number** to avoid conflicts with slave devices of other groups.

Develop an application that can switch between Modbus master/slave mode using a button.

- In slave mode, the device performs the tasks specifies in exercise 9.1.
- In master mode, it periodically generates write and read commands against the registers with address 0..3 of a specific slave and verifies that the register content is as expected. Furthermore, it queries the value of the register with address 4 to retrieve the temperature value.

To verify your master implementation, use a device from another group in slave mode and build a network by attaching the slave's `TX` line to the master's `RX` line and the slave's `RX` line to the master's `TX` line. Attach the cable directly to `PD0` and `PD1`, respectively (see figure 4).

On the LCD of master and slave, show:

- Current mode (Master mode, slave mode)
- Current register values
- Current temperature in °C (on the slave the local temperature and on the master the one retrieved from the slave)

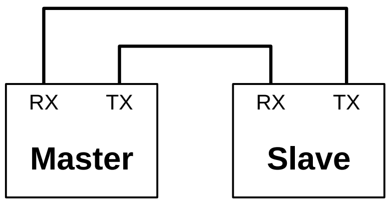


Figure 4: Direct connection of master and slave

Distributed Applications with Multiple Slaves

Exercise 9.3 (optional)

a) Form networks of multiple slave devices and one master device (as shown in figure 1). Make sure that you use a different slave ID for each of the slaves (the group number). Attach the `TX` lines of all slaves to the master’s `RX` line and all slave’s `RX` line to the master’s `TX` line. Visualize the temperature values of all slaves on the master’s LCD.

b) Exchange the master to verify that every node is actually “compliant” to our modified Modbus protocol.