Exercise 9: The Modbus Protocol and Wired Sensor Nodes

Overview

In previous sessions, only a single microcontroller is used to complete a certain task. However, in many real world applications, the workload is distributed into multiple microcontrollers, which interact with each other to achieve a common goal. Such systems are known as distributed systems and those controllers are viewed as nodes in the system structure. Nodes in distributed systems are typically connected via a network and communicate with each other according to predefined protocols. As an example, there are usually more than 100 microcontrollers in a modern car, which communicate using various protocols, such as CAN, MOST, FlexRay, etc. In this session, we will introduce you the widely used Modbus[1] protocol and demonstrate how to program a microcontroller in a distributed system by implementing a simplified version of Modbus.

The Modbus Protocol

Modbus is an openly published serial communication standard. It is now the most commonly available means of connecting industrial electronic devices. The Modbus specification defines the application layer messaging policy, which can be implemented over serial port or Ethernet (TCP/IP). Most Modbus devices communicate over a serial port. For serial connections, two variants exist, namely the RTU (Remote Terminal Unit) mode and the 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.

Modbus is a master-slave based protocol. In a Modbus network, an unique address is assigned to each device intended to communicate. Only nodes assigned as *master* can initiate a command. A Modbus command contains the Modbus address of the device it is intended for. Only the intended device will act on the command, even though other devices might receive it.

The Modbus protocol defines a simple protocol data unit (PDU) independent of the underlying communication layers. A Modbus PDU consists of a function code field and data field. A valid function code is a one-byte value between 1 and 255. The data field contains additional information needed by a specific function code. In this exercise, we will implement a simplified version of Modbus, which 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 specification.

Receiving an RTU frame

The maximum length of a RTU frame is 256 bytes. In RTU mode of Modbus, a command PDU is equipped with a header field, which contains the address of the slave device intended to response the command, and a tailer field, which contains the CRC of the frame for error detection (figure

[1] http://www.modbus.org/

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

Figure 1: RTU Message Frame

1). The device address is coded in one byte, in which address 0 is reserved as broadcast address. A slave device should have an unique address between 1 and 247. The master device does not have a specific address. The CRC field contains two bytes, which are the expected check sum of the frame calculated by the sender. The receiver should compute the CRC of the received frame and compare it with the expected value. An error results if the two values do not match. The detailed information about how the CRC is calculated can be found in the specification.

The starting and ending points of a RTU frame are identified by the so-called $t_{3.5}$ interval (figure 2). 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 ending of the current frame. CRC check is done after receiving a frame to detect erroneous frames. The RTU frame must be transmitted in a continuous byte stream. If a silent interval of more than 1.5 byte time ($t_{1.5}$) occurs between two successive bytes, the frame should be considered as incomplete and discarded. Thus, we need two separate timers, one for $t_{1.5}$ and another one for $t_{3.5}$. For baud rates greater than 19,200 bps, fixed values for the 2 timers should be used: it is recommended to use a value of 750 μs for the inter-character time-out ($t_{1.5}$) and a value of 1.750 ms for inter-frame delay ($t_{3.5}$).

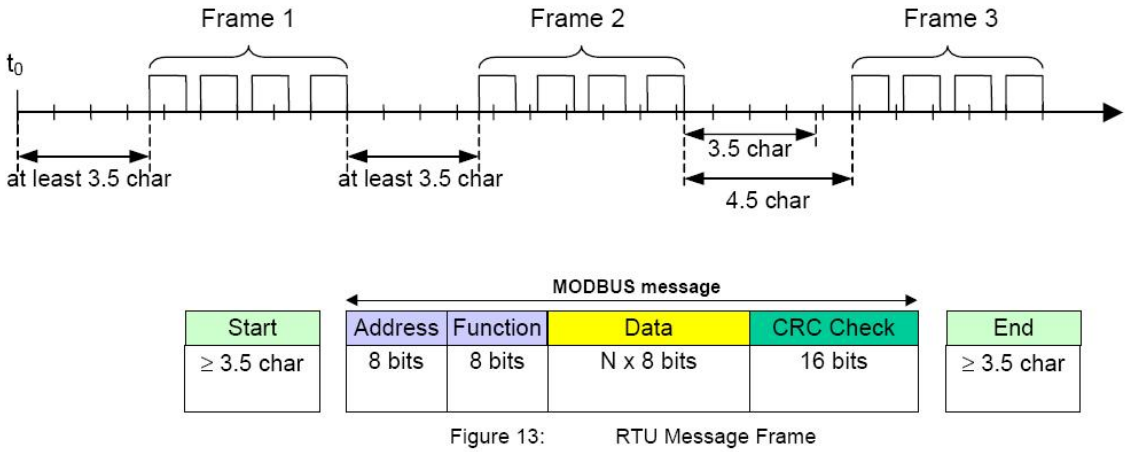

Figure 2: Separation of RTU frames

Modbus Holding Registers

The data model of Modbus consists of four primary types as listed in figure 3. There are separate function codes for accessing different data types. In this session, we will only use the holding register as an example.

In the pure holding register system, the slave devices can be viewed as external memories of the master device. The register contents are store in the respective slave devices labeled with addresses. The value of a holding register can be updated by sending a write single register command (`0x06`) or read by sending a read holding register command (`0x03`). Read the specification for details of the two commands.

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 3: Modbus Data Model

Exercise 9.1

Browse through the Modbus protocol specification to find out the following information.

a) What is the structure of a general Modbus application data unit (ADU) ? Which part of the frame is the protocol data unit (PDU)?

b) What is the meaning of the following read/write function codes: `0x01`, `0x02`, `0x03`, `0x04`, `0x05`, `0x06`? Group them according to figure 3.

c) Summarize for which kind of data the Modbus protocol is optimized and why it was designed that way regarding its primary field of use.

Prerequisites

In this exercise, we will use the RTU mode of the Modbus protocol over a serial line. This mode requires that we use *cyclic redundancy checks* (CRC) to secure the data. Read appendix 6.2.2 on page 39–43 of the "Modbus over Serial Line" specification to find out how Modbus CRC calculation basically works.

Exercise 9.2

- Develop a function with the signature `uint16_t modbus_crc16(uint8_t* buffer, uint16_t n)` that calculates the CRC for arbitrary data in the `buffer` with length `n`. You can use the code shown in appendix 6.2.2 for this purpose.

  You might want to verify that your function yields the following result:

  - Input: `0x01 0x06 0x00 0x0E 0x04 0xD2`
  - Output: `0x94 0x6A`

Note

Since ATmega8515 has quite limited memory space, it might be advantageous to use a function-based implementation rather than the lookup-table based approach as described in the specification. If you run out of memory space, ask the tutors to provide you with a CRC calculation function that does not use lookup tables (however that implementation is of course slower, but this does not cause problems in our usage scenario).

Modbus Slave Implementation

Exercise 9.3

a) First, attach the LC display to the slave 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).

b) Implement a Modbus device that provides 16 holding register for reading/writing, each of which is 16 bits wide. Those registers are maintained in the local memory of the slave device and can be accessed via an RTU command from the master device. Do this task step by step, e.g., first echo the command from the master to verify that the RTU frame can be correctly received and sent. The hints part in the following introduces some software that may assist you with development.

c) Make the register with index 15 represent the current temperature value in $\frac{1}{10}$ degrees from the TSic temperature sensor. For example, if the temperature is 21.3 °C, let the register contain the value 213. Write operations from the master to that specific holding register should simply be ignored. In addition, show the current temperature value on the LCD.

Hints

- To communicate the your slave device, we 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. In addition, the slave ID of your device **must** be set to `0x01`.
  - 1 start bit
  - 8 data bits
  - no parity bit
  - 1 stop bit
  - Baud rate 19,200
- 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 conflict in the serial port, always keep only the 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.