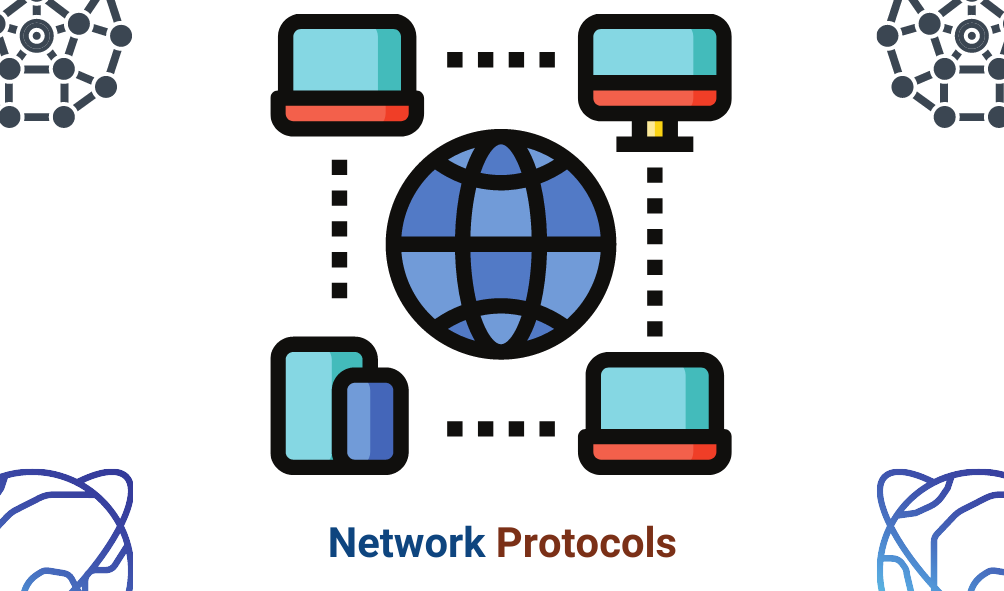
Assignment Point To Point Communication

By July Trendafilov

St. nr.: 4674693



Contents

[Version Table 2](#_Toc123304623)

[Short Introduction 3](#_Toc123304624)

[Transmitting and receiving 3](#_Toc123304625)

[Application 3](#_Toc123304626)

[Implementation 6](#_Toc123304627)

[Transmitting 7](#_Toc123304628)

[Receiving 9](#_Toc123304629)

[Buffering 11](#_Toc123304630)

[References 13](#_Toc123304631)

# Version Table

|  |  |
| --- | --- |
| Version 1 – 02.12.2022 | Document + code submission |
| Version 2 – 30.12.2022 | Added available and buffer functionality to assignment |

# Short Introduction

In this assignment we must make a module for UART without using libraries. The protocol has two wires – TX and RX which are for receiving and transmitting messages. The protocol has a couple of options for the messages:

* Number of data bits
* Parity bit (optional)
* Number of stop bits

This protocol is great for sending data between boards or for debugging. Both systems that are communicating must have the same baud rate.

# Transmitting and receiving

To transmit data, we must time every bit in the middle so that we have some error resistance. We have a start bit which is LOW and the IDLE state of the UART is HIGH. Every time we get a start bit we must time the next bit in the middle. To do so we can use this simple formula:

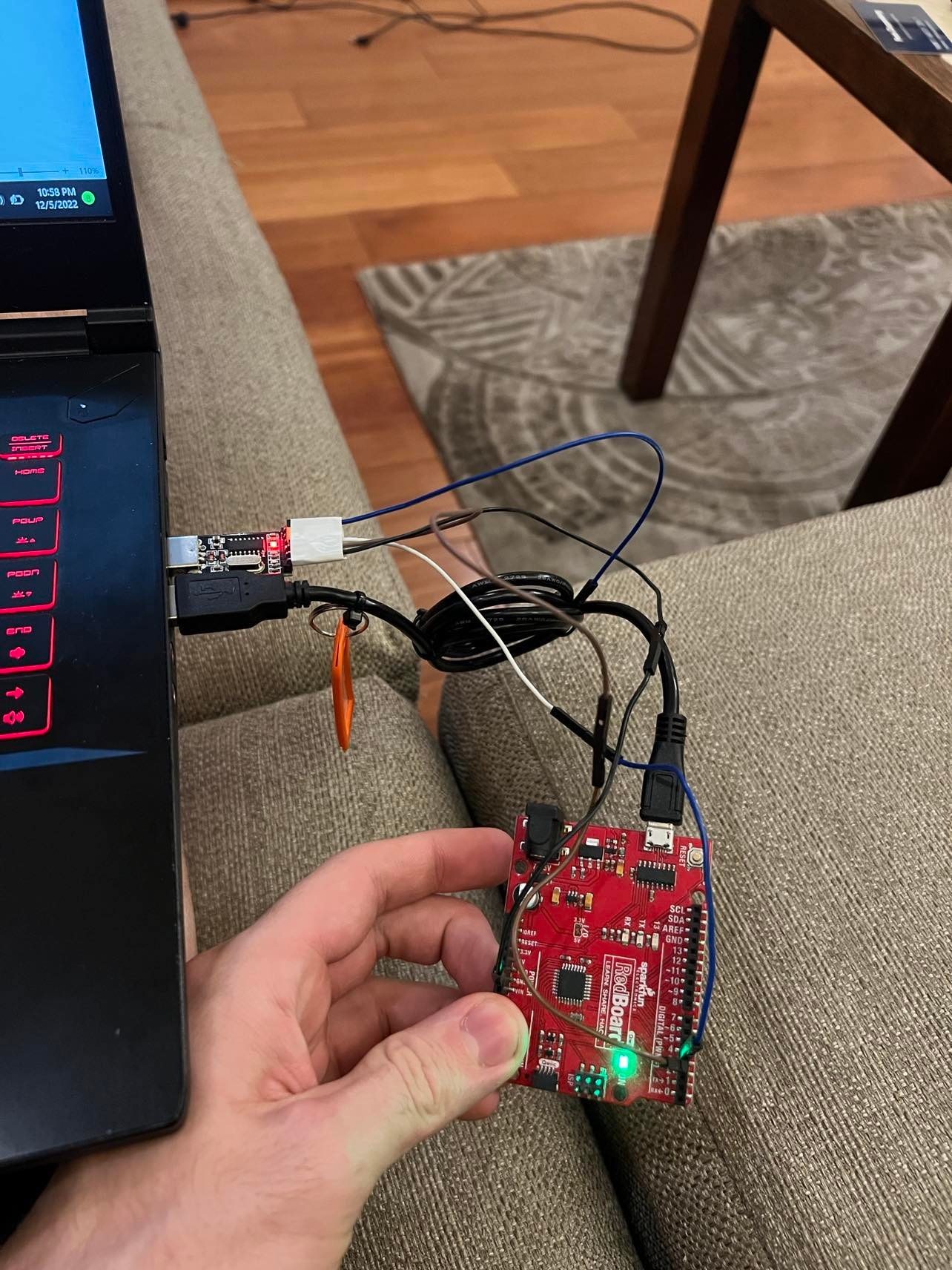
* (1 / baudRate) \* 1000 = how much time does it take for one bit to be sent in Milliseconds.

To get the middle of the bit we must divide the result of the top calculation by 2. The easiest way is to time the middle of the first bit and then just time exactly 1 bit time, so we get aways the middle. After we read all data bits we have a parity bit and after that the stop bits.

If we don’t have a parity bit we sent everything and expect it to be true. If we specify a parity bit it becomes different. The parity bit calculate how many ones we have in the data bits and then if the count is not the same as the parity, the message is corrupted. There are even parity and odd parity where one is true if we have an odd amount of zero and the other is one and vice-versa.

# Application

In our LAB we have one Arduino board and a UART module to interface the USB of the laptop:



Then we have a library that we include to use the UART module called MyUART:

Graphical user interface, text, application

Description automatically generated

This is a simple UART module that has send function for numbers and text and receive function for bits:

Text

Description automatically generated

We keep track of the number of data nits, stop bits and possibly the parity bit. The read function has a timeout if we don’t read something. In this application the timeout is set to 0xFFFFFFFF.

In this application we have a debugger to debug all ports of the Arduino. If we receive ‘A’ that means we must give the value of each analog pin. If ‘D’ we give the value of all digital pins. If the module is implemented correctly the serial monitor looks as:

Text, letter

Description automatically generated

The current test is with setting (7n2) where we have 7 data bits, no parity and 2 stop bits. The pins we use are D3 and D4 on PORT D of the board.

# Implementation

Diagram

Description automatically generated

## Transmitting

Text

Description automatically generated

As explained earlier first we count the count of the 1s in a character and when we have calculated if the count is even we start transmitting the message starting from LSB to MSB.

Text

Description automatically generated

Then if we have a parity which in our case is as such:

* No parity = 0
* Even parity = 1
* Odd parity = 2

Then we send the correct parity bit.

Text

Description automatically generated

Lastly, we have the stop bits where they can be either 1 or 2 but the implementation allows more for everything. We send the count of stop bits.

## Receiving

The receiving part is more special. It is implemented as:

Text

Description automatically generated with medium confidence

First we have the timeout feature of the reading.

A screenshot of a computer

Description automatically generated with medium confidence

Then we have the actual message where we read all data bits and the parity bit which we use if we need to.

Text

Description automatically generated

After that we calculate the parity of the received buffer as we count all 1s in the data bits and then extract it is a variable.

Text

Description automatically generated

Then we check if the parity is set and check if the message is correct. If not a 0 is returned.

Text

Description automatically generated

Lastly, when sending the read data, we must remove the parity bit from the message which is at the MSB position.

## Buffering

As part of the feedback for adding a buffer to the UART where if bytes are available, we read the first value in the buffer.

Due to absence of a queue in Arduino I had to make my own queue. It is a simple queue where we have only push(), pop() and size() operations. The Node structure is private to the class because we don’t want the user to use it.

Text

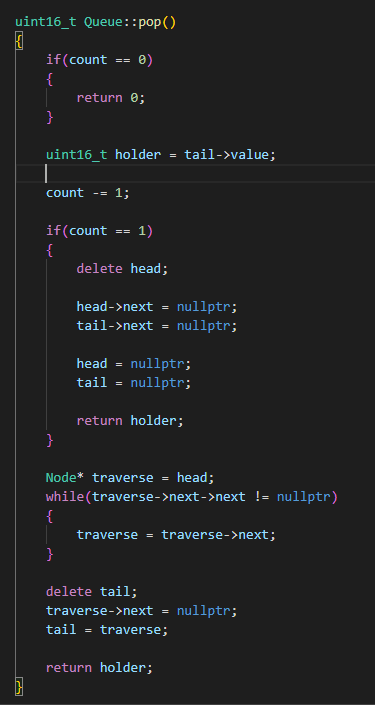
Description automatically generated

The push function is simple. When we call push, we place the new value at the end of the queue.

Text

Description automatically generated

The pop function is somewhat harder because we must deallocate the memory and extract the first value in the queue.



The size function just returns the count of elements. It is used for the available function MyUART.

Text

Description automatically generated

With the newly added library for queue, we now have the following structure:

Graphical user interface, text

Description automatically generated

# References

Dobbelsteen, J. (n.d.). *Github*. Retrieved from https://git.fhict.nl/technology/t-sem3-db/-/tree/master/communication/Toolbox

*StackOverflow*. (n.d.). Retrieved from https://stackoverflow.com

*Wikipedia*. (n.d.). Retrieved from https://www.wikipedia.org