EAST WEST UNIVERSITY

UART Communication Stress Test with NodeMCU ESP8266

Course Title: *Internet of Things*

Course Code: *CSE 406*

Section: *01*

Submitted by

Name: *Shairin Akter Hashi*

ID: *2022-2-60-102*

Date: *13/07/2025*

Submitted to

Dr. Raihan Ul Islam (DRUI)

Associate Professor, Department of Computer Science and Engineering

East West University

Other Group Members

Name	*ID*
Nushrat Jaben Aurnima	*2022-2-60-146*
Zihad khan	*2022-2-60-107*
Shahrukh Hossain Shihab	*2022-1-60-372*

Table of Contents

Introduction 2

Methodology 2

Hardware Setup 2

Software Setup 3

Arduino IDE Configuration 3

CoolTerm Setup and Logging 4

Baud Rate Synchronization 4

Results 5

Data Collection 5

Observations 5

Analysis 6

Throughput 6

Transfer Speed 6

Error Rate 6

Best Configuration 7

Challenges 7

Conclusion 7

Refences 8

Introduction

UART (Universal Asynchronus receiver transmitter) is a widely used communication protocol that allows the asynchronous serial data exchange between TX (transmitter) and RX (obtained) PIN use. It works without a clock signal and depends on the predetermined storage speed to sync data transfer and reception. In this laboratory, we targeted to evaluate the performance of UART communication between two **Nodemcu Esp8266 board**.

The purpose was to measure the **throughput** (the amount of data transferred in byte/seconds), **transfer rate** (message/second) and **error rate** (percentage of unsuccessful/contaminated messages). We performed a stress test using different parameters: **Baud speeds** (9600, 38400, 115200), **message size** (10, 50, 100 byte) and **transfer interval** (0ms, 10ms, 100ms). The boards were connected using expansion boards and a breadboard for wiring.

Methodology

Hardware Setup

For this lab, two NodeMCU ESP8266 boards were connected using jumper wires and a breadboard. The connections were made as follows:

- **NodeMCU 1 D5 (TX) → NodeMCU 2 D6 (RX)**
- **NodeMCU 2 D5 (TX) → NodeMCU 1 D6 (RX)**
- **GND pins** of both boards were connected through the shared ground on the breadboard.

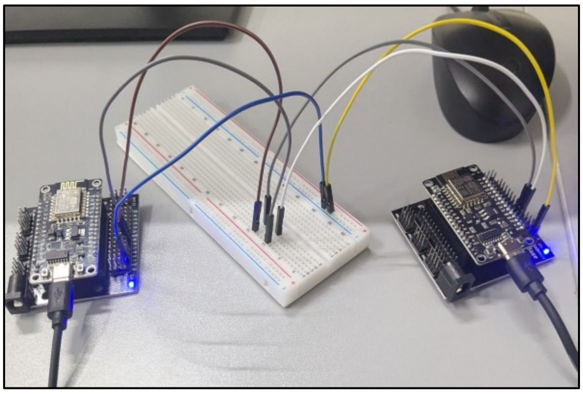

Figure 1: Hardware Setup

Software Setup

Software configurations include **Nodemcu ESP8266** Board Both programming and establishment of serial communication monitoring using **Arduino IDE** and **CoolTerm**.

Arduino IDE Configuration

The Arduino IDE was used to program both NodeMCU 1 (Master) and NodeMCU 2 (Slave). To enable ESP8266 board support, we added the following URL to the **"Additional Board Manager URLs"** field in the **Preferences** window: [*http://arduino.esp8266.com/stable/package_esp8266com_index.json*](http://arduino.esp8266.com/stable/package_esp8266com_index.json)

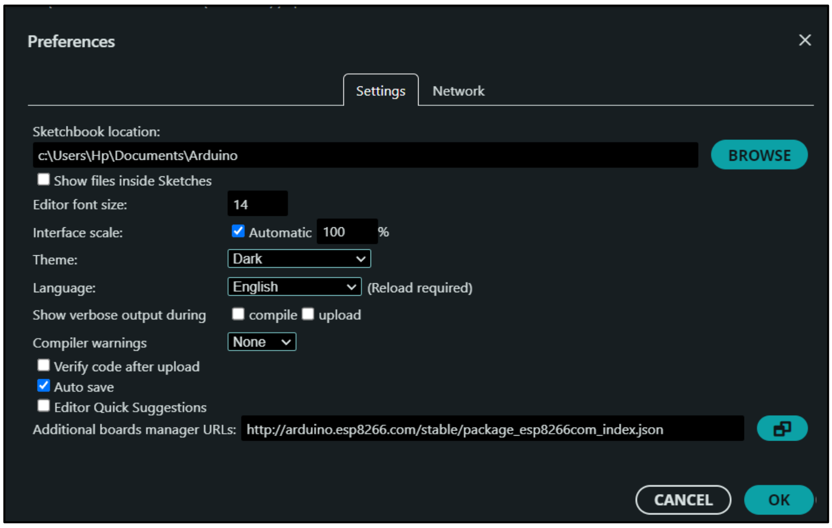

Figure 2: Additional Board Manager URLs

After that, we installed the **ESP8266 Board Package (version 3.1.2)** via the **Boards Manager**. Before running the main test, some basic demo sketches were uploaded and ran to verify that the board drivers, serial communication, and wiring were working correctly.

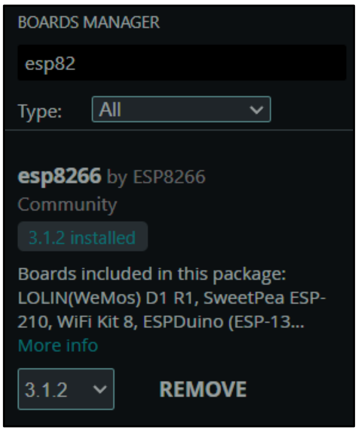

Figure 3: Installation

The **Arduino IDE** was used to upload separate sketches to each board:

- Master_Final.ino was uploaded to **NodeMCU 1 (Master)** connected to **COM7**.
- Slave.ino was uploaded to **NodeMCU 2 (Slave)** connected to **COM6**.

CoolTerm Setup and Logging

To capture and log real-time communication output from each NodeMCU, **CoolTerm** was used which is a serial terminal that allows data capture to text files. Each board was connected to a separate CoolTerm session:

- **NodeMCU 1 (Master) – COM7**
  - CoolTerm was configured with:
    - **Baud rate:** 115200
    - **Data bits:** 8
    - **Parity:** None
    - **Stop bits:** 1
  - Captured output was saved to: `nodemcu1_output.txt`
- **NodeMCU 2 (Slave) – COM6**
  - CoolTerm was configured with the same settings.
  - Captured output was saved to: `nodemcu2_output.txt`

To start logging:

1. Navigate **Connection** > **Capture to Text File** > **Start**, then select the destination .txt file, and then click **Connect** (green plug icon).
2. Once testing is completed, click **Connection** > **Capture to Text File** > **Stop**, followed by **Disconnect**.

Baud Rate Synchronization

Both devices were programmed with the same se rate (BAUD: 115200). This mechanism helped maintain continuous communication settings and reduced the possibility of error passing or data corruption due to deviations with body rate.

Results

Data Collection

The table below summarizes the UART communication performance between **NodeMCU 1** (Master) **and NodeMCU 2** (Slave), using different message sizes and intervals at **three** baud rates (9600, 38400, 115200). The metrics are calculated based on values parsed from the captured logs (`nodemcu1_output.txt` and `nodemcu2_output.txt`).

Baud Rate	Message Size	Interval	Throughput (bytes/s)	Message Rate (msg/s)	Error Rate (%)
9600	10 bytes	0 ms	478.00	48.90	0.00%
9600	10 bytes	10 ms	304.00	31.50	0.63%
9600	10 bytes	100 ms	75.50	8.50	0.00%
9600	50 bytes	0 ms	484.10	9.90	0.00%
9600	50 bytes	10 ms	440.00	9.00	0.00%
9600	50 bytes	100 ms	244.00	5.00	0.00%
9600	100 bytes	0 ms	484.10	4.90	0.00%
9600	100 bytes	10 ms	464.30	4.70	0.00%
9600	100 bytes	100 ms	325.70	3.30	0.00%
38400	10 bytes	0 ms	17.00	2.00	100.00%
115200	10 bytes	0 ms	17.00	2.00	100.00%

Observations

- **Throughput increases** with larger message sizes and shorter intervals.
- **Message rate decreases** with larger message sizes, as expected.
- Both **38400** and **115200** baud rates failed entirely, returning **100% error rates**.

- **Timeout errors** were frequently seen at higher baud rates.
- **Mismatch error** was seen at 9600 baud rate 10ms 10-byte size.
- At **9600 baud**, communication was stable with **0% errors** across most tests.

Analysis

Throughput

The throughput was successful for large message size. For example, at 9600 baud rates, both 50 byte and 100 byte messages are more than 440 bytes/s. However, this advantage when the delay interval was extended to 100 ms. The limitation arises from SoftwareSerial's buffer size and the timing accuracy on ESP8266 when operating at higher data ratesIn 38400 and 115200 booths, the flow rapidly fell to 17 bytes/s due to transmission failure.

Transfer Speed

Short intervals (0ms) enabled higher message rates, reaching nearly **49 msg/s** with 10-byte messages. As message sizes grew, transmission took longer, lowering the rate to around **3.3 msg/s** for 100-byte messages at 100ms intervals.

Error Rate

The error rate analysis showed that the UART communication at **9600 baud rate** was largely stable, with **0%** error in all combinations of message size and interval, except from a single case (10 byte, 10 ms) where a negligible **0.63%** error rate was observed. This small error can be attributed to temporary time problems in software serial buffers. However, it did not repeat continuously, which indicates a transient issue.

On the other hand, communication at higher baud rates (**38400 and 115200)** continuously resulted in **100% error rates**. This issue arises because of the limitations of the **SoftwareSerial library** on the ESP8266. The hardware has **two** UARTs: **UART0 (TX/RX)** typically reserved for USB serial output, and **UART1 (TX only)**. If we connect with **UART0**, it may interfere with USB communication. Therefore, code might not be uploaded correctly on devices. That's why **SoftwareSerial** was used to simulate UART communication on GPIO pins (**D5** and **D6**).

However, **SoftwareSerial** Library over **9600** stores on ESP8266 are not reliable, as there is a lack of control at hardware level and depends heavily on **interrupt-driven timing**. Unlike hardware UARTs that use **FIFO** buffers to manage data effectively, circumvent software such buffering, which makes it prone to defer and disrupt the data, especially at high data rates and dense intervals (**0 ms**).

The **mismatch errors** didn't happen often. It is likely caused by sync problems at the start or missed bytes. Since they were rare and didn't affect results much, no fix was added.

Best Configuration

The most reliable performance was observed with:

- **Baud Rate:** 9600
- **Message Size:** 50 bytes
- **Interval:** 10 ms
  This configuration yielded **440 bytes/s** throughput, **9 msg/s**, and **0% error** offering a balanced trade-off between speed and stability.

Challenges

Finding specific and accurate documents for the UART capabilities of ESP8266; especially software limitations were one of the major problems we encountered. The official technical manual did not clearly explain the lack of bod rate, which made it difficult to determine whether there was expected behavior of failures seen at high baud rate. Additionally, coordinating between master and slave equipment requires necessary testing and manual verification, as ESP8266 did not always work at the intended baud rate when using SoftwareSerial library. Understanding the difference between hardware UART and imitating people was important to explain and solve these issues.

Conclusion

This experiment demonstrated that the UART on ESP8266 is stable in only **960**0 baud using communication **SoftwareSerial** library. At this speed, the message sizes of **50** and **100** bytes performed the best with **484 bytes/s** throughput and **0%** error. However, in **38400** and **115200** baud rates, all tests failed with **100%** error rates, highlighting the boundaries of software-based serial communication on ESP8266. Although UART is generally a reliable method for data transfer, the time sensitivity of the software and the lack of hardware buffering make it unsuitable for high-speed communication. Therefore, to improve performance, future implementation must use hardware UART, avoid software, or consider alternative communication protocols such as SPI or I2C for high data rates.

Refences

- ***https://forum.arduino.cc/t/serial-monitor-and-esp8266/480428/6***
- ***https://www.espressif.com/sites/default/files/documentation/esp8266-technical_reference_en.pdf***