UART Communication Stress Test with NodeMCU ESP8266

Your Name: Md Ashik
Student ID: 2020-3-60-032
Course Name: CSE406
Date: July 25, 2025

1. Introduction

UART (Universal Asynchronous Receiver-Transmitter) is a fundamental asynchronous serial communication protocol widely used in embedded systems for data exchange between microcontrollers or with computers. It relies on dedicated transmit (TX) and receive (RX) pins and a pre-agreed baud rate to ensure successful data transfer. The objective of this lab was to conduct a stress test on UART communication between two NodeMCU ESP8266 boards, connected via expansion boards and a breadboard, to quantify key performance metrics. Specifically, the lab aimed to measure throughput (bytes per second), transfer speed (messages per second), and error rate (percentage of failed or corrupted messages) under varying test parameters. These parameters included three baud rates (9600, 38400, 115200), three message sizes (10, 50, 100 bytes), and three transmission intervals (0ms, 10ms, 100ms).

2. Methodology

Hardware Setup:
The experimental setup involved two NodeMCU ESP8266 boards. NodeMCU 1 was designated as the Master, and NodeMCU 2 as the Slave. Both boards were mounted on expansion boards and connected via a breadboard for stable wiring. The UART communication link was established using SoftwareSerial on specific GPIO pins: NodeMCU 1's D5 (TX) was connected to NodeMCU 2's D6 (RX), and NodeMCU 1's D6 (RX) was connected to NodeMCU 2's D5 (TX). A common ground (GND) connection was established between both NodeMCUs via the breadboard to ensure a stable reference voltage. Initially, challenges were encountered with ensuring reliable connections on the breadboard, which were resolved by verifying pin assignments and ensuring firm wire insertions.
Software Setup:
The Arduino IDE was used to program both NodeMCU boards. NodeMCU 1 was programmed with the NodeMCU1_Master_StressTest.ino sketch, responsible for initiating communication, sending messages with sequence numbers, verifying responses, and calculating performance metrics. NodeMCU 2 was programmed with the NodeMCU2_Slave_StressTest.ino sketch, which primarily echoed back received messages to the Master. Output from both boards was monitored using the Arduino IDE's built-in Serial Monitor, with the intention of logging to files

(e.g., nodemcu1_output.txt, nodemcu2_output.txt) for later analysis. A critical aspect of the software setup was the baud rate synchronization mechanism: the Master sent a "BAUD:[baudrate]" command to the Slave via SoftwareSerial, prompting the Slave to dynamically switch its SoftwareSerial baud rate to match the Master's current test speed, ensuring consistent communication.

3. Results

Data Collection:
The stress test was conducted for each combination of baud rate, message size, and interval, with each test permutation running for 10 seconds. The results, including throughput, message rate, and error rate, were logged to the Master NodeMCU's Serial Monitor. The following table summarizes the key findings:

Baud Rate	**Message Size (bytes)**	**Interval (ms)**	**Messages Sent**	**Messages Received**	**Errors**	**Error Rate (%)**	**Throughput (bytes/second)**	**Message Rate (messages/second)**
9600	10	0	489	489	0	0.00	478.00	48.90
9600	10	10	331	331	0	0.00	320.00	33.10
9600	10	100	85	85	0	0.00	75.50	8.50
9600	50	0	99	99	0	0.00	484.10	9.90
9600	50	10	90	90	0	0.00	440.00	9.00
9600	50	100	50	50	0	0.00	244.00	5.00
9600	100	0	49	49	0	0.00	484.10	4.90
9600	100	10	47	47	0	0.00	464.30	4.70
9600	100	100	33	33	0	0.00	325.70	3.30
38400	10	0	1804	1804	0	0.00	1873.40	180.40
38400	10	10	655	655	0	0.00	644.00	65.50

38400	10	100	96	96	0	0.00	85.40	9.60
38400	50	0	384	384	0	0.00	1909.00	38.40
38400	50	10	278	278	0	0.00	1379.00	27.80
38400	50	100	80	80	0	0.00	391.00	8.00
38400	100	0	193	193	0	0.00	1919.00	19.30
38400	100	10	162	162	0	0.00	1609.00	16.20
38400	100	100	66	66	0	0.00	652.40	6.60

For 115200 baud, reliable data could not be consistently obtained due to a very high error rate, often resulting in timeouts or corrupted messages. The values for 115200 baud are indicative of severe communication issues rather than quantifiable performance.

Observations:

- **Error Rate:** A consistent 0.00% error rate was observed across all test cases for both 9600 baud and 38400 baud. This indicates highly reliable communication at these speeds under the given test conditions. However, at 115200 baud, the error rate became extremely high, leading to unreliable data transfer, characterized by frequent "Mismatch error" and "Timeout error" messages.
- **Throughput and Message Rate vs. Interval:** For a given baud rate and message size, both throughput (bytes/second) and message rate (messages/second) increased significantly as the transmission interval decreased (i.e., messages were sent more frequently). This is a direct consequence of sending more data within the 10-second test duration.
- **Throughput vs. Message Size:** At a fixed baud rate and interval, increasing the message size generally led to higher throughput (bytes/second) but a lower message rate (messages/second). This is because larger messages carry more data per transmission, but take longer to send, thus reducing the number of individual messages transmitted per second.

- **Baud Rate Impact:** As expected, increasing the baud rate from 9600 to 38400 significantly increased both throughput and message rate for comparable message sizes and intervals, demonstrating the benefit of higher transmission speeds when reliable.
- **Mismatch Errors at 115200 Baud:** During initial tests at 115200 baud, frequent "Mismatch error" messages were observed, such as "Expected '4:DXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX', Got '4:DXXXXX\XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'". These errors manifested as truncated messages or messages containing corrupted characters (e.g., \X). This indicated that the data was not being received completely or accurately.

4. Analysis

Throughput:
The analysis of throughput clearly demonstrates its direct correlation with both baud rate and message size. Higher baud rates (e.g., 38400 vs. 9600) allow for more bits to be transmitted per second, directly increasing the potential throughput. Similarly, larger message sizes contribute to higher throughput by amortizing the fixed overhead of starting and ending a message over a greater number of data bytes. For instance, sending one 100-byte message is more efficient in terms of bytes per second than sending ten 10-byte messages, as the start/stop bits and processing overhead occur once per message. The limitations of SoftwareSerial, particularly at 115200 baud, became evident as throughput became unreliable due to excessive errors, highlighting the software-emulated nature's inability to keep up with the rapid bit transitions. Potential breadboard noise, though not explicitly causing errors at lower speeds, could also contribute to signal degradation at higher frequencies.
Transfer Speed:
Message rate, or transfer speed in terms of messages per second, showed an inverse relationship with message size. Smaller messages resulted in higher message rates because each message takes less time to transmit, allowing more individual messages to be sent within the same time frame. Conversely, shorter intervals (e.g., 0ms) drastically increased the message rate by minimizing the idle time between transmissions. The trade-off between speed and reliability was evident: while 0ms intervals offered the highest potential message rates, they also pushed the limits of SoftwareSerial, leading to significant errors at 115200 baud.
Error Rate:
The most critical finding was the stark difference in error rates across baud rates. At 9600 baud and 38400 baud, the error rate was consistently 0.00%, demonstrating robust and reliable communication. This indicates that SoftwareSerial on the ESP8266 is well-suited for

these speeds in a controlled environment. However, at 115200 baud, the error rate became unacceptably high, often exceeding 50% or resulting in a complete communication breakdown. The primary causes of these errors are attributed to:

- **SoftwareSerial Limitations:** At 115200 baud, the CPU overhead required for SoftwareSerial's bit-banging operations becomes too high. The ESP8266's CPU, even at 80MHz/160MHz, can be interrupted by other background processes (e.g., Wi-Fi stack, internal timers), causing it to miss critical timing windows for sampling or generating bits.
- **Buffer Overflow:** With continuous data streams at high speeds (especially at 0ms interval), the SoftwareSerial receive buffer on the slave might overflow if the loop() function cannot process incoming bytes fast enough, leading to data loss or corruption.
- **Signal Integrity:** While less likely to be the sole cause, even minor electrical noise or impedance mismatches on the breadboard and jumper wires can become significant at higher frequencies, causing bits to be misinterpreted.

The observed "Mismatch error" messages, particularly those showing \X characters or truncated strings (e.g., "Got '31:DXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"), are direct evidence of data corruption and incomplete reception. The trim() function in the code was crucial in mitigating potential mismatch errors caused by hidden carriage return (\r) characters that readStringUntil('\n') might leave in the buffer, ensuring that string comparisons were accurate for correctly received data.

Best Configuration:
Based on the experimental results, the optimal configuration for reliable and high-performance UART communication using SoftwareSerial on the NodeMCU ESP8266 boards, balancing speed and error rate, is 38400 baud. For this baud rate, a message size of 100 bytes with an interval of 0ms achieved the highest throughput (1919.00 bytes/second) with a 0.00% error rate. This configuration maximized data transfer without sacrificing reliability.
Challenges:
Several challenges were encountered during the lab. Initially, ensuring correct SoftwareSerial pin connections (TX to RX, RX to TX) and a shared ground was crucial. A significant challenge was the implementation of the baud rate synchronization mechanism, where the Master sends a "BAUD:" command to the Slave. The initial slave code incorrectly attempted to read this command from the USB Serial (Serial.available()) instead of the SoftwareSerial (mySerial.available()), leading to a failure in baud rate switching. This was resolved by correcting the slave's code to listen for the baud command on its SoftwareSerial port. The most persistent challenge was achieving reliable communication at 115200 baud with SoftwareSerial, which ultimately proved infeasible due to the inherent limitations of software-emulated serial communication at such high speeds.

5. Conclusion

This lab successfully demonstrated and quantified the performance of UART communication between two NodeMCU ESP8266 boards under various conditions. Key findings indicate that SoftwareSerial provides highly reliable (0.00% error rate) communication at baud rates of 9600 and 38400. At 38400 baud, a 100-byte message size with a 0ms interval yielded the highest throughput of 1919.00 bytes/second, showcasing the maximum practical data transfer rate for this setup without errors. However, the limitations of SoftwareSerial became critically apparent at 115200 baud, where communication suffered from severe data corruption and timeouts, rendering it unsuitable for reliable data transfer. This highlights that while UART is a robust protocol for data transfer, the choice between hardware and software implementations significantly impacts performance and reliability at higher speeds. For future improvements, utilizing the ESP8266's dedicated hardware UART (UART0 or UART1) is strongly recommended for applications requiring 115200 baud or higher. Additionally, improving wiring quality and considering error correction protocols could further enhance robustness in noisy environments.