CMPE121L

Xingyu Zhou

Anujan Varma

2019/5/4

Lab3 Report

**Introduction:**

The purpose of this lab was to taught us the basics of the UART and how to use it. To start we needed to read through the datasheet and then set up the UART by following the instructions of the lab manual. The lab was consisted of 5 parts and 4 of them were focused on transferring data from the Tx end to Rx using different methods. The first part of the lab was to transmit/receive the data without using interrupts. The second part was basically the same as part 1, but with Rx/Tx interrupts enabled. The third part was to observe the UART with/without the hardware flow control and the fourth part was transferring/receiving data between two UARTs. The last part was to figure out at what frequency can the UART transfer/receive data without giving errors.

**Part 1: Transmit/Receive Based on Polling**

A screenshot of a cell phone

Description generated with very high confidence

**Figure 1: Part1 Top Design**

In this part, Rx/Tx interrupts were disabled. The first thing to do was to transmit the byte value of 0xa5 continuously and identify the start, stop, parity and data bytes on the waveform. This will be provided in figure 2.

A picture containing text, shoji

Description generated with very high confidence

**Figure 2: waveform with bytes indicated**

Then we needed to set up two byte arrays. One of them was increasing from 0 to 255 and repeating while the other was all zeros. It was the same as Lab2 so I just used that code with minor modifications. Then for the data transmission part, the first thing I did was to check if the FIFO is full. The code would stay in a while loop if the FIFO was full until it has room. Doing this prevented the data overflow. When the FIFO has room, the transmission shall begin. After that, I read the data received in Rx and stored in my receiving array. A timer was added to calculated the time taken to transfer all the data. After I ran the program, I got 4080 mismatches, which was almost everything. Then I realized I was reading the data before the transmission was done. I added a while loop saying while the transmission was not done just do nothing, then all the mismatches just went away.

**Part 1 Question:**

|  |  |  |
| --- | --- | --- |
| Transfer Rate | Calculated Time | Measured Time |
| 3490Bytes/Sec | 1.17s |  |

**Table1: Calculated Time VS Measured Time (4096 bytes to transfer)**

As you can see in the table, the calculated time and the measure time did not agree with each other. My assumption is that I probably calculated the time using the wrong method ((max period – counter)/clock). Say the measured time is slower than the calculated time, that is due to various delays, for example, the delays in the external wire connecting the Rx and Tx. One way to fix it might be switching to internal wire (connect them in top design) instead of using external wire.

**Part 2:Transmit/ Receive Using Interrupts**

The set up process of was the same as part 1, but we needed to turn on Rx and Tx interrupts for the data transmission. I had my Rx interrupt triggered on byte received and Tx interrupt triggered on FIFO empty. I had two flags that were set to FALSE at the beginning. When the ISR was triggered, those flags would be set to TRUE indicating the FIFO was empty/a byte was received. Two counters were also added inside my ISR to keep track of the times entered. Since the FIFO was initially empty, my code would enter the ISR immediately, set the flag to true and start filling the transmit array. Then the code would check if there was anything in the receiver FIFO and added to my receiver array. In order to optimize the times entering my Tx ISR, I had it set to transmit 4 bytes at a time. It then lowers the time entering the Tx ISR to 1024 times.

**A screenshot of a cell phone

Description generated with high confidence**

**Figure 3: Part2 Top design**

**Part 3: Hardware Flow Control**

This part of the lab was similar to part 2, but we were not allowed to use the Rx interrupt. Instead we would need to add a timer to our top design that generated an interrupt at around 0.5 millisecond. My Tx ISR was triggered on FIFO empty, and in the ISR, the program would transmit four bytes at a time.(I just move the code from part2 into the ISR). Then in the timer ISR, mt code would check for Rx status and store the data into receiver array if there was anything in the Rx FIFO. Then we needed to enable the hardware control of the UART and connect RTS\_N and CTS\_N together, run the experiment again to see if there were any differences.

**Part3 Questions:**

1. When the timer period was set to 1.1 milliseconds, I noticed that I was getting both Rx errors as well as mismatches. This was due to how often the interrupt was generated. The data transmission was fast and 1.1ms per interrupt was not fast enough. As a result, some of the data was lost and thus the mismatches appeared. I guess the maximum period should be around 0.5 – 0.7 ms.
2. The errors went away when the hardware flow control was enabled. This was due to the fact that the transmitter would only send out a byte when the clear to send signal is received. There would be no data overflow or other errors under this situation.

**A screenshot of a social media post

Description generated with very high confidence**

**Figure 4: Part3 Top Design**

**Part 4: Data Transfer between Two UARTs**

In this part of the lab we were told to simultaneously transfer and receive data from another UART. The set up process was the same as the previous parts, except I narrowed down the array size to 256 and only transfer 1 byte at a time in the Tx ISR. The Rx ISR was a little bit different and I will take about it later. We then needed to add second UART, with only the Rx interrupt enabled. In the second Rx interrupt, what I did was simply read whatever that was being received and add to the Tx FIFO. Then on the PSoC 5 board, we needed to connect the Rx of the first UART to the Tx of the second UART and vice versa. By removing the cables between them, I should be able to see errors/mismatches increasing on the LCD and reconnecting the cable should sync up the transfer and errors should stop. Here how I modify my first Rx ISR to accomplish this.

1. Added a second counter(rx2) and increase it together with the original one when comparing the mismatches. Also added another variable called cur\_error(initialize to 0).
2. When the jumper cable was removed and reconnected, errors would be generated, it was going to be greater than the cur\_error. Then I would set cur\_error equal to my error counter and stop incrementing my rx2. When the main loop went through and came back to the comparison of error and cur\_error, they would be the same. At that time I would start increment my rx2 again, so that it can start from where it was left off, thus sync up with the original data transfer.

A timer was added and configured to generate an interrupt of every 1s. A counter was put to the end of the first Rx interrupt to keep track of the transfer rate. Then I would update this through on the LCD every second. The theorical through put, as calculated above is 3490 bytes/second. I was getting around 3482 to 3489 bytes/second, which was pretty close.

A picture containing screenshot

Description generated with high confidence

**Figure 5: Part4 Top Design**

**Part 5: Receiver Clock Tolerance**

Both the top design and the code was nearly the same as part 4, with some minor changes to the configurations to both UARTs. This time we were told to add a timer to the second UART and by modifying the divider, observe if there were any errors generated in the data transfer.

A close up of a device

Description generated with high confidence

**Figure 6: Part 5 Top Design**

**Results:**

**Part 1:** both the error checking method and the data transfer/receive works fine.

**Part 2:** By transferring 4 bytes at a time, I was able to reduced the time entered the Tx ISR down to 1024 while the time enter the Rx ISR was 4096 since it was triggered on byte received.

**Part 3:** The error I got at 1.1ms, without hardware flow control was around 200, and the mismatches were wound 160. There should be something wrong with how I check for mismatches because ideally I should be getting around 4k mismatches.

**Part 4:** I was able to sync up with the transmitter after disconnecting and reconnecting the wire.

**Part 5:** I was only able to transmit/receive data without error when the divider was set to 26, that was around 923kHz. Any other frequencies below/over that would gave me errors.

**Conclusions:**

The purpose of this lab was to figure out how the UART works and how to use the configuration to manipulate and transfer the memory. I’m excited to learn about all these ways to manipulate memory and feel like more involved. I also like the fact that each part is built on top of the previous one since while I’m learning the new stuff, I’m also reviewing the old ones.A close up of text on a white background

Description generated with very high confidence