Andrew Mikhail

CMPE 121L: Microprocessor System Design Lab

Lab Exercise 3

28 October 2016

Table of Contents:

Introduction:

The purpose of this lab exercise is to learn about the functionality and usefulness of the Universal Asynchronous Receiver/Transmitter (UART) in the PSoC 5 system. The UART is serial transmitter/receiver system for relatively low-speed applications. The UART consists of shift registers used for parallel/serial conversion. Receiver may be synchronous or asynchronous. This technology is used widely in I/O protocols. This lab is also meant to introduce us to manage data flow as well as servicing interrupts. We began this lab by configuring the UART to be full-duplex, 38400 baud rate, 8 bit data, odd parity, 1 stop bit, no hardware flow, internal clock, and 16x oversampling.

Part 1: Transmit/Receive Based On Polling

To introduce us to using the UART we first wrote a program to transmit the values 0x45 and 0x5A continuously using the "UART_PutChar" command. We then observed the output of the UART on the oscilloscope and were able to identify the start, stop, data, and parity bits.

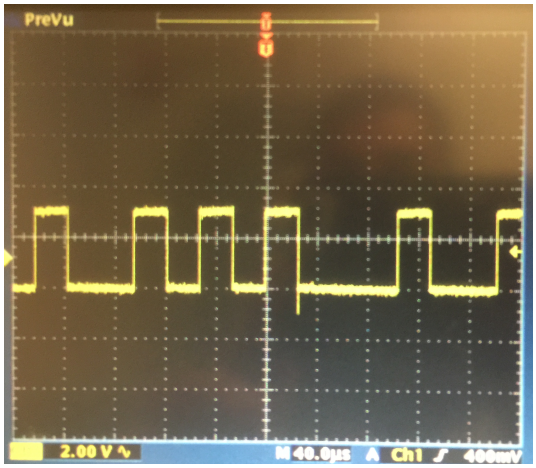

Figure 1: Oscilloscope Depiction of Data When 0x45 is Written to UART

Figure 1 shows the output we observed on the oscilloscope when 0x45 was written to the UART. First, we noted that 0x45 was 0b1000101. Then, we recognized that our output contained more bits because the UART adds a start bit, a stop bit, and a parity bit. Moreover, we also knew that a 0 (zero) signified a start bit and a 1 (one) was the stop bit. Furthermore, since there is an odd number of 1s in our binary number, the parity bit would be zero. Thus, we expected our output to be:

0	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**
Start									Parity	Stop

This output can be easily seen in the oscilloscope depiction in **Figure 1** if we read the waveform from right to left. Furthermore, we went on to write the value 0x5A to the UART, which is 0b1011010 in binary. Similarly, we observed the waveform in **Figure 2** on the oscilloscope

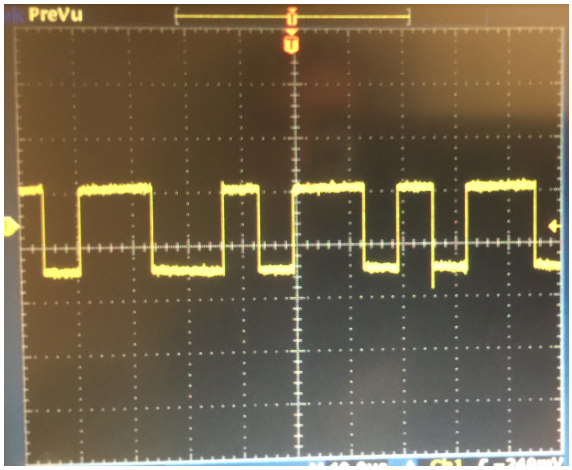

Figure 2: Oscilloscope Depiction of Data When 0x45 is Written to UART

We then extracted the following bits by reading the waveform from right to left.

0	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**
Start									Parity	Stop

In this section of the lab, we were also asked to transfer and receive data using the UART and measure the time it takes to complete the transfer. Our data came from a transfer array of size 4096 bytes that contained values 00, 01, ... FF. Using a for loop the data in the transfer array would then be transmitted and received using the UART then put into a receive array. To manage the transfer, we polled the transmitter (TX) status register to check if it was full before adding another byte to the FIFO. If the TX FIFO was not full, then we would add another byte to the FIFO then increment the index of the transfer array. Moreover, we polled the receiver (RX) status register to check it there was

a byte received. If the RX FIFO was not empty, then we placed the byte in it in the receive array then incremented the index of the array.

To measure the time it took to complete the transfer, we added a Timer block to the top-level schematic, began the timer as soon as the first byte was received, and stopped the timer when the last byte was received. Theoretically, we expected the elapsed transfer time to be about 1.173 seconds. We calculated that time based on the given 38,400-baud rate and the addition of the start, stop, and parity bits.

$$\frac{38400\ baud}{11\ bits} = 3490.90 \frac{bytes}{second}$$

$$\frac{4096\ bytes}{3490.90 \frac{bytes}{second}} = 1.173\ seconds$$

Our experimentally calculated elapsed time to complete the transfer was 1.171 seconds. Thus, we almost perfectly matched the theoretical time.

Part 2: Transmit/Receive Using Interrupts

This section of the lab was similar to Part 1, though we used interrupts to check the status of the RX and TX status registers for this part. Though, we had to minimize the total number of interrupts generated during the execution of the program. To do so, the transmitter only interrupted when the TX FIFO was empty; at which point, four more bytes were added to the FIFO. On the receiver side, the RX FIFO issued an interrupt every time a byte was received; at which point the received byte would be added to the receive array. While the data was being transmitted and received, the main program maintained in an idle state until the transfer was complete. This was done by creating a while loop having a condition:

```
while(flag != 1)
```

Then, once the transfer was complete, flag would be set to 1 and the main program continued executing. At that point, the transfer and receive arrays were compared for errors. The number of errors would then be displayed on the LCD screen.

We also placed counter variables txCountISR and rxCountISR inside both the RX and TX interrupt service routines (ISR) that would be incremented every time the ISR was entered. The counts of how many times the ISRs were entered were also printed on the LCD display. As expected, the RX ISR was entered four times more than the TX ISR. The RX ISR was entered a count of 4096 times while the TX was entered a count of 1092 times. This was expected because the TX only interrupts when the TX FIFO is empty. Moreover, the TX FIFO has a buffer size of 4 bytes. Thus, the TX will interrupt 4 times less than the RX since the RX FIFO interrupts on every byte received.

Part 3: Hardware Flow Control

We began this part of the lab by disabling the RX interrupt and enabling only the TX interrupt. As noted previously, the transmitter only interrupts when the TX FIFO is empty. Moreover, we added a timer block to the top level schematic that would interrupt every millisecond. Every millisecond the timer ISR would check the status of the RX FIFO, add any received data to the receive array, and then check for errors. Stop errors, parity errors, as well as FIFO overrun errors were all checked with an error flag. After the data transfer was complete, the transfer and receive arrays were also compared for errors. This configuration resulted in a number of errors, both comparison errors as well as status errors.

For me personally, my code was caught in an infinite loop not displaying anything on the LCD. This was due to the fact that my TX FIFO would transmit all of its data and stop interrupting before the RX FIFO had time to receive all of that data. Moreover, the RX FIFO would become overrun with data and not able to handle these errors. However, all of the errors were resolved by enabling hardware flow control. The error flag also did not show any stop, parity, frame, or FIFO overrun errors. Enabling hardware flow control brought out the CTS_N and RTS_N wires on the UART block. The CTS_N input and RTS_N output were wired together on the top-level schematic and assigned to a pin driving LED3 on the board. The LED would turn on when the transfer was initiated and remain on until the transfer was complete. Hardware flow control got rid of all the errors because the hardware became in control of the data transfer. This meant that although the software had finished its transmission of data, the hardware controlled the flow of bytes

into both the TX and RX FIFOs. Consequently, the infinite loops and FIFO overrun errors were eliminated and the program was able to run correctly.

Part 4: Data Transfer With Remote System

For this part of the lab, I actually did not receive full credit for a couple of reasons. My UART's performance was not calculated correctly and thus did not stop when the TX pin was disconnected from the RX pin. I calculated the TX throughput by taking the current index of the transmitter array and multiplying it by 1000 within the timer interrupt. Since the timer interrupted every second, I thought that this would provide us with how many kilobytes had been sent per second. However, the correct implementation of calculating the throughput would have been to remember the index of the array from a second ago and subtract it from the current index of the array to determine how much data had been transferred. Though if I had calculated throughput correctly, I would expect it to be about 3490.90 kilobytes per second as calculated in Part 1.

I was also docked points for not syncing my system with the remote system that would be attached. I was supposed to reset the error count, realign with the position of the index of the remote transmit/receive array, and recalculate the throughput. However, my code did not accomplish that. I struggled with this part of the lab because I did not know how I would re-sync my transmission of data. However, I would begin by checking if the index of the transmit array is equal to the index of the receive array. If they were identical, then I would perform the necessary readjustments. Though, if they weren't I would make my program idle until they were.

Nonetheless, my program still checked for errors in the transmit and receive arrays correctly. I also checked for parity errors, frame errors, as well as FIFO overrun errors along the transmission of data properly. Moreover, as long as the TX pin was connected to the RX pin, data flow was continuous, accurate, and did not contain any errors.

Conclusion:

I thought this lab was very helpful and informational to understanding the capabilities of the UART. This lab provided a complete introduction to using UART to transfer and receive data on the PSoC-5 microcontroller board. We also became familiar with data protection with parity bits as well as error detection with status bits. This lab provided us an opportunity to combine concepts of serial data flow from lecture with a relevant experiment. Overall, I thought this lab was well presented and quite informational on the topic of UART. However, I think that this lab can be improved by clarify the instructions for Parts 3 and 4. I felt that as the lab went on, the instructions became more vague and unclear with what the assigned task was.