Lab 3

Kyle Jeffrey

Abstract—The third lab explores the UART communication protocol.

1 Introduction

The UART Asynchronous communication protocol is a fairly low overhead serial communication protocol that is used in medium speed systems. This lab implements the protocol on the PSoC 6 using two different techniques: Interrupts, and Direct Memory Access. The UART Transmit and Recieve side both have a First In First Out (FIFO) register of 128 bits for storing the data before it is sent and storing data before it is read respectively.

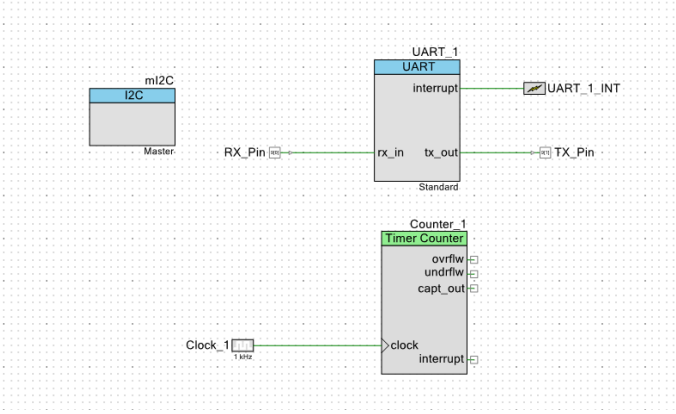

Fig. 1. Component Design

2 Part 1: UART Transmit/Recieve Based on Interrupts

In this part, a 4096 block of datta was transmitted through the UART and recieved by the same UART, and then checked for data integrity. The UART had flags set for TX NOT FULL and RX NOT EMPTY that would trigger interrupts as shown in the top design. Also notice the counter, which was used to calculate the baud rate of the system. On the first recieve byte the counter was started and on the last the counter was stopped. Knowing that 4096 bytes were transmitted and know knowing the amount of time it took to transmit, the baud rate could then be calculated. The Baud was 115.2k with an odd parity and 1 stop bit, meaning every byte of data was carried by an 11 bit sequence. Another UART was setup for terminal printing to check the data output of the tx at the end of the sequence. The top level design was a wire connecting the TX transmit to the RX recieve input.

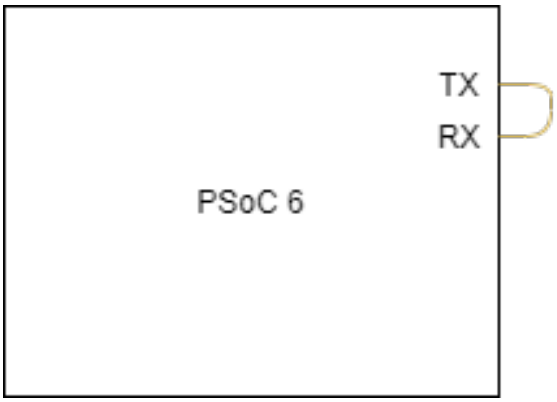

Fig. 2. Top Level Design

2.1 Throughput and Interrupts

The interrupts for the Transmit and Recieve ISR were counted. The expected value for both would be 4096 as the tx transmitted byte by byte and the recieve rx recieved byte by byte. Interestingly though, there were 4096 transmit interrupts but there were 4196 recieve interrupts. This is because the recieve side would occasionally read junk data.

The baud rate of the PSoC was calculated, and came out to 115,825. The theoretical Baud of the design was 115,741. The minor discrepency could do with the extra interrupts occuring on the recieve ISR, which could be eliminated by reducing the number of interrupts to 4096.

3 Part 2: UART Transmit/Recieve Based on DMA

In this section, the UART was implemented using Direct Memory Access instead of interrupts. Both the Rx and Tx had a DMA component

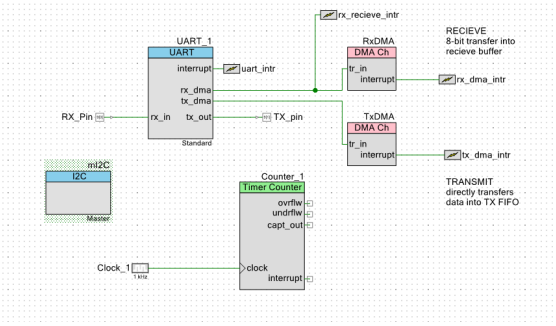

Fig. 3. Component Design

that were connected to an ISR. The ISR was triggered on completion of a 4096 data transfer to the Rx. The $txdma$ signal was active for FIFO Level less than 127 meaning that the dma would transfer data to the TX FIFO when the FIFO was below 127 bytes. The UART FIFO's

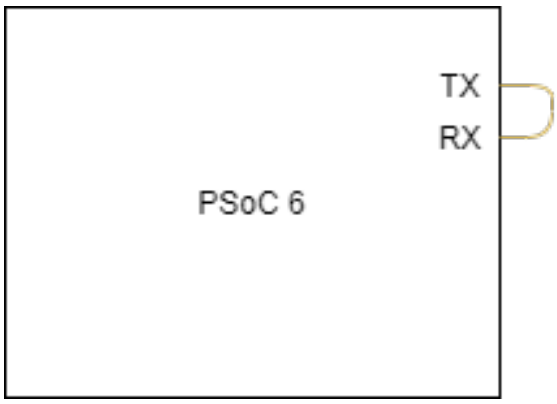

Fig. 4. Top Level Design

are only 128 bytes, so this like triggering at FIFO NOT FULL. The $rxdma$ signal was active for FIFO level 0, or FIFO NOT EMPTY. Using the DMA greatly reduces the software needed for implementing the UART.

3.1 Baud

The calculated baud for this implementation was lower than the actual baud at 115,528 instead of 115,741. The reason for this is unknown but could simply be small variations in precision of the UART component.

4 Part 3: Hardware Flow Control

This section uses the design of the first part and shifts some things around. Instead of using the UART Interrupt for the RX interrupt, a counter is used to generate an ISR every .5 ms. This means that the UART Recieve should read a byte every .5s. Doing the interrupt at

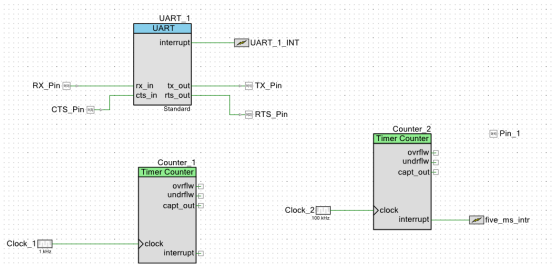

Fig. 5. Component Design

every .5ms, after transferring the 4096 bytes, there were 3924 errors, and 734 overflow errors. Obviously this was a very bad replacement implementation for part 1. Changing the interrupt

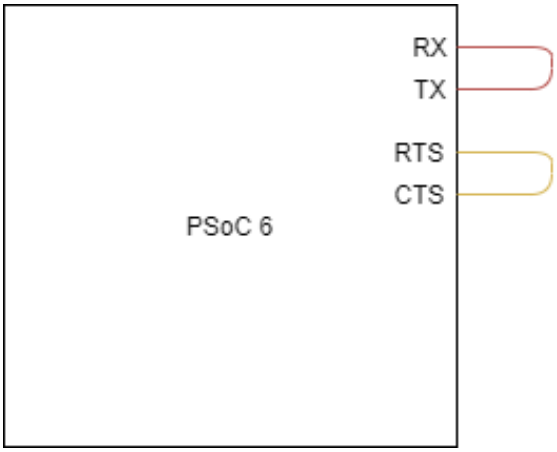

Fig. 6. Top Level Design

time to 2 ms, there were 3946 errors and 187 overflow errors. This is expected as an interrupt triggered less frequency would read less

overflow errors. For a 115.2k baud, the device would need at least an interrupt time of 8.6 microseconds, or use a 115 Khz clock.

4.1 Using Hardware Flow Control

After testing with the .5 ms interrupts and recieving many errors, hardware flow control is enabled on the UART. This is the Clear To Send (CTS) and Ready To Send (RTS) signal. After connecting them together with a wire, the errors recieved went down to 65, with no overflow errors at .5 ms interrupts. The errors went down as now the UART doesn't send any data unless the recieve side responds with a ready to send signal.

5 Part 4: Receiver Clock Tolerance

In this section we test the tolerance of the UART protocol to differences in clock speed on the recieve and transmit side of the serial link. The UART is popular because of it's tolerance to these inaccuracies. Here the recieve UART was

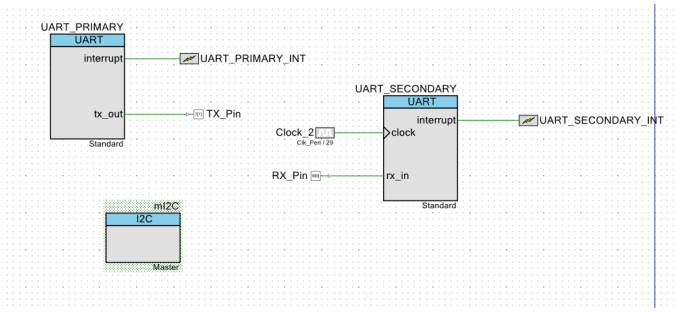

Fig. 7. Component Design

clocked by a peripheral clock.

5.1 Clock Tolerance

The UART was tested for 8x and 16x oversampling on the recieve side which. Using 8x oversampling, the UART recieve had a range of 893Khz to 943Khz. The 16x oversampling had a clock range of 1.78 to 1.92Mhz. Notice that the 16x sampling rate has a clock range of 140 Khz and the 8x sampling rate has a clock range of 50Khz. The 16x sampling rate has a greater range of tolerance for the recieve clock.

$$\frac{140Khz}{1.92Mhz} * \frac{1}{2} = 3.6\% for 16x \quad (1)$$

$$\frac{50Khz}{921.6Khz} * \frac{1}{2} = 2.71\% for 8x \quad (2)$$

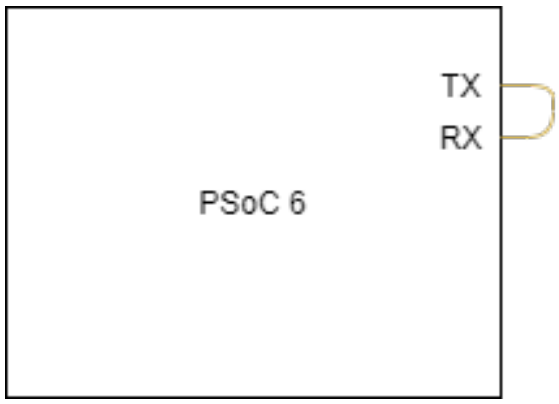

Fig. 8. Top Level Design

6 Conclusion

This lab explored different approaches to implementing the UART serial protocol. We find that it's advantages include a small overhead and good clock error tolerance between the transmit and recieve clock.

Kyle Jeffrey is a Senior Robotics Engineering Student at 6the University of California Santa Cruz. He is the Secretary of the Engineering Fraternity Tau Beta Pi and the lead Hardware Engineer at the on campus startup Yektasonics.