Leonid Shuster

CSE 121

Lab Section 1C

11/4/19

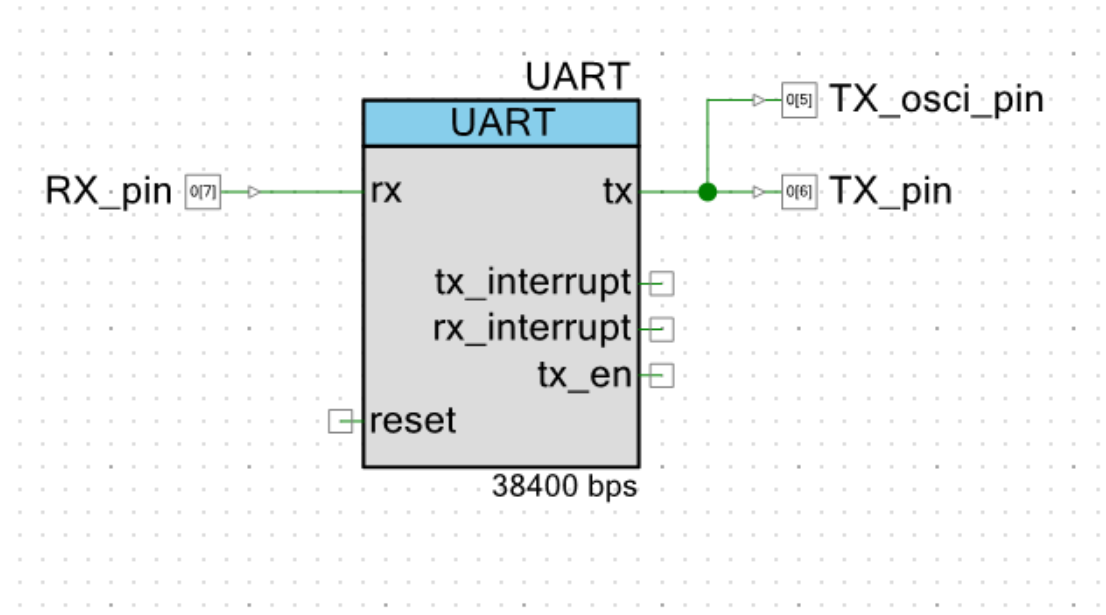
Lab Report 3

**Introduction**

In this lab, we were tasked with learning how to use the UART in the PSoC 5. In the first part, we set up the UART to transmit and receive data using either polling or interrupts, and analyzed the time taken to transfer all the data using polling or the number of interrupts that were triggered using interrupts. In the next part, we added a timer that generated interrupts at a certain interval to slow down reads from the RX FIFO in order to overflow the FIFO, and added hardware flow control that prevented the FIFO from overflowing. In the third part, we linked two UARTs that continuously transmitted and received data to and from each other, and measured the number of errors. In the fourth part, we again used two UARTs but only linked the first one to the second one, and connected a clock to the second one in order to determine what the maximum clock tolerance could be in order for the second UART to work properly. And finally, in the last part, we created a UART using only software, and used a second UART to test the soft UART.

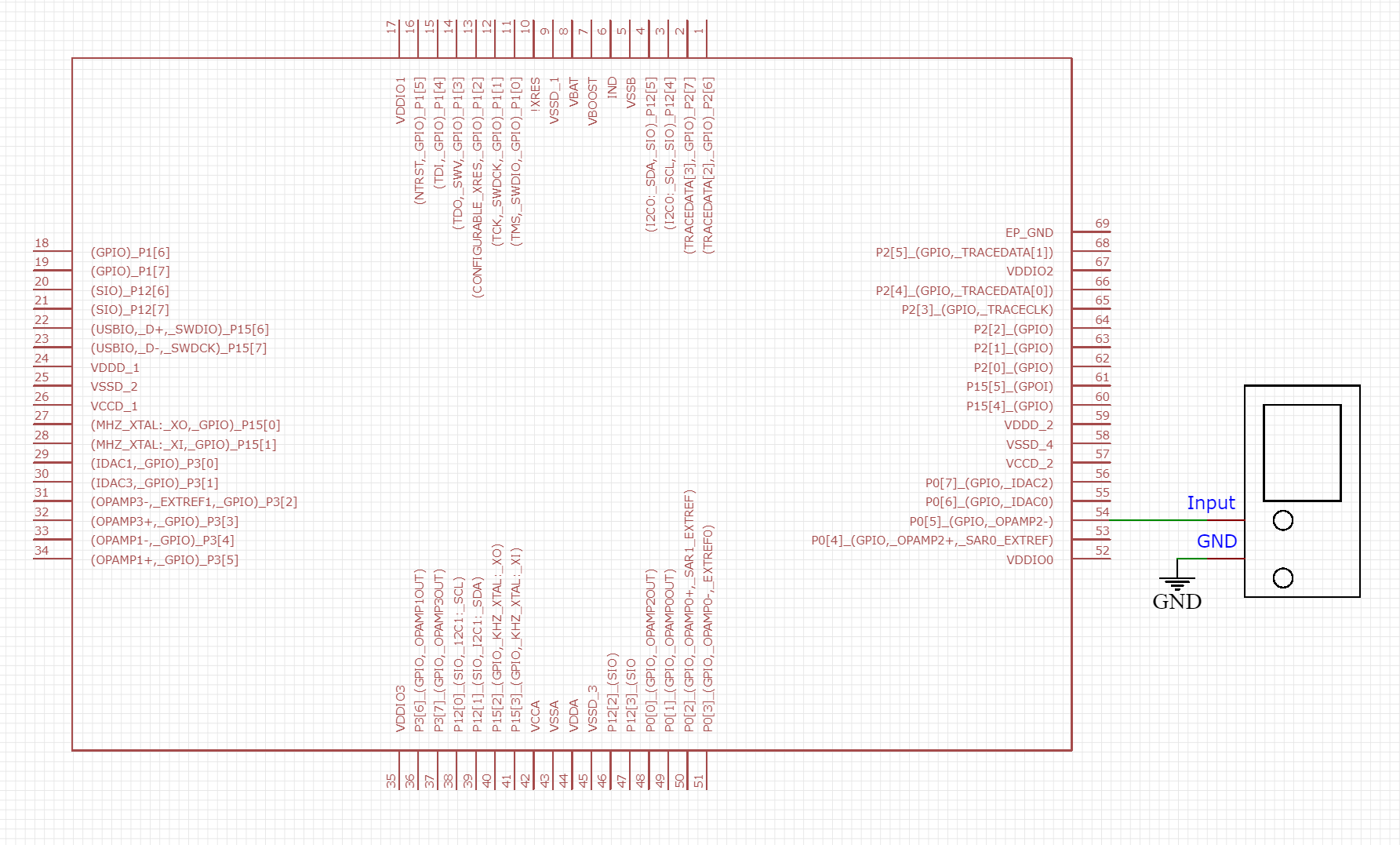
**Part 1(a): Transmit/Receive Based on Polling**

In the first part of the lab, we were told to use the UART to continuously transmit the byte value 0xa5 and measure the serial data output on an oscilloscope. In my top design, I added a UART, and configured it as stated in the lab manual (see Figure 1).



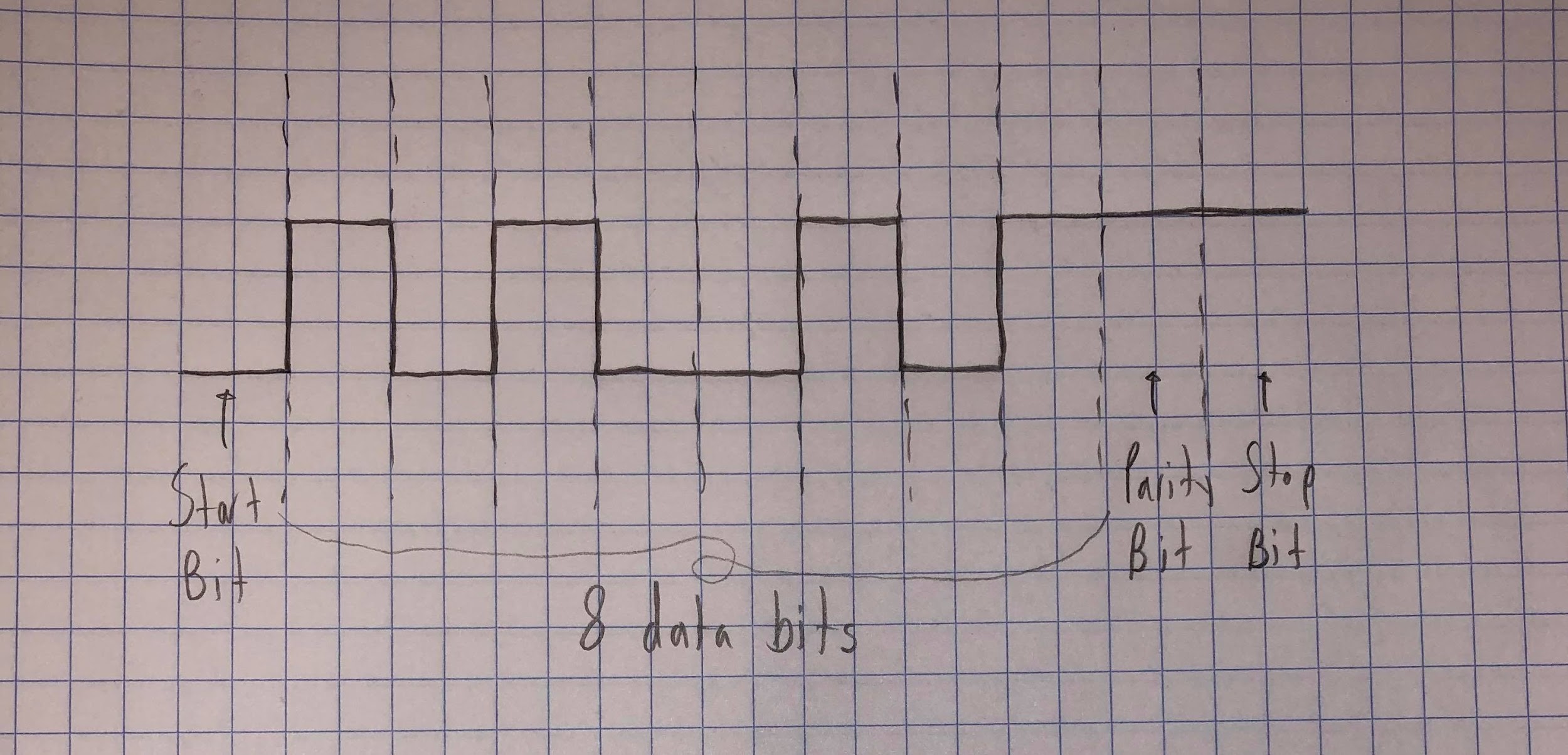
*Figure 1: Part 1(a) Part 1 Top Design*

In main.c, I kept transmitting the byte value 0xa5 to the TX FIFO in an endless for loop. On the outside of the PSoC, I only had one of the GPIO pins connected to an oscilloscope (see Figure 2).



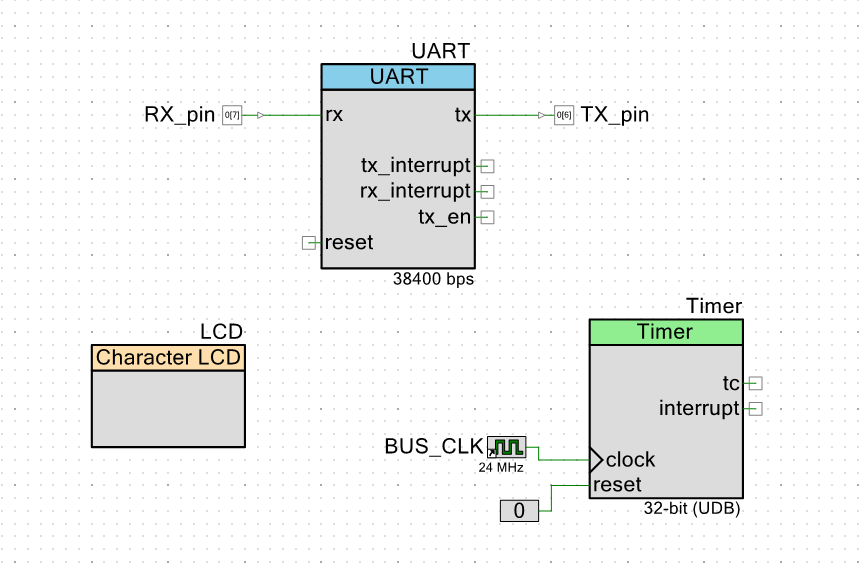
*Figure 2: Part 1(a) Part 1 External Schematic*

Each transmit consisted of one start bit, eight data bits, a parity bit, and a stop bit, so 11 bits in total. The bits were organized as 0 for the start bit, 10100101 for 0xa5, 1 for the parity bit since the parity was set to odd and the number of 1’s in the data sequence was even, and 1 for the stop bit, resulting in 01010010111 (see Figure 3).



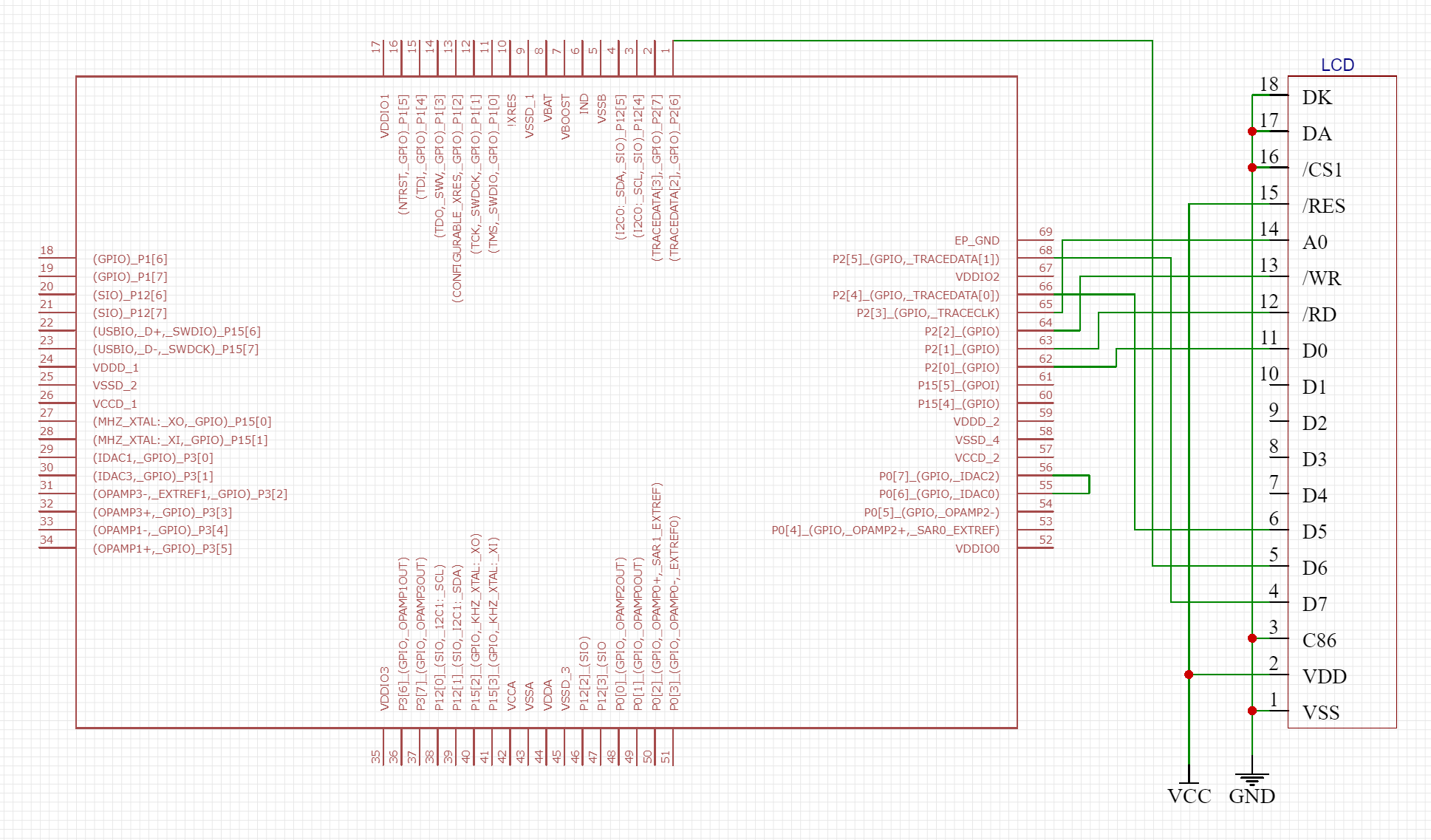
*Figure 3: Part 1(a) Part 1 Waveform*

After transmitting a byte value continuously, we then had to transmit and receive data through the same UART by polling the status of the UART registers, and present the time taken to complete the transfer on an LCD display. In my top design, I added a UART with an input pin connected to the RX input and an output pin connected to the TX output, a timer, and an LCD display (see Figure 4).



*Figure 4: Part 1(a) Part 2 Top Design*

On the outside of the PSoC, I connected the RX and TX pins to each other with an external wire, and the LCD to the PSoC (see Figure 5).



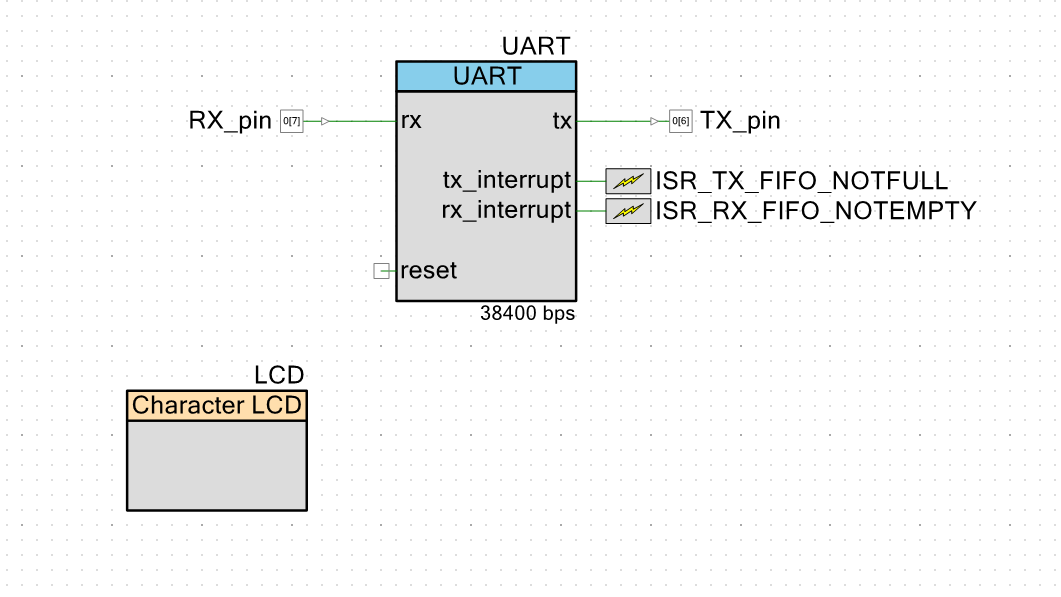
*Figure 5: Part 1(a) Part 2 External Schematic*

In main.c, I created two arrays of size 4096 bytes in SRAM memory that would be used to transmit and receive data, initialized the transmit array with an increasing pattern of bytes, and started the UART, timer, and LCD. In an endless for loop, I constantly polled the TX FIFO and RX FIFO registers of the UART checking if the TX FIFO was not full and if the RX FIFO was not empty. If the TX FIFO was not full, I added a byte to the TX FIFO from the transmit array and incremented a counter that kept track of the amount of transmits and the index of the transmit array. If the RX FIFO was not empty, I read a byte from the RX FIFO into the receive array and incremented another counter that kept track of the amount of receives and the index of the receive array. Once the number of receives reached the size of the transmit/receive array, I stopped the timer and printed the time taken to transfer the data onto the LCD, as well as the number of mismatches between the transmit and receive arrays.

When running the program, I didn’t have any mismatches. However, the expected time when setting the UART at 38,400 baud rate should have been around 1.17 seconds, but my elapsed time was a little bit higher at 1.63 seconds. I believe this discrepancy comes from the fact that there is a short amount of time that goes by when the amount of bytes received is being checked, especially when the last byte is received and before I stop the timer. To fix this, I could stop the timer immediately after the last byte is received.

**Part 1(b): Transmit/Receive Based Using Interrupts**

In the next part, we were tasked with creating a design similar to the previous part, except that the UART would transmit and receive data using interrupts rather than polling. In my top design, I added a UART with an input pin connected to the RX input and an output pin connected to the TX output with interrupts enabled that would trigger when a byte was received on the RX FIFO or when the TX FIFO was not full, with interrupt components connected to the UART interrupt ports. I also had an LCD (see Figure 6).

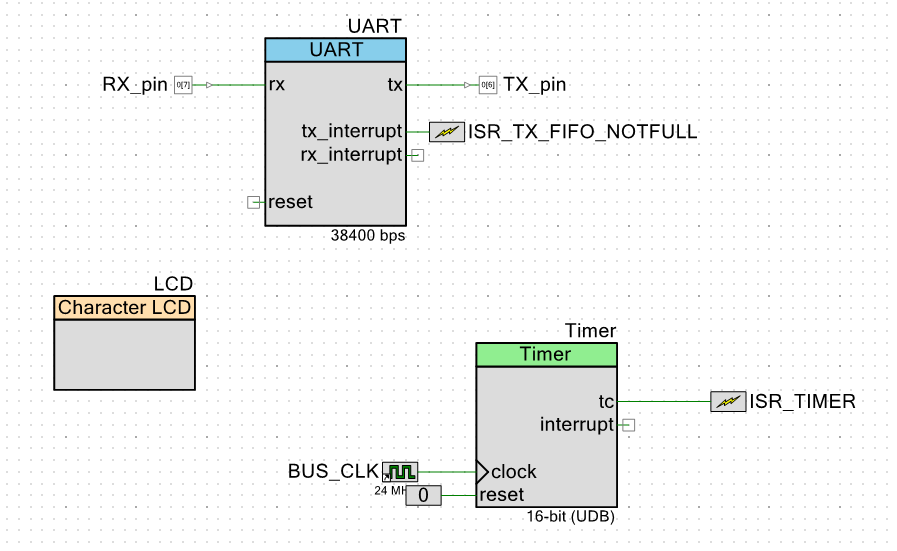


*Figure 6: Part 1(b) Top Design*

The outside of my PSoC remained the same, and in main.c, I again created two arrays of size 4096 bytes in SRAM memory that would be used to transmit and receive data, initialized the transmit array with an increasing pattern of bytes, and started the UART, LCD, and interrupt procedures. In the interrupt procedure that was triggered when the TX FIFO was not full, I had a while loop that constantly polled the status of the TX register and put bytes into the TX FIFO until it was not full, once again incrementing a counter that kept track of the number of transmits and the index of the transmit array. I also had a separate counter outside the while loop that kept track of the amount of times the TX FIFO interrupt was triggered. Meanwhile, in the interrupt procedure that was triggered when the RX FIFO was not empty, I had another while loop that constantly polled the status of the RX register and read bytes from the RX FIFO until it was not empty, also incrementing a counter that kept track of the number of receives and the index of the receive array. I also had a separate counter outside the while loop that kept track of the amount of times the interrupt was triggered. This interrupt also had a flag that would become true once the number of receives matched the size of the transmit/receive array. In my main part of my program, my program remained in an idle loop until the flag became true symbolizing that the entire transfer was complete. Afterwards, I printed the number of times the TX FIFO and RX FIFO interrupts were triggered, as well as the number of mismatches between the transmit array and receive array. When I first ran the program, I had no mismatches, but because I set the TX FIFO interrupt to trigger when it was not full, the program entered the TX FIFO interrupt procedure 4093 times and the RX FIFO interrupt 4096 times. When I changed the TX FIFO interrupt to trigger when it was not empty, I saw that the number dropped significantly to 1024 times, which was the lowest I could get it to.

**Part 2: Hardware Flow Control**

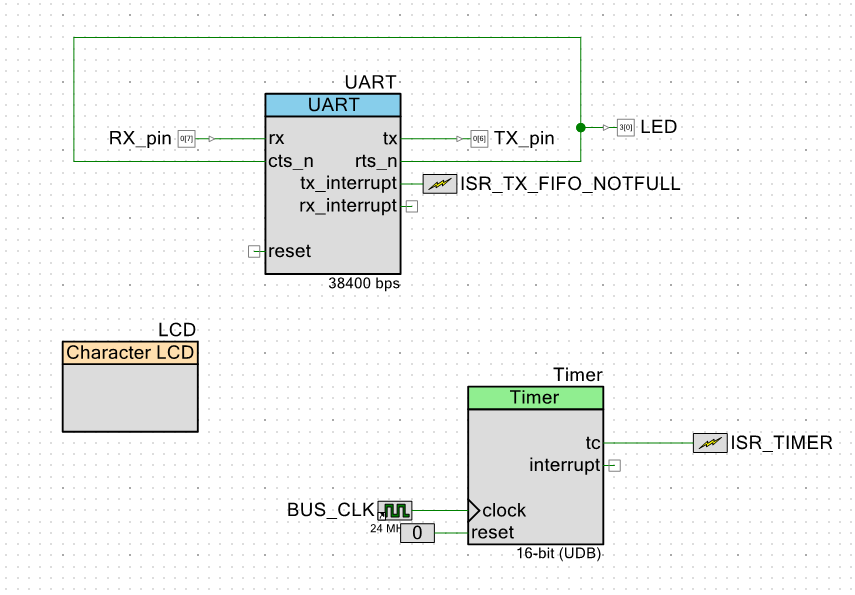
In the next part, we were tasked with replacing the RX FIFO interrupt with an interrupt that would be triggered by a timer at a certain interval of time in order to intentionally create an RX FIFO overflow, and then later add hardware flow control to fix this problem. For the first part without hardware flow control, my top design had a UART with the same setup as before, except that it only had a TX FIFO interrupt connected to an interrupt component that was triggered when the TX FIFO was not full. I also had a timer that went off at a certain interval of time with an interrupt component connected to its TC, which meant that the interrupt would trigger every time the timer went off. Finally, I also had an LCD (see Figure 7).



*Figure 7: Part 2 Part 1 Top Design*

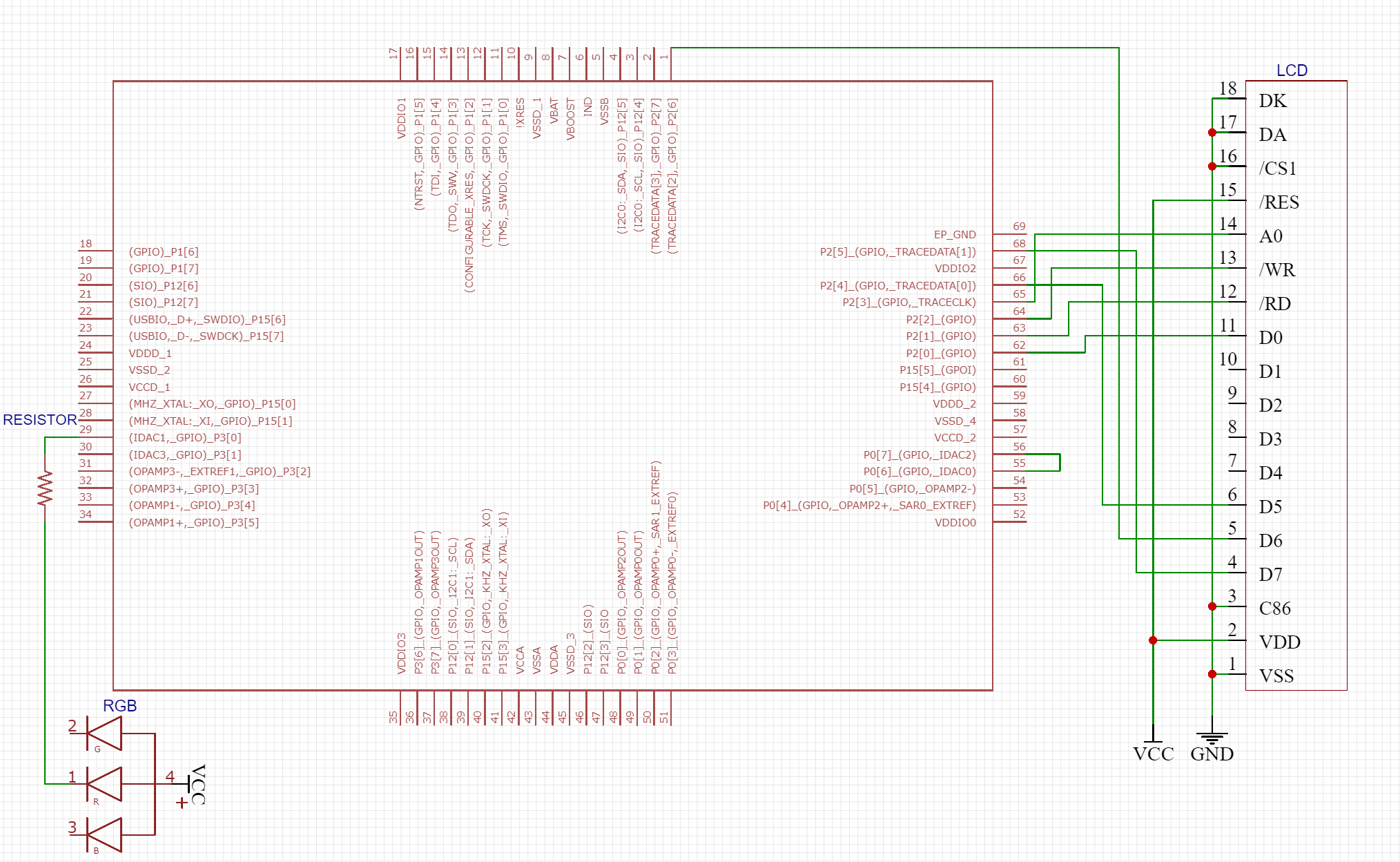
My layout on the outside of the PSoC was the same as before, and in main.c, I reused the same code from part 1(b) but removed the RX FIFO interrupt procedure. Instead, I had an interrupt procedure that triggered from the clock that performed the same actions as the RX FIFO interrupt did, including reading from the RX FIFO. In this interrupt, I also had a flag that became true if the RX FIFO overflowed and a counter that kept track of the amount of errors. My main part of the program was the same as before; I started the UART, timer, LCD, and interrupt procedures and remained in an idle loop until the flag keeping track of the number of receives became true, and then printed out the number of mismatches and if there were any errors or overflows. When I set my timer to go off every 0.5 milliseconds, I got a lot of mismatches, but no errors or overflows. I believe this is the case because the timer and thus its interrupt still go off fast enough to read from the RX FIFO. However, when I set my timer to go off every 1.2 milliseconds, I got errors and overflows, which I believe comes from the fact that the timer and its interrupt are now slow enough so that the RX FIFO cannot keep up with the TX FIFO transmits. I estimate that the maximum period the timer can be set to without seeing any errors is around 0.7 milliseconds.

Next, we were told to enable hardware flow control in the UART. In my top design, I had it the same as before except that I looped the CTS\_N input and RTS\_N output to each other, and connected an output pin to this loop which would go to an LED (see Figure 8).



*Figure 8: Part 2 Part 2 Top Design*

On the outside of the PSoC, I had an LED connected to the output pin and an LCD connected to the PSoC (see Figure 9).

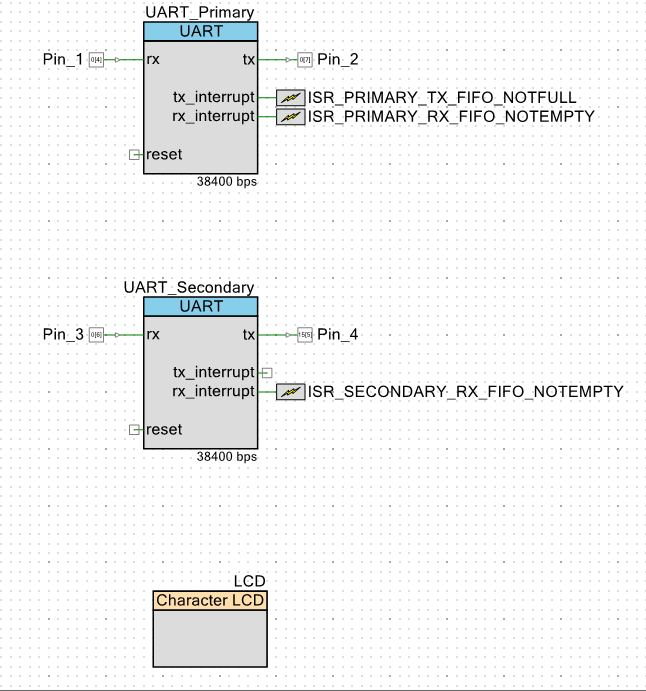


*Figure 9: Part 2 Part 2 External Schematic*

My main.c was the same as before, and when I ran the program with my timer set to 1.2 milliseconds, I had no mismatches, errors, or overflows, which I believe is due to the hardware flow control first ensuring that the bytes can be transmitted and received before it transmits, preventing any RX FIFO overflows.

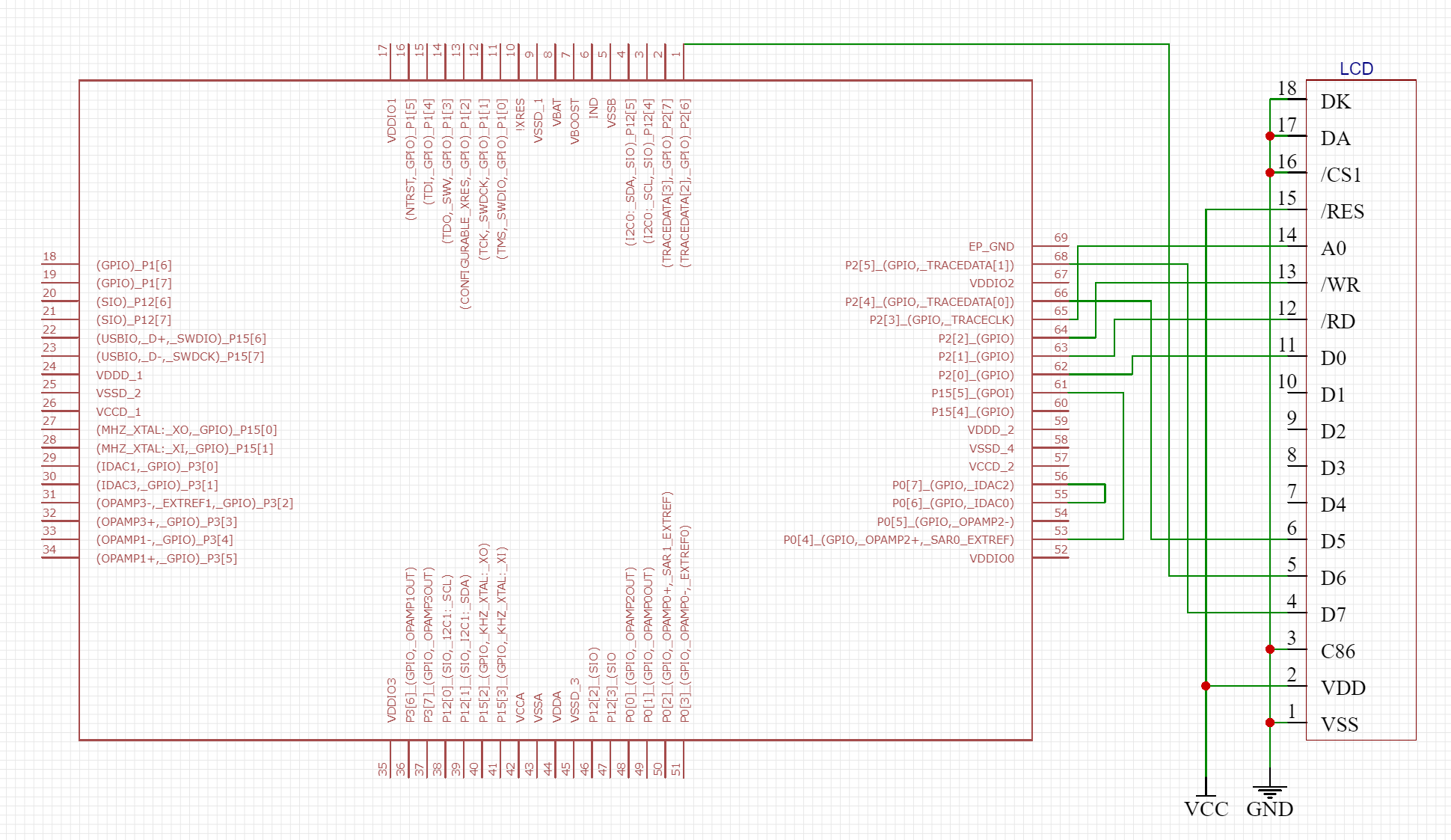
**Part 3: Data Transfer between Two UARTs**

In the next part, we were tasked with linking two UARTs that would continuously transmit and receive data to and from each other. The first UART would transmit data to the second UART, and the second UART would act as an echo and read and transmit the data back to the first UART. In my top design, I had two UARTs, with the first UART having an input RX pin, output TX pin, and RX FIFO interrupt on byte received and TX FIFO interrupt on TX FIFO not full, and the second UART having an input RX pin, output TX pin, and only an RX FIFO interrupt on byte received. I also had an LCD (see Figure 10).



*Figure 10: Part 3 Top Design*

On the outside of the PSoC, I connected the first UART’s TX pin to the second UART’s RX pin, and the second UART’s TX pin back to the first UART’s RX pin. I also connected the LCD to the PSoC (see Figure 11).



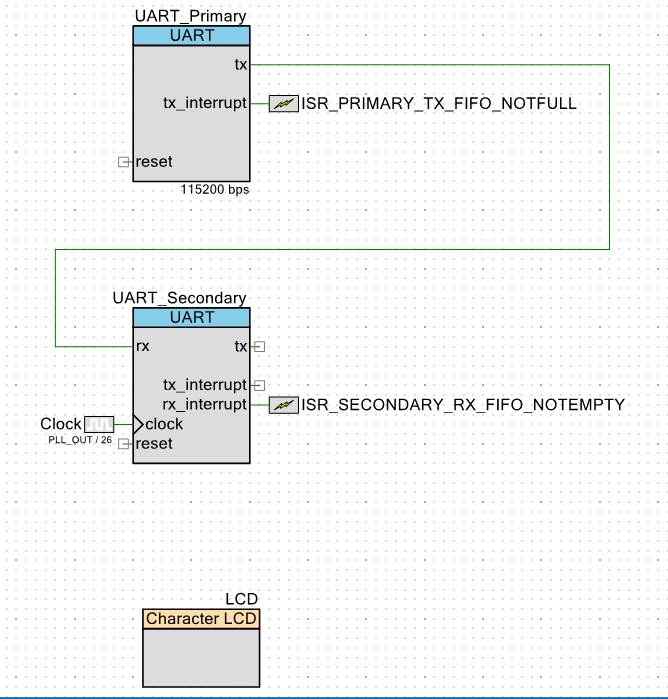
*Figure 11: Part 3 External Schematic*

In main.c, the interrupt procedures for the interrupts from the first UART were set up the same as from part 1(b) where the TX FIFO interrupt put data from the transmit array into the TX FIFO while it was not full and the RX FIFO read data into the receive array while it was not empty. Meanwhile, the interrupt procedure for the second UART simultaneously read data coming in and sent data back out while the second UART’s RX FIFO was not empty. By doing so, the second UART was essentially an echo for the first UART, merely transmitting the data back to the first UART. In the interrupt procedure for the first UART’s RX FIFO, I also checked whether the value from the receive array matched the same one from the transmit array, and printed the value out in the main part of my program.

When I ran the program, I was not able to correctly count the number of mismatches between the receive array and transmit array. I know that I was supposed to continuously transmit and receive data between the two UARTs, and if I broke the connection between the two and reconnected them it would sync back up, but I was not able to implement this. I’m not sure why, but the number of transmits from the first UART was not matching the number of receives from the second UART. However, when I connected the two UARTs in my top design rather than through a physical wire, I was getting the right amount of transmits and receives and thus no mismatches, but by doing so there was no way I could unplug their connection and then reconnect them so that they could sync up again. I believe the reason connecting them physically did not work is because I was comparing the two arrays and their values inside of the first UART’s RX FIFO interrupt procedure using the same index, which may not be synced at that point.

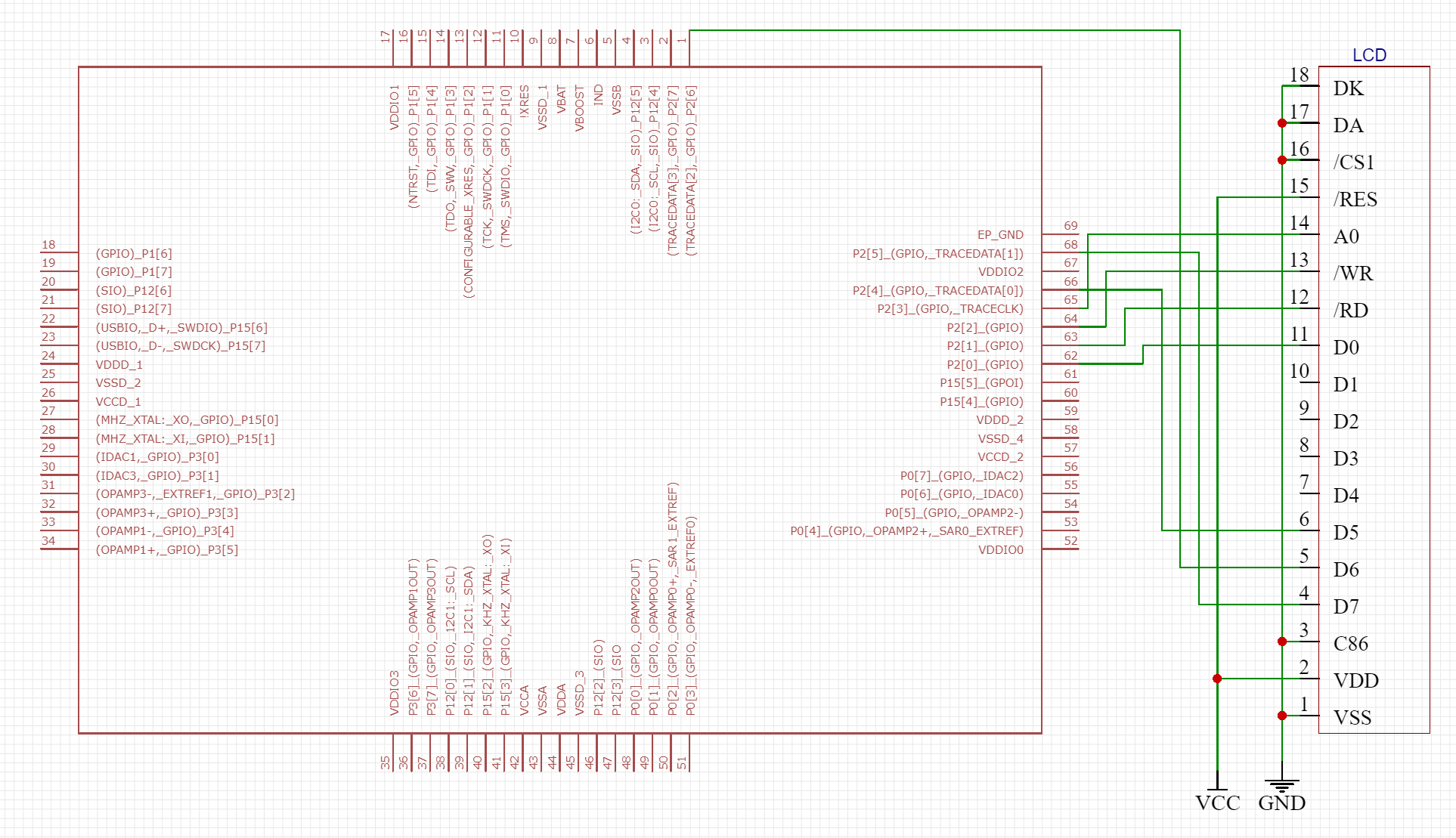
**Part 4: Receiver Clock Tolerance**

In this part, we were tasked with using two UARTs in which one of the UARTs transmitted data to another UART which was connected to an external clock. In my top design, I added two UARTs and connected the first UART’s TX pin to the second UART’s RX pin, and added a TX FIFO interrupt when not full to the first UART and an RX FIFO interrupt when not empty to the second UART. I also added an LCD and an external clock to the second UART, which I configured to be 24MHz with a divider value of 26 in order to match the first UART’s clock frequency as close as possible (see Figure 12).



*Figure 12: Part 4 Top Design*

On the outside of the PSoC, I only had an LCD connected to the PSoC (see Figure 13).



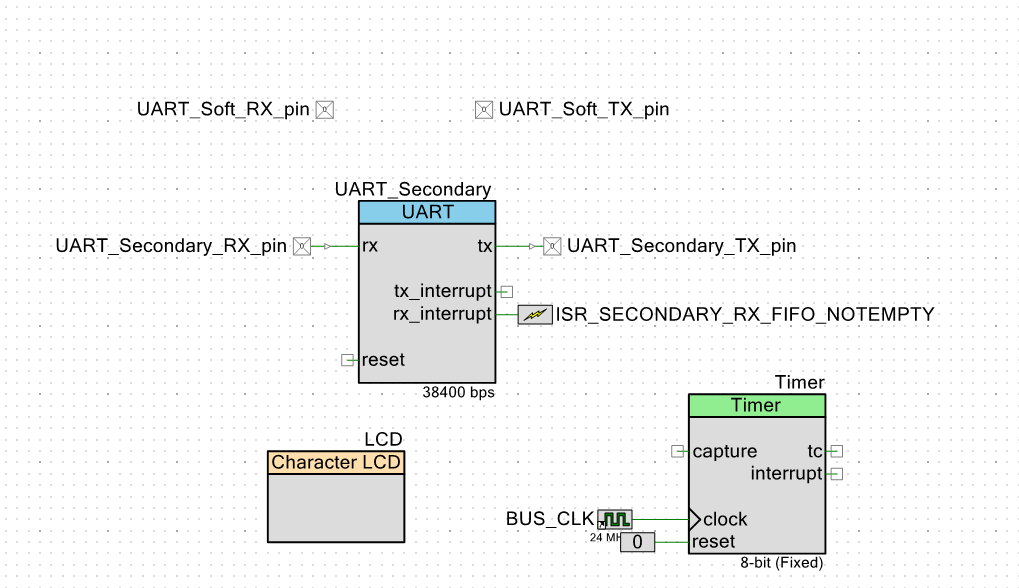
*Figure 13: Part 4 External Schematic*

In main.c, I used the same interrupt procedures from part 1(b), except that I used the TX FIFO interrupt procedure for the first UART in which the TX FIFO keeps getting filled until full and the RX FIFO procedure for the second UART in which the RX FIFO keeps getting read until empty. Like before, in the main part of my program I also printed the number of mismatches between the transmit and receive array.

When I ran the program with the external clock connected to the second UART set to 24 MHz with a divider value of 26, I did not encounter any mismatches. The lowest I could set the clock without seeing any errors was at a divider value of 25, while the highest I could go was a divider value of 24, which agrees with the findings I found of the clock tolerance having to be +/- 2% of the first UART’s clock.

**Part 5: Soft UART**

In the final part, we were tasked with creating a UART through software that we would test with a second UART. In my top design, I only had an input pin and output pin for the soft UART, and a second UART component with an input RX pin and output TX pin as well as an interrupt enabled when the RX FIFO received a byte. I also had a timer and an LCD (see Figure 14).



*Figure 14: Part 5 Top Design*

Unfortunately, that was all I was able to do, but in my main.c I would’ve started with the soft transmitter first and set it up so that TX would output the starting bit, shift the data bits 8 times, and then finally send out the parity and stop bits, which could be generated by using a timer that was set to the length of a bit. For the receiver, I would’ve created a state machine that ensures that the data coming into it matches the plan of a start bit, 8 data bits, parity bit, and stop bit.

**Conclusion**

After having gone through the lab, I feel like I fundamentally understand how DMAs work now. From using polling and interrupts in the first part, adding hardware flow control in the second part, connecting two UARTs to transfer data in the third part, connecting two UARTs and adding an external clock to the second one, and creating a UART through software, I feel confident that I can effectively use UARTs and integrate them into my future designs. If I were to do this lab again, I would use the TX FIFO interrupt when empty rather than when not full, because I believe a lot of my problems came from using that type of interrupt.