Week 12:

Date: 11/11/2022

Hours: 14

Description of design efforts:

PCB Testing:

This week was dedicated to testing proper functionality on our PCB. Fortunately, our determination in creating a proper PCB was worth having few errors to resolve, none significant enough to need to reorder the PCB. We have a few fly-wires, connected on the button matrix PCB to fix a trace that was not correctly connected by JLCPCB – this was not our fault (though it could be argued that we should have tested every trace before soldering components on it).

Since all the microcontroller was written by me, it was my responsibility to debug any issues that surfaced through using the PCB instead of the ECE 362 development board. Nothing severe was caught – most of this work was smooth sailing, with methodical soldering and testing. The only errors we found were that mentioned above on the button matrix PCB, pads too small for the external oscillator, and a notable change that does require attention.

We found that our UART messages were not properly being sent or received. Instead of receiving 1 from the computer, issuing a request for data, we found messages such as 255 or 253. When returning messages, garbage was received (either 0 or -32). As expected, this was due to an incorrect baud rate. Using the oscilloscope, we found that our microcontroller was using around 18,800 baud, while we expected it to be 115,200. Our actual value was nearly 6 times smaller than we expected, which corresponds to using an 8 MHz clock instead of 48 MHz.

Then, the confusion began. I knew the software was not the culprit, as the code was the same for the development board and for the PCB. However, there was no hardware on the development board's schematic that was set to change this. So, as it currently stands, we believe there is some non-volatile memory that holds these values, that perhaps were loaded to in an initialization code for the development boards that we do not have on our microcontroller on our PCB. We found, through the debugger, some values that should be set to multiply the clock in RCC to be 48 MHz that are unset but are on the development board. The clock tree is shown in Figure 1, below.

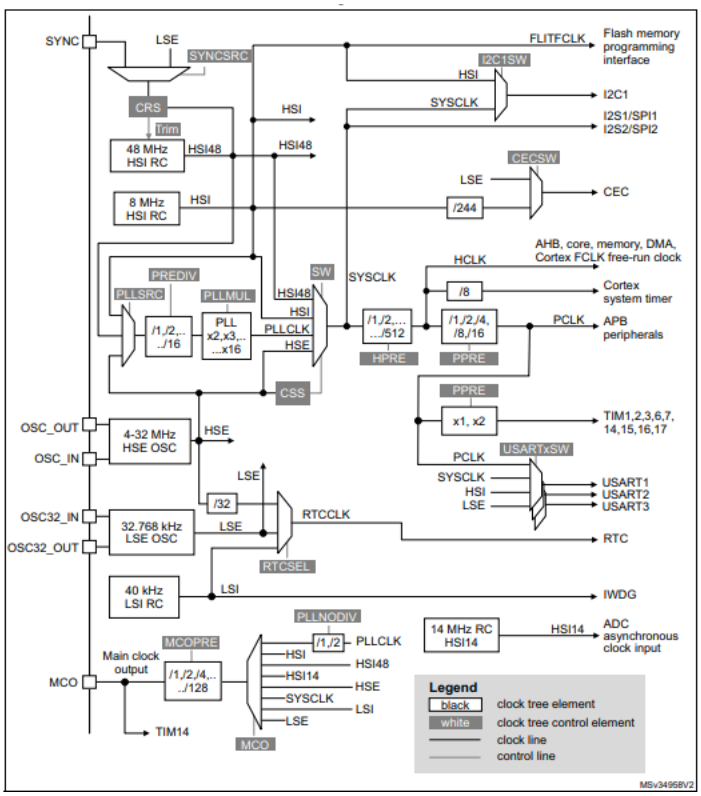

Figure 1. Clock tree.

For simplicity of explanation, the values that need to be set will be defined in reverse order, beginning at USARTxSW on the right side. Here, for USART5 (the UART port we are using), we need to set USART5SW to run on PCLK. Though, we found that there is no explicit field in RCC to modify this value for any USART peripherals other than 1-3. Likely, 3-8 are all corresponding to that of USART3SW, as many other functionalities are shared for those ports (i.e., interrupts). PCLK is the default value, anyway, and does not need to be changed. Following the PCLK line backwards, if we leave PPRE and HPRE as 0 (or no modification to the input value), we can set SYSCLK to be 48 MHz by setting SW to 2 (to select PLLCLK), PLLMUL to 4 (to multiply 8 MHz oscillator by 6), PREDIV to 0, and PLLSRC to 2 (to select HSE, for the external 8 MHz oscillator). These values were explicitly set, yet we still had to change our UART baud rate divider register, as we still ran on 8 MHz. Resolving this is on the to-do list; we can run all our necessary functions on MHz, but (probably) not my trilateration algorithm.

Microphone Array Trilateration:

Work on the trilateration this week brought both success and failure. My wiring setup worked properly, and I was able to detect a “delay” from each microphone to the bounce point. Usually, they were close; if I bounced the ball directly in a corner, on one of the sensors, that sensor was always at a “0” delay (as expected), and the catty-corner sensor had a delay of nearly

sqrt(2) * the delay of the other 2 sensors. Bouncing the ball in various parts of the table resulted in other similarly expected results. Directly in the center, nearly every delay was 0, as expected. However, these work “as expected”, most of the time. Sometimes, a bounce on the corner would yield 2 sensors with approximately sqrt(2) * the expected closer sensor delay. Unfortunately, I believe this means we are limited by our hardware, and no perfect software could correctly solve the location every time.

That said, I have still been porting my working Python software to C and will soon try an implementation of it on the microcontroller. Until that works properly, I will slowly add complexity to my new simple algorithm, which infers a location based on the delays, in a generalized way. Right now, it correctly determines which quadrant the ball was bounced in, by checking which sensor has the “0” delay and considering it closest to that sensor. This, however, has strange results when dropped in the center of the table, with a seemingly random choice of quadrant.

As our stretch goal states, we must infer the location within a 6”x6” square on the table, within a second of the bounce. As our table is almost perfectly 30”x60”, that means we have 2 sides of 30”x30”, where we can fit a grid of 5x5 6-inch squares. So, if my algorithm can be as close as picking a square out of 25, it will be considered successful. To chase that, I will begin adding complexity to the new algorithm, by adding a new row and column each iteration. As in, I will next test the ability to determine which of 9 equally sized and spaced squares the ball is in. Having an odd number will resolve weird situations in the center. Of course, there will always be uncertainty on the boundary lines, but if problematic in the center, it will be much more likely for the user to notice a discrepancy.

Next Week:

Next week, I intend to pursue completion of the following tasks:

1. Test ADC sample rate limits for proper user interaction.
    - Modify ISR priorities to better fit the needs of the program.
    - Perhaps after detecting a bounce, switch to no longer using event-driven handling, and instead collect all the sensor data in the main for loop. For this to work better, with the ADC interrupt no longer having precedence, we should disable all interrupts during data collection, then re-enable them when complete. This may also make our data collection more consistent!
2. Continue debugging trilateration algorithm.
3. Split my work into 2 repositories:
    - 1 to handle basic bounce detect code, cleaned, efficient, and ready for use on the PCB, in case trilateration does not work out.
    - 1 to store trilateration logic currently being implemented, and any other experimental microcontroller changes.