Sergy Pretetsky
cmpe121/L with Anujan Varma
Due 6/12/2014

Final Project

===Purpose==

Each of the students in the cmpe121 course will design a “connect 4” game on their PSOC 5 board that will display the game on an 8x8 LED Matrix and will connect with any other board to play against an opponent. The game will check if either player wins and give this information to the player. All of the player and opponent moves will be logged to an SD card to be seen at a later time.

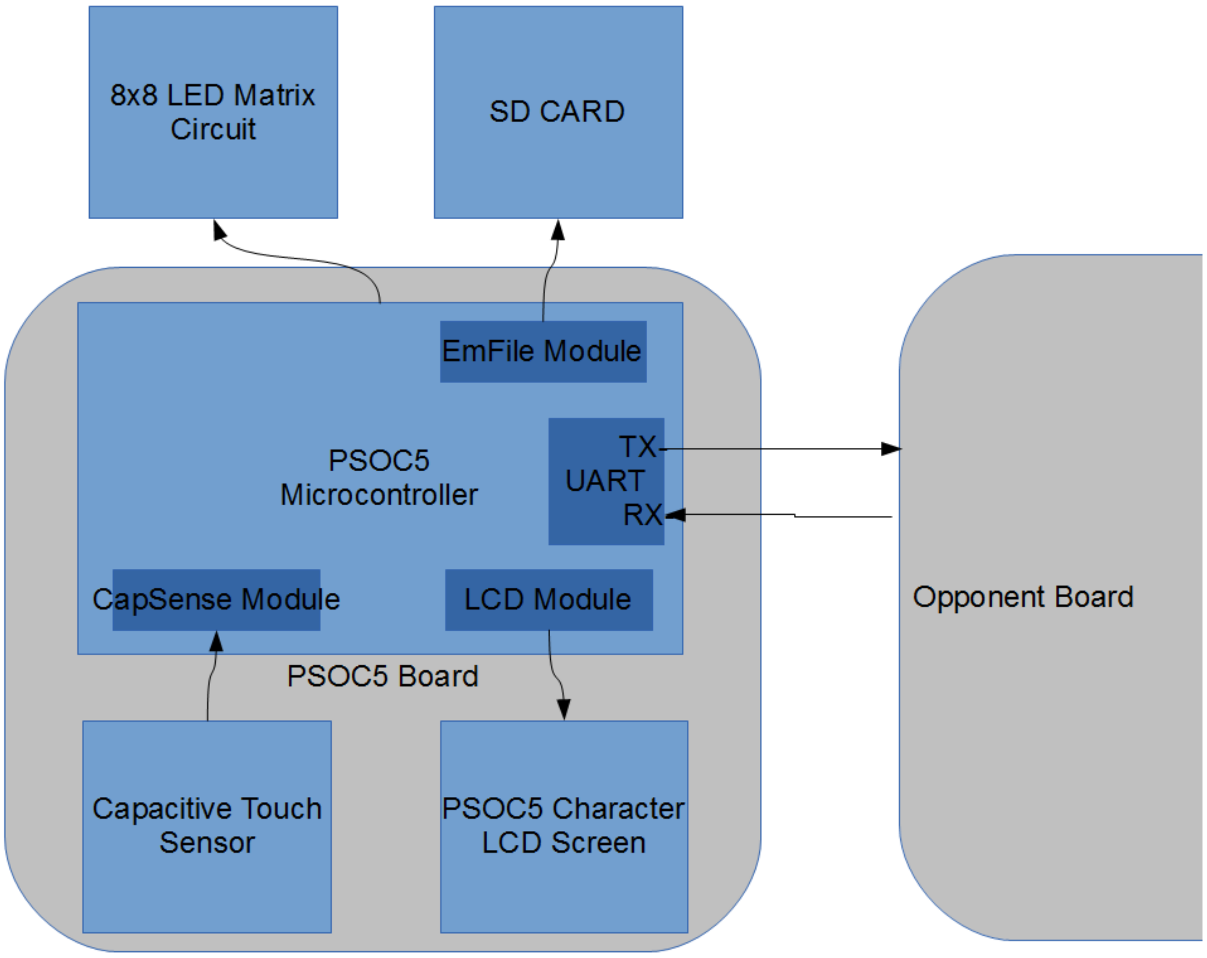
Our board will allow control using the capacitive touch sensors and will connect to other players directly using the UART module.

===Design ==

---------------Hardware Design--

Our hardware will be designed around the block diagram shown in Fig 1.

Fig 1.



The micro-controller will output the connect 4 display to our 8x8 LED matrix design using direct pins. We will get input from the on board capacitive touch sensor by using the CapSense module provided in the PSOC software. We will use the LCD character module to output the current state of the game to the LCD screen for the player to see. Communication with the opponent will be done using the UART module provided by PSOC. Communication to the SD card requires that we download the appropriate library from PSOC's website and use the emFile module. We will only use this to write data to the SD Card, not read data.

8x8 LED Matrix:

The LED matrix we are using has 8 common anodes for the rows and 3 cathodes per column, which means 24 pins for the 8 columns, which sums to 32 pins. Because of this, we cannot individually light up all of the LED's. Instead, we will “scan” the rows from top to bottom quickly enough so that it looks like a single image. We only need two colors for our connect four game, so we will only use red and green. In order to use less pins, we will not wire directly to the row and column pins. Instead, we will use 8-bit shift registers. One shift register will drive the rows, one shift register will sink the red columns, and one register will sink the green columns.

We are using the BL-M23B881RGB-11 LED matrix from Betlux. The lowest maximum forward current is 25mA (the red LED's). For simplicity, we will use this as the maximum for each of the LED's, regardless of color. The average of the forward voltage for red and green is 2V (red is 1.8, green is 2.2). We will power our LED matrix using 3.3V. The minimum resistance we need is:

$$Min\,Resistance=\frac{(3.3\mathrm{V}-2.0\mathrm{V})}{25\mathrm{mA}}=52\,\Omega$$

We have plenty of 330 ohm resisters. After testing using 330 ohms, we find that the LED's are still quite visible, and now we have plenty of safety margin as well.

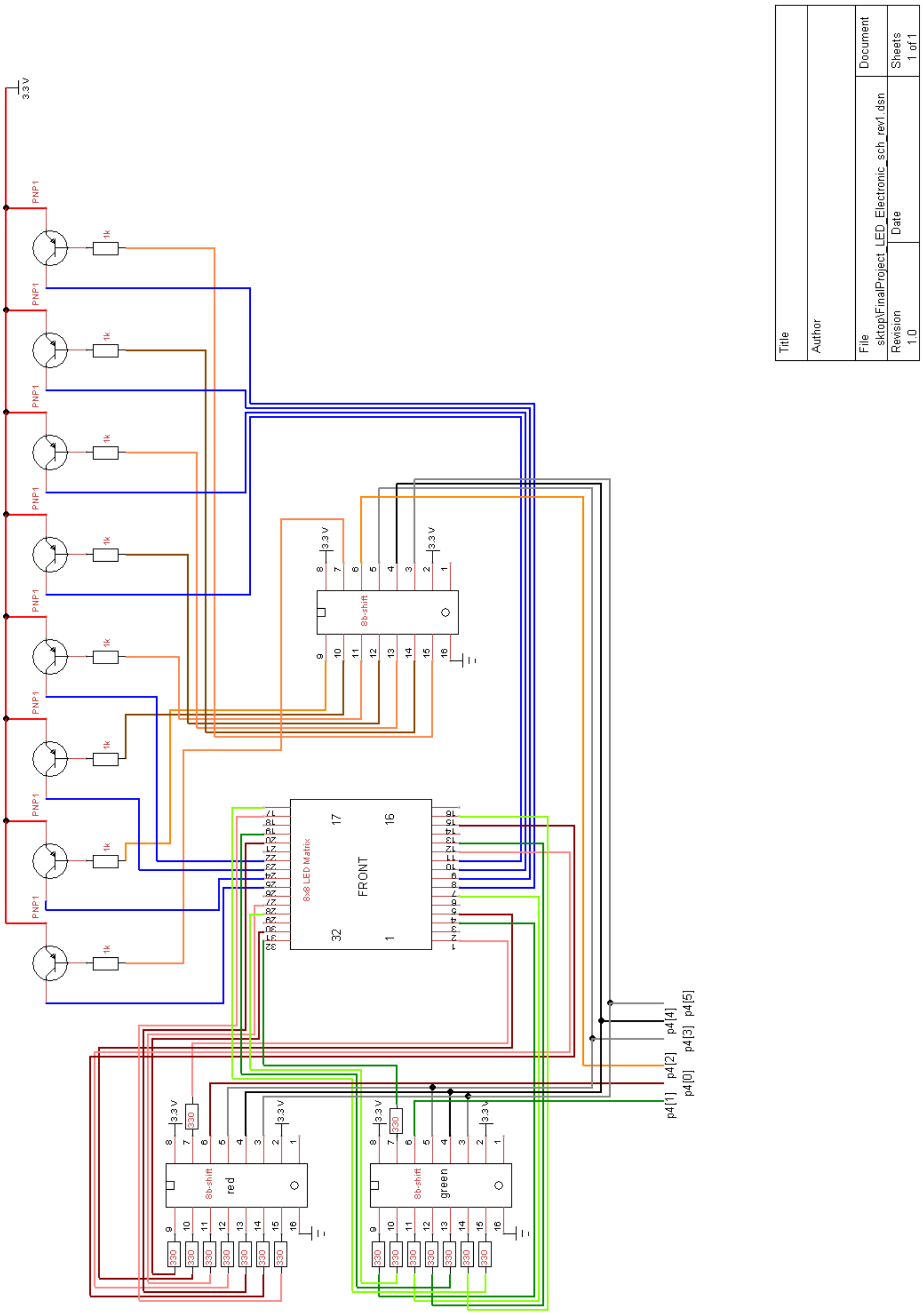
At 330 ohms, each LED will be running at

$$LED\,Current=\frac{(3.3\mathrm{V}-2.0\mathrm{V})}{330\,\Omega}=3.93\,mA$$

This means that an entire row can use up to 3.93 mA * (8 green + 8 red) = 62.8 mA of current. The 8-bit shift register we are using, the sn74hc595n, can only provide +/- 6mA of drive. We will, therefor, need to use a current amplifying transistor. We will use the PNP 2n3906-ap transistor. This transistor has a current gain of 60 when the collector current, I_C, is 50mA, and a current gain of 30 when I_C is 100mA. Our predicted current of 62.8mA is between these two boundaries, but we will assume the worst case scenario: the gain is 30. With a gain of 30, that means that when we are driving 62.8mA of current at the collector, we will need to sink 62.8ma/30 = 2.09 mA of current. This is well within the limits of our shift register. In order to limit the base current at the transistor we put in 1k resisters. This way, if we assume a worst case scenario of 3.3V appearing at the base, our base current should never exceed 3.3V/ 1000 ohms = 3.3 mA, well within our shift register's capabilities.

Our design will be active-low for both the row and column selection. The schematic can be seen in Fig 2.

Fig 2 (file can be found at ./report not finalized/FinalProject_LED_Electronic_sch_rev2.png



SD card:

Since we've already decided to run 3.3V, there is no need for us to do any voltage conversion for the SD card which also runs at 3.3V. We simply wire the SD Card directly to pins on our board. We use the following pin-out:

SD CARD HEADER	PSOC 5 BOARD
CD	GND
DO	P12[4]
GND	GND
SCK	P12[6]
VCC	V3.3
DI	P12[5]
CS	P12[7]

---------------Software Design---

Our game will be in one of five states: beginning, ourMove, opponentMove, weWin, weLose. In the beginning state either player can move to begin the game and we are listening for data to see if the opponent moves. If it's ourMove then our board doesn't send or receive any data and simply waits for the player to make a move. As soon as the player makes a move then we switch to the opponentMove state or weWin state, where we repeatedly send the opponent the move that we made and also listen for the opponent's move if we didn't win. If we receive a valid move from the opponent, then we switch to ourMove state or weLose state.

After either the ourMove or opponentMove state we check the last move made to see if it's a winning move. If the opponent made a winning move we switch to the weLose state. If the player made a winning move then we switch to the weWin state.

Display:

In order to operate our display we will need to create code to interface with the shift registers that drive the rows and columns of the display. We need to draw a single row on the LED display. Once every millisecond we will turn off the current row and switch to drawing the next row. We do this by first placing a timer in our top design that will go active once every millisecond and activate an interrupt. In retrospect, we could have simply made this a clock instead of a timer to use less resources.

The interrupt will drive an ISR routine called “refreshHandler” that will perform all of the display logic. We normally attempt to keep interrupts small and simple, and when necessary, have interrupts flip flags to inform our regular software loop to execute some complex code. However, because the display is very sensitive to timing issues, we do not have this luxury. All of the display code is fit into the ISR routine “refreshHandler” and priority is increased above all other interrupts. This guarantees that we have a smooth unflickering display. If I had designed this game again I most likely would have created a hardware or DMA based design so that I would never need to worry about timing

Inside the interrupt “refreshHandler” we turn off the display, sequentially load the red, green, and row number data into the shift registers, and then turn on the display again. Inside of this routine we also increment a global variable called “g_clock” which is used in any places that we need some sort of delay.

checkTimerMs(…);

As discussed at the end of the Display section, we have a global variable called “g_clock” that is used to create delays. This will be used in various places in our code where we want some sort of a pause, but do not want to stop executing our code. The way the checkTimerMs function works is it receives a 16-bit integer that represents some previous g_clock value and compares it to the current g_clock value. From this comparison it returns an integer representing the number of milliseconds that have passed between the supplied previous g_clock value and the current g_clock value.

CapSense:

We use the CapSense module and libraries. We we will use the “scroller” as well as the two buttons. It's important to set the sensitivity to an appropriate level for the buttons and the scroller. Otherwise the capacitive sensors will display unwanted behavior from the slightest interference. We use sensitivities of 4.

The top capacitive button will be a soft reset button. Originally this was created because it was noticed that doing a hard reset of the game will sometimes create unwanted behavior in the opponent's board. It was later discovered that this was due to very sensitive CapSense values. This feature is now not needed. The button detects if the user holds the button for at least 1 second and then resets the game states, the current display, and the SD card.

The “scroller” is used to scroll the top cursor, which points at the location where the player can drop a disc. We design a function called scrollCursor.

The design of the scrollCursor is interesting. We will implement a scroller that is familiar to tablet and smart phone users. When the player swipes the scroller, it will move the cursor a relative distance from it's current position based on how much distance was registered from the player's swipe. We lower the resolution of the capSense scroller to a much lower value because we will not need it and because we notice that the capSense scroller seems to have 6 Discrete “steps” that are easy to get to, and a finer spectrum in between that is very difficult to control. We will only use those 6 steps and so set the resolution to 6.

Anytime the scroller does not detect a nearby capacitive object it simply outputs all binary 1's. We use this knowledge in our design. If the program registers that the user has just touched the scroller, then we record the place where they touched the scroller and record the position of the current cursor. Now, anytime the user moves his finger on the scroller we record the displacement from his initial position and apply that displacement to the cursor's position. Once the user removes his finger we stop moving the cursor.

This method of scrolling feels very intuitive and also allows us to scroll the cursor through more columns than the resolution that our capSense provides (our capSense has 6 discrete easy steps we can control, but the game has 8 columns).

Finally, the bottom capacitive button allows us to drop a disc whenever it is pressed. It toggle's a flag called “drop” which another function uses to animate the dropping disc and determine where the disc lands. This button will only work in the game state beginning or ourMove, only if we're not already in a drop state, the button hasn't been pressed for at least two seconds, and only if there is space below the cursor to drop the disc.

DropMode:

Immediately following the button to toggle a disc “drop”, we have code that actually handles the drop. The code checks if the dropping disc has room below it and if some animation delay has passed by using the checkTimerMs(…) function. It then drops the disc one level down. If the disc has no room below it or has reached the bottom of the screen then it has landed, and we record that into the payload to transmit to the opponent, record it in our SD card, check if it's a winning move, and either switch to the game state weWin or opponentMove.

UART:

We will design our UART in such a way as to be non-blocking to the rest of the code. This means that we don't want to ever end up in a situation where we are waiting for the UART to transmit or receive data.

We put our main receiving and transmitting functions into a separate header and implementation file.

Both our transmitter and receiver will activate interrupts, however, the receiver interrupt is mostly for debugging. Both our transmitter and receiver interrupts increment an “indicator” variable that is used to flash the left onboard LED if we're currently transmitting and flash the right onbaord LED if we're currently receiving data.

The transmitter will interrupt whenever its FIFO is empty and the game state is opponentMove or weWin. At that point, our transmitHandler routine will flip a flag that tells our software that the transmitter is ready to send data. When our software checks the flag and finds that the transmitter is ready it calls the transmitPayload function declared in connect4UART.h. This function takes a payload array consisting of a player id, a row, and a column, and also takes a pointer to the function used for transmitting (UART_1_WriteTxData in this case) and then transmits the entire frame one byte each time it is called. It uses the starting sequence defined as 0x55 0xaa 0x04 0x00 and also calculates the checksum to add at the very end of the frame.

Since the function only transmits one byte a time it keeps track of its state using a static variable and also protects against switching payloads in the middle of a frame by copying the payload and calculating the checksum after it's done transmitting one frame and before it's transmitted the next frame. Because of this, it is completely safe to switch to a new payload without checking if the function has completed transmitted a full frame.

We only call this function once in our For loop and then wait again until the transmit interrupt tells us the FIFO is empty again.

One of the reasons we only transmit one byte at a time in our For loop is to give us easy control over how fast we send each byte. In case our opponent can't receive our data too quickly for whatever reason, we could easily implement a delay using our checkTimerMs() function and slow down the transmission of each byte. We did not find this to be necessary in our testing.

Our receiver, on the other hand simply polls the receiver to see if it's empty and if we're in the gamestate opponentMove. It will then attempt to empty the receiver FIFO, as long as we don't exceed 4 reads. This is just in case we keep receiving data that we do not spend too much in this part of our code. In every attempt to read the data we call a receiver helper function first. The receiveHelper function calls the receivePayload function which, similar to the transmitPayload function, requires a function pointer as a parameter (UART_1_ReadRxData in this case), an array to place the data, and receives only one byte at a time. The receivePayload function has three states, first it much match the first four received bytes against the intitial sequence (0x55 0xaa 0x04 0x00). If any of those matches fail it restarts checking at the first byte. After that it receives three bytes of payload data, which it does not check (the receiver Helper function will check those). Finally, it receives the checksum and checks that it matches. All of this data is being written to the supplied array, and if the final checksum check passes then the receivePayload function returns a 1 signifying that we have received a valid frame.

At this point the original receiverHelper function that called the receivePayload function checks to make sure that the opponent ID matches, that the row and column are valid numbers, and that this is not a move that was already made by the opponent. We check that last criteria by seeing if the row and column supplied by the frame is already taken by the opponent's disc.

If it passes all of those checks then we color in the display at the given row and column red, we log the move on our SD card, and we check if the move is a winning move. If the move is a winning move then we switch to the game state weLost, otherwise we switch to the game state ourMove.

isConnect4(...) and isVectConnected(...):

In order to check if four discs of the same color have been connected we use the isConnect4 function. This function takes a row and a column and checks if there is a vertical, horizontal, 45 degree diagonal, or 135 degree diagonal connection of four colors. It does this by calling isVectConnected four times with four different vectors and the same row and column.

isVectConnected will take a given row and column and remember its color (except black). It will then take the vector it was given, <x,y> or <row, col> and continuously keep adding it to the original column and vector counting how many times it finds a square that is the same color. Once it reaches a square that is a different color or it gets to the end of the LED matrix, it switches to subtracting that same vector from the original row and column and, again, counting each time it finds a square with the same color as the original row and column. Once it finishes, if it finds that at least four squares are the same along multiples of that vector, then it returns a 1. Otherwise it returns a 0. The four vectors we are interested in for the connect 4 game is <0,1> <1,1> <1,0> <-1,1>.

Wrapping up our main for Loop

Finally, at the end of our main FOR loop we make a call to gameStateCheck(). This function simply checks the current gamestate and displays appropriate text on the LCD screen. If it detects that the game is over (weWin or weLose) then it also shuts down the SD card for any further writing.

===Testing ===

--------------LED Matrix---

The LED matrix was first built and tested on a breadboard. After it was tested at 3.3V and it was determined the LED's were clearly visible then the electronic schematic was created using tinyCAD software and the components were soldered to the prototype solder board.

The program used to test it can be found in ./testProjects/LEDTest.c. The program would draw random colors at random spots on the board with some delay. After a certain amount of time the whole display was cleared and the process began again. This allowed me to see individual colors and check to see any pixel ghosting was visible.

---------------CapSense---

The capacitive sensors were tested using the program in ./testProjects/CapTest.c. The program simply gets the value from the capacitive scroller and prints it out onto the LCD character display. It was in this test program that I realized that, without getting into more serious tuning, the scroller was not smooth, and had 6 discrete steps that were easy to register and many regions in between those 6 steps that were very difficult to control. This was part of the motivation for me to ditch any form of direct control method and instead create a more intuitive "tablet-like" experience.

Long after I integrated the CapSense module I realized that every time my opponent reset his PSOC board it would cause my board to do erratic things. It was not a big priority for me at the time, so I didn't know what the problem was for some time. When the time came to isolate the problem I realized that every time my opponent reset his board it affected my capacitive touch buttons. I changed the sensitivity from 2 to 4 and the problem went away.

---------------UART---
The UART was tested using the program in ./testProjects/UARTTest.c. This program would simply keep checking for a particular byte (in my case 0x55, the first byte of our frame) and then display the next 7 bytes after that without discrimination. This allowed me to see if I could match at least the first byte with people and then display the rest of the frame regardless if the rest of the frame was erroneous.

Simultaneously, the program transmitted some data frame that began with the correct first 4 bytes but then transmitted some incrementing data that wasn't necessarily valid.

After I integrated the UART into my project I had numerous small bugs that I had to fix and the receiver went through various iterations. At first the receiver was interrupt driven just like the transmitter, but at some point I realized there were many strange behaviors associated with this setup that I, unfortunately, did not have time to fully understand. I decided that there would be no real performance penalty with simply polling the receiver in my for loop. However, I still kept the receiver interrupt as it was connected to the receiver indicator LED which was very useful for debugging.
---------------SD Card---
There was no separate testing phase for the SD Card. My SD Card was integrated near the end of my design and was simply coded into the main project. Pins were soldered onto the SD Card header and it was attached to the small prototype area in the middle of PSOC board

===Lessons learned ==

1) While driving the LED display using software is fairly easy, it did introduce another issue. Timing for the LED Matrix is very important, otherwise LED's may appear darker/brighter and also flicker. There were several times when I had to change the way the LED's were controlled throughout the creation of my project. In a more complex project this might be even harder to control. I would strongly consider doing hardware LED control in the future.

2) It seems pretty obvious now, but when I connected the TX and RX connections between my board and my opponents board I got random data and errors that made no sense. Using a common ground between a transmitter and a receiver is important. Connecting a common ground fixed my issues.

3) Building some sort of debugging systems directly into your project can be very useful. I helped many people (including myself) by making my UART display transmitting and receiving data (even if it was invalid) directly to the LCD character display. I also found my transmitting and receiving blinking LED indicators very useful in debugging my project as well as others' projects.

4) The Fritzing software, although workable, was difficult for me to use, and had many features that I didn't need. The hardest thing for me, in Fritzing, was creating custom components that didn't already exist. I couldn't figure out, for example, how to create an 8x8 LED Matrix with the same pinout as what we had.

I instead found an open source, very small, and simple schematic software called “tinyCad”. TinyCad was extremely small (I don't have much hard drive space) and allowed me to very easily and simply create an 8x8 LED Matrix component with the correct number of pins and also create an 8-bit shift register.

===Conclusion ===

We designed a “connect 4” game on our PSOC 5 board that displays the game on an 8x8 LED Matrix and connects with any other board to play against an opponent. The game checks if either player wins and it give this information to the player. All of the player and opponent moves are logged to an SD card to be seen at a later time.

Our board is controlled using the capacitive touch sensors and connects to other players directly using the UART module.

===References ==

1) 2n3906 transistor *http://www.mccsemi.com/up_pdf/2N3906(TO-92).pdf*

2) sn74h595 shift-register *http://www.ti.com/lit/ds/symlink/sn74hc595.pdf*

3) Bl-M23B881RGB-11 8x8 LED Matrix
http://shop.emscdn.com/im/meggyjr/BL-M23B881RGB.PDF