6.1151 Final Project Report

David Ologan

January 11, 2023

1 Introduction and Background

This 6.1151 final project aims to create an assortment of games that would run exclusively on the PSOC5 Big Board. Each game would require different functionality from the wireless controller that I would also create using the PSOC Stick. The two devices will run separate programs but operate concurrently, relaying important information and providing a smooth UI for the user in each of the games. Given how commonly the 8051 was used to cheaply implement a variety of game controller currently on the market, the game controller was initially going to be run using the 8051. However, after contemplating the addition of the II portion of the project, as well as continuity, the project proceeded with the PSOC stick instead. Tentatively, the software side of this project involves the creation of three distinct games; Tic-Tac-Toe, Pong, and a personal interpretation of Duck Hunt.

The bulk of the software in this project will be coding each of the three games in C. The three games, Tic Tac Toe, Pong, and a version of Duck Hunt, would be single player games, run off the PSOC Big Board and the connected display. A TFT display was to be used as the main interface that the user would interact with, however, after speaking with multiple TA's. the potential of using VGA to connect to one of the lab monitors took the forefront and became the direction of the project. At startup, the player would be prompted to press a button to select one of three games, and a desired difficulty level. Afterwards, the game would proceed, allowing the player to progress to more difficult levels given their completion of game objectives.

Tic-Tac-Toe is self explanatory for the most part. The game would default to the computer going first, prompting the user for a response. After each game, the board would automatically reset, giving the player a fresh grid to play on. Regarding the Pong game, the game would take the form of almost table hockey, with the player trying to break bricks and keep the ball out of their own goal. Finally, the most involved game would be the Duck Hunt Style Game. The general idea is that the gain would have some set of randomly moving targets translating across the screen at a given speed. The player would have to move a cursor with the joystick and ”shoot” each target before it passes the boundaries of the screen. As the levels progress, the speed of the targets double, and their paths change, making the game more unpredictable and difficult. After hitting a target, a explosion animation would show up, letting players know that they’ve successfully hit their target.

The project’s controller would be relatively simple, built in the model of a PS3 controller seen below. With four buttons and a joystick, the various buttons would be useful for Tic-Tac-Toe while, the joystick would enable functionality for the shooter and pong games.

This project also explored a variety of wireless technologies, including RF modules and the HCO5, HC06 Bluetooth modules. In an effort to make the controller wirelessly communicate between stick and big board, these modules will allow for the encoding of button states critical for determining cursor position and game states. Reach goals for the project include creating a second controller, adding additional levels or games, or the creation of two player games.

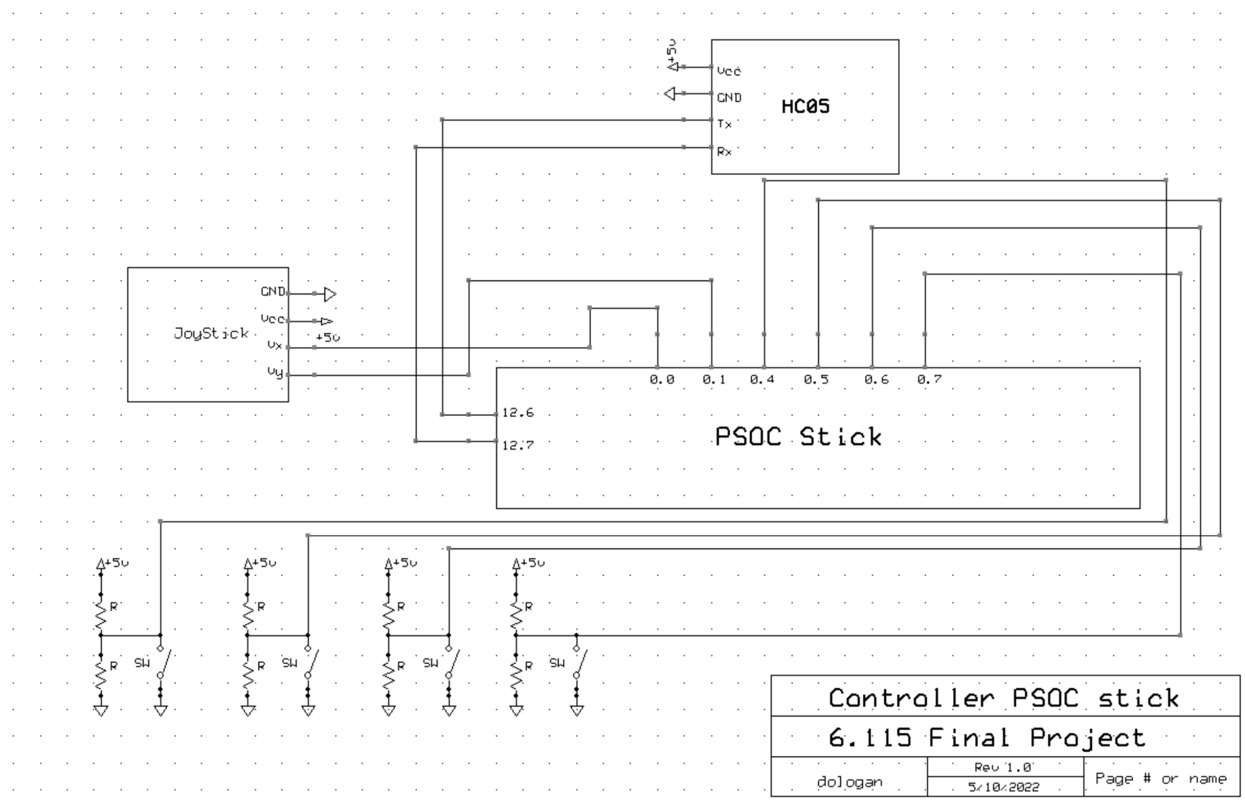
2 Hardware Description

The majority of the hardware for this project can be split into two primary parts. These include the wireless controller powered by the PSOC stick and the Receiver/VGA Connector attached to the PSOC5 Big Board.

2.1 Wireless Controller

The Wireless Controller was created using four buttons, and a joystick potentiometer from the ELEGOO kits. The buttons were wired active low using a simple voltage divider circuit seen below. The joystick potentiometer and the 4 buttons were wired to the PSOC stick, which would encode the button states and transmit them over Rx/Tx pins connected to the HC05 Bluetooth modules. Two HC05 Bluetooth modules were used, with the controller equivalent in MASTER mode and the Receiver equivalent in SLAVE mode. The Serial communication was set to 9600 baud in a process described below. The entire controller was run off the PSOC stick's power source, but the HC05 was programmed using a 3.3V power source generated by a LM317T.

2.2 Wireless Controller Schematic

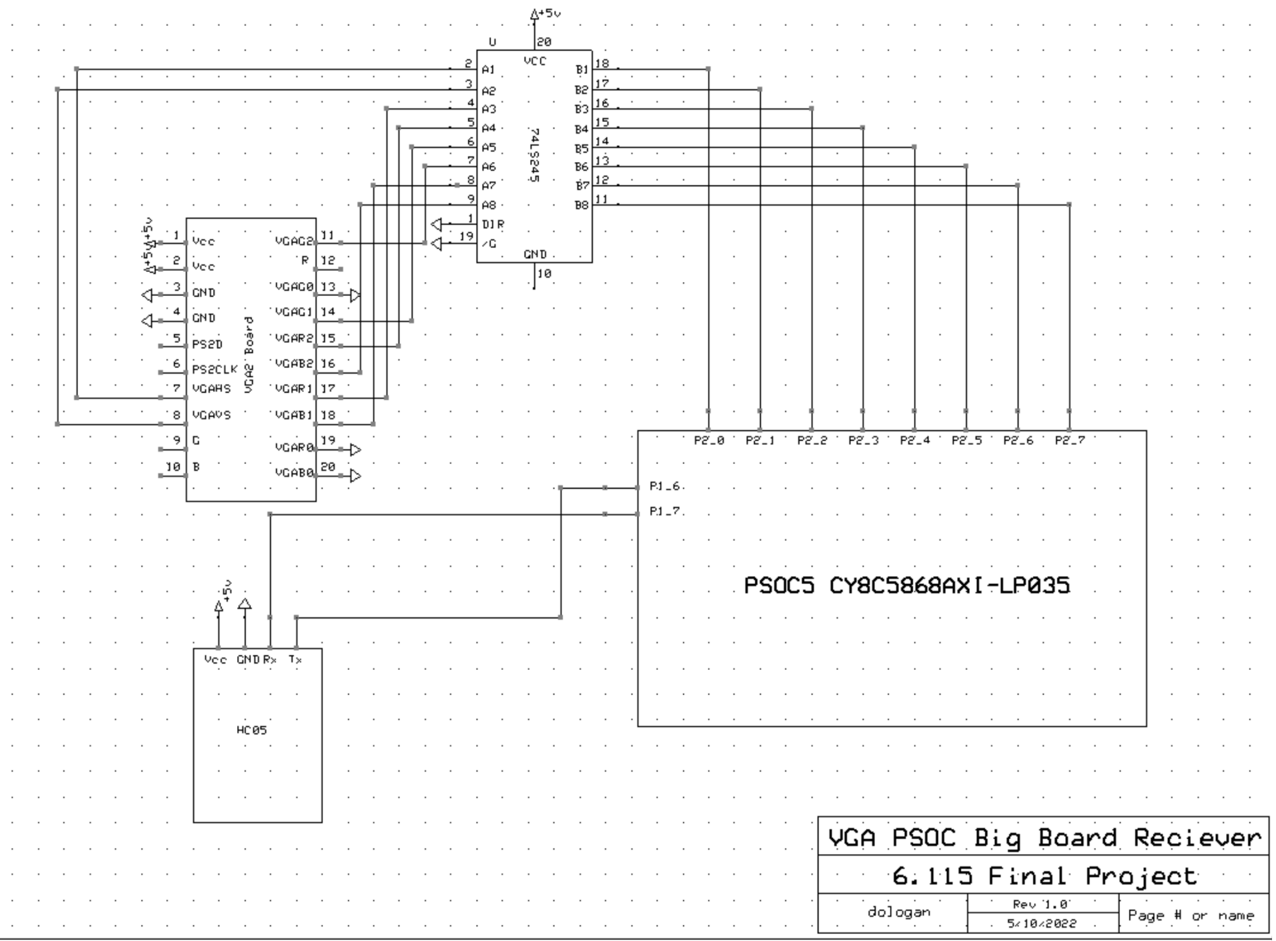


2.3 Receiver/ VGA Connector

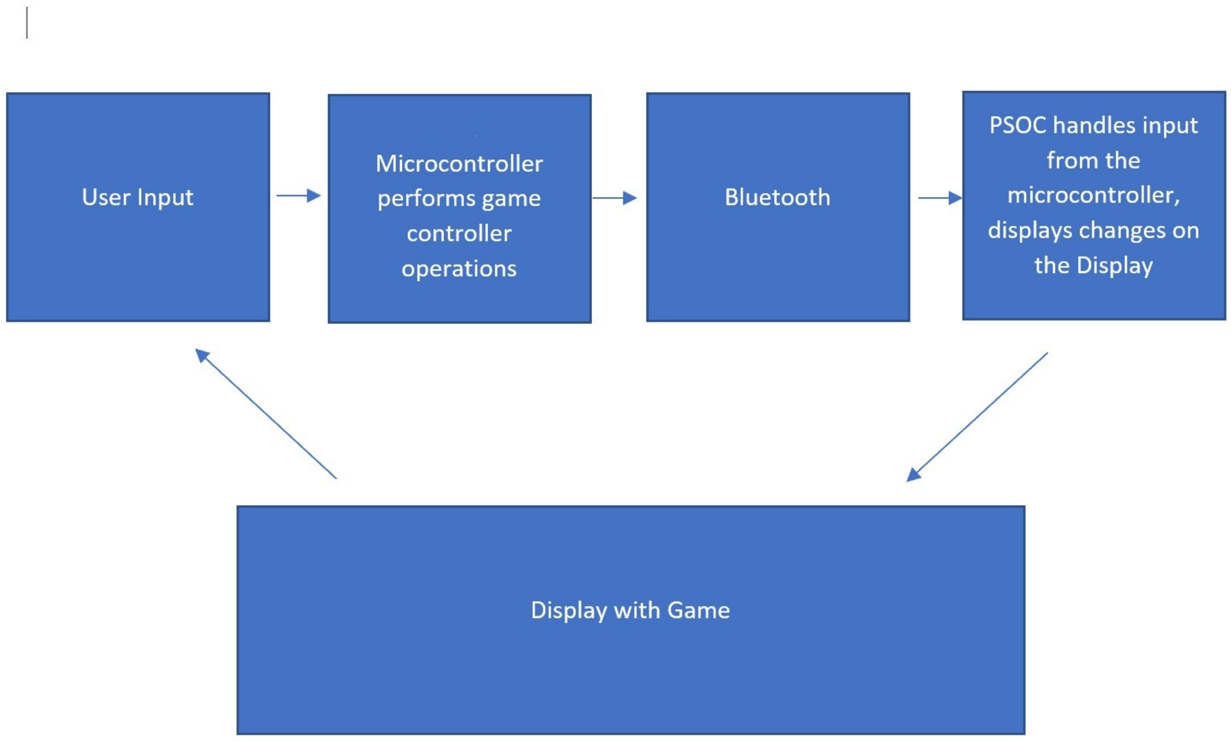
The general setup of the Receiver is similar to the Wireless Controller with and HC05 in SLAVE mode wired to Rx/Tx. However, the addition of the VGA2 Connector complicates the design. The VGA2 connector was chosen primarily for its quick refresh rate. Rather than using serial to send frames, this project uses the PSOC to update the buffer on its own, speeding up the process, and allowing us to achieve an acceptable frame rate for our game. By editing the buffer directly, and then using the PSOC to send these frames to the monitor, we can have relatively seamless animations, and response time, something especially critical for the shooting game, where the relative positions of objects on the screen are important. The VGA2 Board is wired for 2-bit color for simplicity, and is connected to an 74LS245N. The low byte was disconnected as a result leaving VGAR0, VGAB0, and VGAG0 grounded. The PSOC5 cannot provide enough current to sufficiently drive the monitor signal, so in order to avoid weak looking signals, we bump up the current with this chip. From there, the LS245N is connected directly to the PSOC Big Board at Pin 2. An LCD display was also added to the Big Bard for debugging purposes in an effort to check the accuracy of the incoming Bluetooth signal. Once again, this circuit was run on 5 volts, and the HC05 was programmed using an

3.3V signal.

2.4 Receiver/VGA Connector Schematic



3 Software Description


User Input
Microcontroller performs game controller operations
Bluetooth
PSOC handles input from the microcontroller, displays changes on the Display
Display with Game

3.1 Button State Encoding

Each of the games I planned on building relied heavily on the need for the controller to dictate what happened on the PSOC Big Board. As a result, if bluetooth was to be used as a means to communicate between these two chips, a way to encode potentionmeter states, as well as button states into byte sized quantities that would be sent between the chips using bluetooth was needed. However, with 4 buttons and a Vx and Vy from the potentionmeter, compiling all that information into one byte became difficult, especially if the resolution of the potentionmeter was to remain viable. As a result, three distinct bytes were sent instead of 1, the first two bytes describing the Vx and Vy of the pot, and the third dedicated solely to the button states.

In order to do this, quick test code was written to see the range of outputs provided by the potentiometer extremes. By displaying the potentiometer values at pins Vx and Vy, visualizing the outputs became simple. After doing so, the potentiometer outputted a value of 5 at its minimum and 819 at its maximum. Noting this, the potentiometer outputs were divided into 9 distinct regions, ranging from 0 to 900. Rather than sending the raw pot value at a resolution way to high to be noticed by the user, the values were categorized into each of these regions 0-100, 100-200, etc. From there, each region was mapped to a specific ASCII character that would be sent to the PSOC Big Board. This process would be repeated for both x (A-I) and y(I-R) directions, allowing us to scale the cursor speed from 1 to 9.

To encode the button states, a similar encoding process was used. Button On states and off states were mapped to specific ASCII characters, by doing so, one character was enough to describe the state of all the buttons.

After completing the encoding, information about the state of the controller was condensed into a 3 digit word composed of three distinct ASCII characters. This would be sent over bluetooth and decoded in the ISR of the Big Board. After decoding, cursor positions, and button states could be updated to accurately reflect the intentions of the player. The full encoding are seen in the image below.

4 Bluetooth ISR

For this project, the Big Board was left with the brunt of the computing required for running the game. Its responsible for updating game states, receiving inputs from the controller, as well as updating the screen using VGA. Running all of these parallel tasks, proved difficult, especially since the slightest bit of blocking in the code would be immediately noticeable to the user (the screen would no longer refresh).

In order to solve this issue, interrupts were used to update the screen and to sample the bluetooth inputs. In order to avoid priority issues, the screen update ISR was placed at a higher priority than the Bluetooth ISR, since users would more likely notice a failing screen than slight unresponsiveness in the controller. Furthermore, the Bluetooth ISR operated with a much slower clock. Essentially, the bluetooth interrupt, which operated on a 60 kHz clock, was orders of magnitude slower than that of the VGA Interrupt. Initializing them this way, allows us to update the screen more quickly than we check for bluetooth values. Its still too fast for the user to notice and lag or drop off in responsiveness, but we avoid issues with the display cutting out (the display would cut out when it decided to switch tasks). Slowing down the Bluetooth ISR also allowed for a less responsive controller, which made it easier to handle, a bonus given that the joystick is incredibly sensitive.

Within the ISR, a character array, response, is generated to hold the three byte input. It waits for exactly three bytes in the RxBuffer and transfers them to response. Using a subroutine called decode, the three bytes are used to update critical global variables like the cursor's x and y velocities, and the current button state. Finally, the ISR ends but lowering the interrupt flag, and calculating the cursors new x and y position as a function of the updated velocities.

5 VGA ISR

The VGA ISR operates similarly to that of the bluetooth one. Although the VGA ISR is triggered more frequently, it takes the current state of buf, an array that is updated in the main with objects from the game, and copies it into vbuf. In doing so the changes we made to buf are reflected on the screen, and the next frame that we designed is uploaded to the monitor for the player to interact with. This ISR was ranked a priority 0, since keeping the refresh rate of the screen up is important, in order to avoid screen wiggle, and distortion.

6 Duck Hunt

6.1 General Structure, Main

The biggest obstacle when generating frames, was figuring out how to construct animations. Essentially, simulating the motion of a target or a cursor moving across the screen. At first the easiest solution to this would be to have set animations that were stored as fixed constant arrays. Essentially, predefined paths would be set for each target and the image of the target would be projected along that path, in buf. However, the PSOC has limited memory, and that strategy wouldn't work for a game like duck hunt, were the relative position of objects is necessary. As a result, an alternate strategy where, one would keep track of center point of each of the targets and the cursor. These center points would be updated in time and then the image of each target would be constructed around these center points in buf. By approaching the problem this way, identifying when the cursor is on top of the target becomes as simple as comparing their center locations.

As a result, the main for the duck hunt can be divided into 10 steps. Firstly, a blank screen is copied from memory and used to populate buf. Next, the current level is displayed in the upper left hand corner, and the precise position of each of the targets is calculated. Subsequently, a boundary check is performed to keep targets in bounds, and a hit target subroutine is run to see if the player has hit any targets. Finally, we add the image of the cursor, and remaining targets to buf, check to see if a level change is necessary, and update the screen. An explosion animation was also added to let players know when they've successfully hit a target. This was added later, and would added to buf before the update. Each of these subroutines will be discussed below.

6.2 Print Level, Print Targets

Print Targets simply checks each targets state to see if its been hit, and if not, draws the image of the target around its center point index in buf. Similarly, Print Level, just checks the current level, and accesses memory to place the current level in the upper left hand corner of the display. It also adjusts the velocities of each of the targets to their respective quantities. It doubles the speed of certain targets to make the game more difficult at levels 2,3 for example.

6.3 Calculating Target Positions

Target Positions are calculated based on their current position and their velocity. Since each target travels at a different velocity and these velocities can be positive or negative we want to add the targets velocity to their current position to reflect their new position on the frame. However, doing this on every update of the screen causes the targets to move super fast across the screen. At the very least much more quickly than a person could feasible hit.
As a result, we use a counter to update target positions every 5th refresh, allowing the targets to move at a reasonable speed across the display. Additionally, by keeping the targets at a set position on the screen for a given amount of time, it gives time for the player and the PSOC to register hits, and act accordingly.

6.4 Boundary Check

Since each of the target's x and y positions are defined as unsigned 8 bit integers, there exists a region of space that the targets may appear outside the confines of the screen. Basically, the screen dimensions are 128 in width and 96 pixels in length, but the targets x and y positions range from 0-255. If we're adding a velocity to the x and y positions, the possibility exists that the target positions exceed the bounds of the screen. In boundary check, a loop is created so that if the x position exceeds 128, it automatically resets the x to 0, allowing it to continue its path on the other side of the screen. This greatly improved the progression of the targets on the screen and stopped them from getting stuck off screen. With nearly 3/4 of the unsigned int space inaccessible, leaving the boundaries as is would be frustrating for the player.

6.5 Hit Targets

Since aligning the one center pixel of the cursor with the one center pixel of a given target is nearly impossible, adding a range of effect improves play ability. By padding the area of effect or shooting range of a given player, its easier for players to hit targets, and even if they are a little off, we compensate by giving them the target anyway. Assume the player is shooting something akin to a shotgun with a little bit of spread. After testing, it appeared that a 5x5 square around the center point provided the best playing experience without making the game too easy.

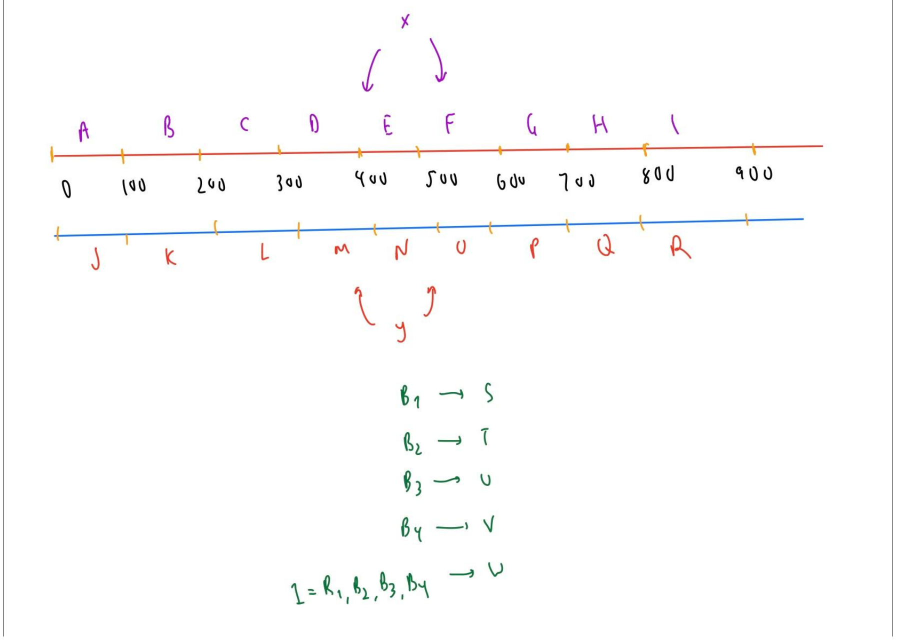
After defining the range of shot areas, we compare each targets x and y position with the x and y position of each pixel in the shot area. If there is a match, and the button has been pressed, we update the remaining targets array and initialize the explosion animation. By editing the remaining targets array, we no longer display that target amongst the remaining targets.

6.6 Game State

Game State allows us to check how many targets are still in play. As currently designed, at each level,the game releases 3 targets, but as the levels change their velocities and starting points change, which in turn impacts the path they take across the display. Game state checks to see if there are no active targets, and if so, advances the level and resets the game state to have 3 active targets. Print state and print level take care of readjusting the velocities and the level indicator in the corner.

6.7 Decode

Despite not being the shortest routine to include in an ISR, decode takes in the 3 digit transmission from the controller. Using cases, it iterates through each character and checks each value. For instance a character from (A-I) will update the cursor x velocity to the right quantity. Interestingly enough, the resolution of 9 ended up being much too high, with the controller being too responsive and difficult to control on precise short movements. As a result, all of the velocities were halved. For instance an A, which would represent the joystick all the way to the left, would typically set the cursor velocity to 4. However, with this adjustment, the joystick velocity is now 2. With this, the user has better control of the cursor.



6.8 Print Explosion

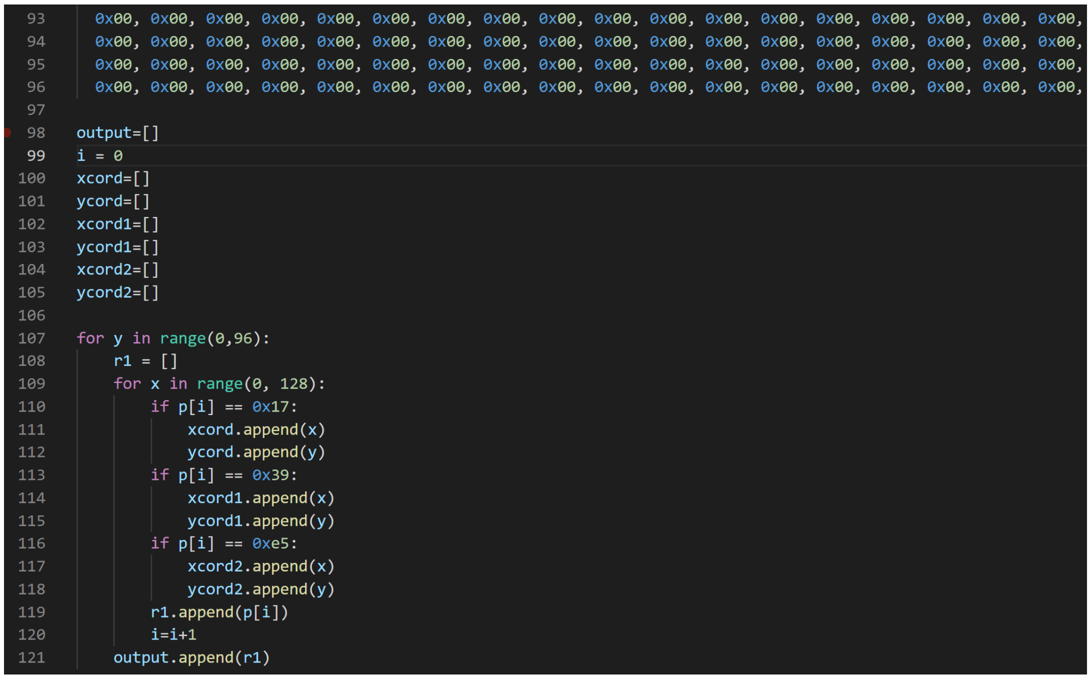
Animating the explosions proved to be quite difficult. Much like the print targets scenario, if the explosions occurred to quickly, players might miss the explosion. We also want the explosions to be a temporary event. Unlike the targets which either always exist or don't, the explosions should only be visible for a set amount of time. Additionally, the explosion can't be saved in between refreshes, so the animation needed to be cleared and happen fast enough to not bog down the code.

This subroutine checks to see which targets have been hit, and if the explode marker has been risen. If so, we update buf with the explosion animation, and decrease the explode marker. Having the explode marker double as a counter helps here, since we can use it to dictate how long we want the explosion to appear on the screen before dying out into the background.

6.9 Miscellaneous/ Start Screen

The start screen and initialize screens are preset arrays saved in memory. They're called and followed by an infinite while loop, allowing us to change screens and start the game when the player presses any button. In order to obtain the preset start screen array, the image was drawn in Paint, with the dimensions fixed to the display dimensions. Afterwards, the file was saved as a bmp and converted into a C array. After obtaining the C array, the array was sorted by color, in order to extract index pairs of pixels with a specific color. Saving these positions to an array in permanent memory, it was then possible to update buf directly from memory.

Overall, that summarizes the key components of the Duck Hunt Game. The game has three total levels, and once complete, ends the game by raising the current level to 4 and triggering the exit protocol.



```
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

output=[]
i = 0
xcord=[]
ycord=[]
xcord1=[]
ycord1=[]
xcord2=[]
ycord2=[]

for y in range(0,96):
    r1 = []
    for x in range(0, 128):
        if p[i] == 0x17:
            xcord.append(x)
            ycord.append(y)
        if p[i] == 0x39:
            xcord1.append(x)
            ycord1.append(y)
        if p[i] == 0xe5:
            xcord2.append(x)
            ycord2.append(y)
        r1.append(p[i])
        i=i+1
    output.append(r1)
```

7 Tic Tac Toe

7.1 General Structure, Main

The general structure of the Tic Tac Toe code is loosely based on the framework developed in for the Duck Hunt Game. Using the same set of interrupts and encoding for the controller, Tic-Tac-Toe's main is the sum of 6 subroutines. The program begins by clearing the display, with the start screen read from memory, Then the primary grid is drawn on top of the blank screen. Check Game State is called, which allows players to make a move. The program then checks to see if the move resulted in a win in Check Win, and updates the display with the symbols that have been played. Finally the cursor is added, and the updated buf is sent to the display.

7.2 Print Grid

We begin by dividing the available display space into a 3X3 array. To do so, we draw four lines , 2 vertical, and 2 horizontal and color them white. Since these are fixed positions, we hard code the constant coordinate, and iterate through the dimension of the display space. (i.e a vertical line at 6, always has an x coordinate of 6, but a variable y coordinate). After updating buf with these pixels, we thicken the lines on either side to improve their visual appearance.

7.3 Check Win

Given that there are only 8 possible ways to win in a game of Tic Tac Toe, namely three horizontal, three vertical, and the two diagonals, the subroutine iterates through the game state array to check to see if any of these arrangements have occurred for all 'X' or all 'O'. If so, update the winner, end the game, and stop letting players take turns. These were hard coded in as conditions, that would be checked every time the function was called.

7.4 Print Symbols

After going through the process of updating the game state, print symbols, takes the game state and projects the necessary symbols in the correct spot. At the start of the program, as we partitioned the display into a grid of nine spaces, we also assigned a point at the center of each of those spaces. We then build the desired shape around that center point, so a 'O' or 'X' appears perfectly center each time. We update buf with all existing 'X' and 'O', return to main, and eventually update the display.

7.5 Check Game State

The bulk of the code for Tic-Tac-Toe happens in this subroutine. It begins by determining whose turn it currently is, and acting accordingly. If its the computers turn, it randomly selects a free space on the grid, populates it, and updates the game state array. It reassigns turn, and exits. On the other hand, if its the player's turn, we wait for a button press, and when triggered, we take the current position of the cursor, determine which of the 9 grid spaces it currently resides, and checks to see if there's an existing mark in the space. If not, the player is allowed to play in that spot, and we update the game array and reassign the turn. As a result, the player can choose where to place their mark by navigating the cursor to that position and pressing the button.

8 II Component

Bluetooth commonly refers to short range wireless communication between devices. Bluetooth uses a 2.4 GHz frequency and typically generates a 10m wireless network that allows it to connect devices. Since it operates a lower power, it avoids the drawbacks of technologies like RF, and Wifi. It suffers less from interference but is slower than Wifi. For my application, Bluetooth seems to be sufficient, especially given the fact that its well documented, has been Incorporated into the PSOC software, and I don't necessarily need the speed that Wifi provides in order to send 3 bytes. Furthermore, if I'm only sampling once every 60 MHz, the 2.4GHz is plenty for my application.

8.1 Configuring the HC05 Chip

After obtaining two HC05's from the EDS, it became necessary to set one to MASTER and one to SLAVE mode, as well as set a bunch of other settings like baud rate, etc. To do this, I connected directly to the HC05 using a USB to ttl connector. By bringing the chip enable high (3.3V), the chip transitioned to program mode, where it was then possible to communicate with it through serial.

After connecting to the HC05, the command $AT + ROLE = 1$, to assign the HC05 to MASTER mode. The opposite was done with the other HC05($AT + ROLE = 0$). Afterwards, the HC05 was set to mode 1 ($AT + CMODE = 1$), which allows it to connect to the other HC05 in the vicinity. $AT + UART = 9600$ helped set the baud rate for both HCo5's. Resetting with $AT + RMAAD$, $AT + ROLE = 1$, AT+RESET was necessary, before using $AT + INQM = 0, 5, 9$, $AT + INIT$, $AT + INQ$ to find the other available HC05. Finally, $AT + PAIR =< addr >, < timeout >$ and $AT + BIND =< addr >$, was used to bind the two together once the connection was established.

Some code was used to verify the connection as well as determine what bytes were being sent between the two modules. See below.

8.2 Implementation

As a result of time constraints, the RF Transmitter Receiver purchased for the project wasn't adequately explored. In theory, the connection process would've been more or less identical to the HC05's. The pin out, is basically identical with a Rx/Tx pin, Vcc, and GND. The only difference being they transmit on a set frequency, and one half is fixed to MASTER and the other to SLAVE mode. Given that the pin out is identical, transmitting data would've working in the same way, albeit I would've had to adjust the sampling rate.

To include the HC05's, UART blocks in the TopDesign of both the controller and the receiver were added. After pairing and linking the two devices together, the baud rate of both the UART blocks was set to 9600. The RxBuffer or the TxBuffer would get check and the PutChar or GetChar command was used to send or receive a byte of information.