*Pong: Digital II*

*Recreation of the classic video game using an MSP430F5529*



Course: ECE 09342 – Dr. Tang

Group Members: Traz Daczkowski, Christian Heimlich

Project Number: 3

Due Date: 12/17/2015

Date Turned In: 12/17/2015

# **I. Introduction**

The purpose of this final project was to demonstrate an understanding of the C programming language as well as knowledge of the MSP430 and its hardware limitations. Additionally, this was the first instance of students creating their own goals. It was chosen to recreate the 1972 video game Pong seen in Figure 1. The premise of this game is similar to that of the real game of Table Tennis (also known as Ping Pong, the name that led to the name of the video game) in that there are 2 players controlling a paddle with a ball between them. The objective of the game is to prevent the ball from reaching the left or right terminus of the screen by deflecting it with your paddle. If the ball does reach the edge of the screen with and your paddle is not there to intercept it then the other player scores a point. The game ends when a player reaches the specified number of points, which in this instance is five. To prevent a stalemate the ball speed gradually increases until a point is scored, at which point the ball will reset to the center of the screen with the original starting speed. Another key facet of the original game that needed to be emulated is for the ball to bounce in different directions dependent on where the ball impacts paddle.

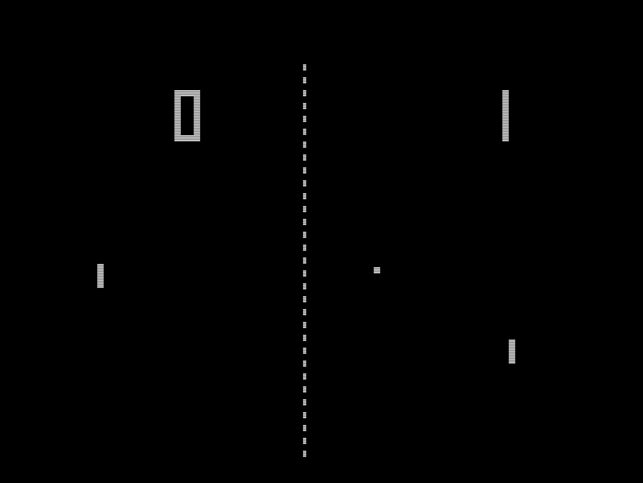


Figure 1: The original Pong Game

By emulating the original pong utilizing an msp430 as well as outputting a display of the game to an LCD display, it is shown that there is an understanding of the course material as well as the ability to apply it to a real world application. This is due to the multitude of concepts applied to this project, including interrupts, serial communication, port mapping, register manipulation, PWM, etc.

**II. Design Specification**

The game runs on a system that is composed of a hardware component and a software component. Both components are essentially black-boxed to each other, because each piece of hardware is unidirectional in terms of communication, so that only inputs and outputs are exchanged. Additionally, all I/O logic was handled via software and no hardware communicated directly. These properties made designing and organizing the program a lot easier from a high-level perspective. Essentially, solely the software component of the program runs the game logic, while the hardware component is what allows the players to interact with the game logic. A general overview of how the program functions can been seen in Figure 2.

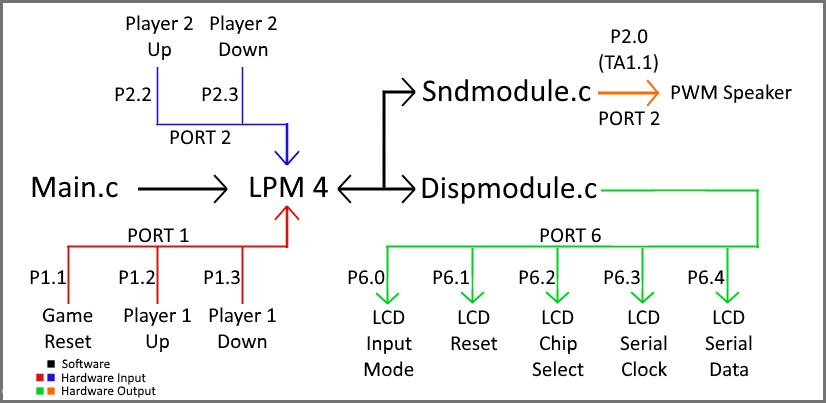


Figure 2: An overview of the program’s operation in similar form to a state machine.

Overall the game’s core functionality is centered around interrupts, timers, and synchronous frame rendering. After running some initialization code in the main function the programs default state becomes low-power mode four where all logic is driven by interrupts. Each interrupt vector is either that of a timer, for automated changes in game state, or a port, for user input. A timer acts as the clock signal for general game pacing and responsible for all game logic that is independent of user input, as well as speeding up the game over time. This timer is the primary interrupt trigger as it occurs every time the ball needs to move, which is the main variation in game state. Additionally, a separate interrupt triggers that moves a paddle each time either player presses a paddle control button. Finally, the game draws the position of both paddles and the ball at a synchronized rate that is separate from the game's pacing using another timer and interrupt vector. Due to limitations of the LCD, 30FPS was chosen as anything lower caused too the ball to move too many pixels per frame, and anything higher made the ball very hard to see because it would move before the LCD could finish drawing it entirely, creating a “ghosting” effect. Each of these interrupts require some form of communication with connected hardware. In the case of paddle control the input is simply received inside the interrupt vector for port 1 or 2, while for drawing the LCD or playing sound the program calls either of the two helper functions, sndmodule.c or dispmodule.c, which take care of sending the correct outputs to the correct hardware. The dispmodule.c code also contains functionality for pausing the game when a score occurs, displaying the score for a set duration, and then resuming the game.

In the end, this allows for game to function in a similar fashion to state machine where LPM 4 is the default state. As the users press their paddle buttons the program changes the appropriate variables, in addition to calculating and updating the ball’s position, and then draws the current position of all elements every frame and plays sounds when required. This is how the program is able to emulate pong and handle many changes in game state in quick succession.

The following is a brief specification summary, and functionality explanation of all the components that make up the game:

**Software Specifications:**

* Main.c - Initializes all registers related to ports, timers, interrupts, I/O, variable storage, etc., initializes the LCD, changes the clock speed of the MSP430 to about 2MHz, and sets its to low-power mode
* LPM 4 - Several interrupts vectors are pending for trigger while in low-power mode that execute upon player input for paddle location or game reset, when the ball changes position internally, when a sound needs to be played, or when a new frame needs to be drawn
* Sndmodule.c - Manages all functionality for playing and stopping sounds, including frequency and duration, for multiple scenarios
* Dispmodule.c - Takes care of all LCD related functions, including drawing the ball and paddles, clearing the screen, and displaying the score
* MSP430 built in modules used:
  + Timer A0, Timer A1, Timer A2
  + Timer B
  + Digitally Controlled Oscillator

**Hardware Specifications (I/O):**

* LCD - 128x64 dot matrix display, monochrome, SPI, 400ns minimum serial clock period, dedicated RAM and controller, backlight
* Paddle Buttons - Momentary, off-mom switches, 4-pins for control of up to two signals
* Reset Button - Momentary, off-mom switch built onto the launchpad
* Speaker - PWM based piezo buzzer, 75dB @ 3V
* Inputs:
  + PORT 1.1 - Game reset button: Restarts game at zero points for each player
  + PORT 1.2 - Player 1 Up: Moves player 1’s paddle up
  + PORT 1.3 - Player 1 Down: Moves player 1’s paddle down
  + PORT 2.2 - Player 2 Up: Moves player 2’s paddle up
  + PORT 2.3 - Player 2 Down: Moves player 2’s paddle down
* Outputs:
  + PORT 2.0 (TA1.1) - Piezo Input: Generates the PWM signal for the buzzer when sounds are played
  + PORT 6.0 - LCD A0: Dictates whether or not the next byte sent to the LCD is RAM data or a command
  + PORT 6.1 - LCD /RESET: Resets the LCD if set high
  + PORT 6.2 - LCD /CS: Set low while serial data is being sent to LCD, high otherwise
  + PORT 6.3 - LCD SCL: Serial clock input for the LCD
  + PORT 6.4 - LCD SI: Serial data input or the LCD

**II. Design Approach**

The MSP430 contains multiple on inputs and outputs that were used to communicate with the required hardware. Five buttons were used overall. Four were used for the up and down of each of the two players, as this is what the original game used, and one was for resetting the game. It was decided to use external buttons for everything but the reset button because they felt better to play on. It was also difficult to press the button 64 times before the ball would travel the length of the screen so a lower resolution grid was made for where the player can put their paddle. For this lab an external display was used that interfaced with the MSP430 via Serial Peripheral Interface (SPI). The serial communication was implemented via software using a technique called bit banging, where the program manually sets the serial clock low, changes the bit on the data signal, and then sets the serial clock high. This was prefered over using the MSP430’s hardware SPI module because it was much simpler to implement and the robust features of hardware SPI were not required for this project. It was decided to use a piezoelectric speaker for the game because they are optimal for producing sound with digital systems via PWM, and use very little power so it would not be limited by the maximum 2 milliamps that each pin of the MSP430 is able to output. Additionally, since piezoelectric buzzers are primarily used to generate pure tones it was perfect for emulating the sounds from the original game. The required pins from the MSP430 were broken out to a breadboard to interface with the screen and buttons. Every Timer A as well as Timer B was used for this project. It was decided to mostly use interrupts for the software component of this project because they were the learned method for the most efficient processor usage, as seen in previous labs. The only timer where the exact timer used actually mattered was with Timer A1, where Capture/Compare timer 1 was used in order to map the output directly to port 2.0. This prevented frequent interrupts from being needed

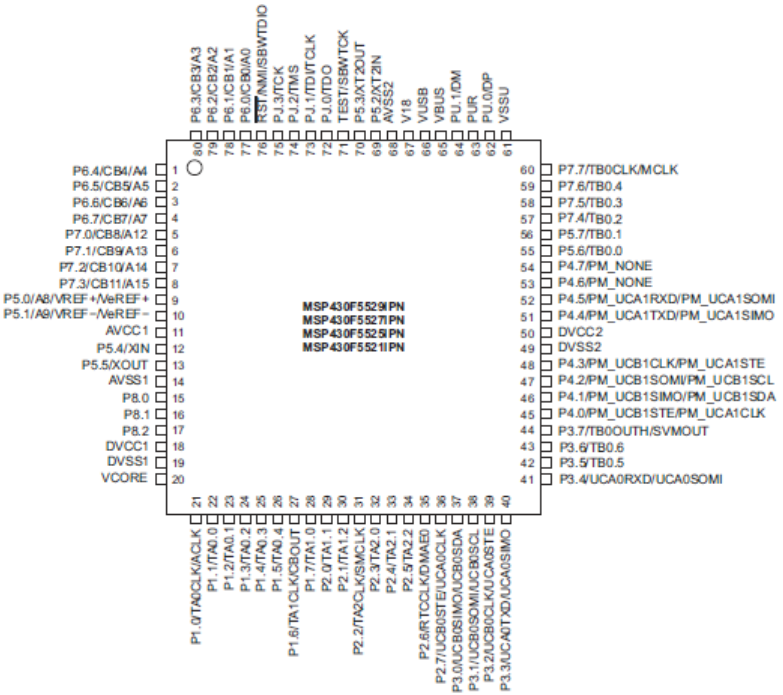


Figure 3: Diagram of the MSP430F552’s Port Map Controller which details the additional functionality of each port

The LCD screen purchased is 128 by 64 pixels. Because of the small size of each pixel it was decided to make the ball a 3x3 pixel object. The ball’s position within the game is determined by its center pixel in order to keep bounce angle calculations simple. The ball angles were one area where it was chosen to be improved on from the original pong. In the original version of pong the ball could only be bounced at 8 different angles. In this project it was decided to allow for the angle to be generated using the angle of incident with the paddle as well as a modifier that is calculated based off of point on the paddle where the ball impacts the paddle. Due to the subpixel location of the paddles’ grid, the ball is able to bounce at an effectively infinite number of angles.

The following are a collection of all pertinent figures and tables obtained from documentation belonging to the MSP430 or other used hardware.

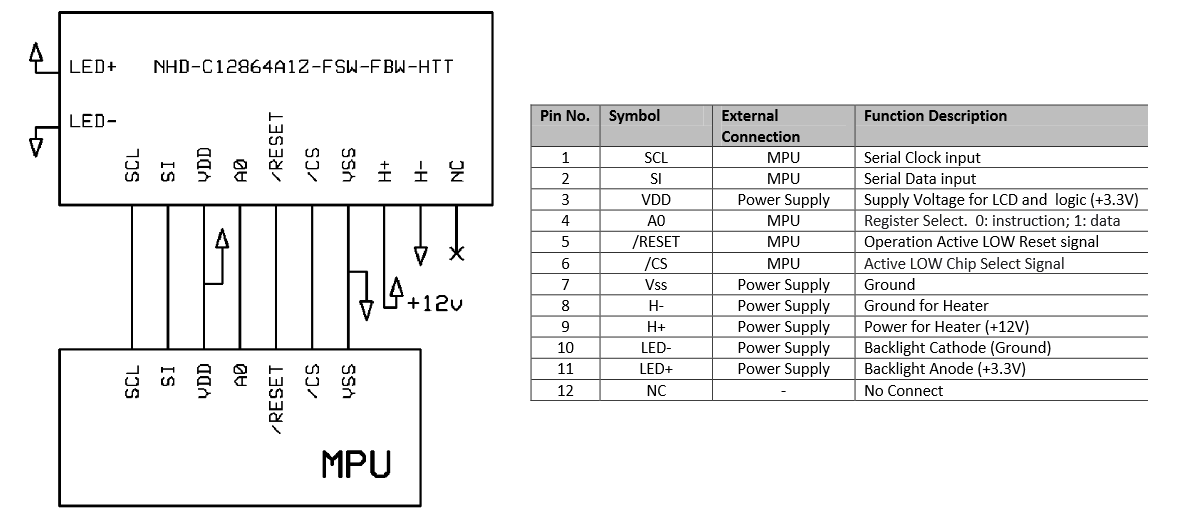


Figure 3. Diagram of the pinout of the LCD and the table that shows the function of each pin

Figure 3 is an overview of the connections made to the LCD that were discussed in the design specification. LED +/- were simply the Vcc and ground connections required for the LCD’s backlight, and H+/- were the Vcc and ground connections for an optional heating element that was not used for this project.

Learning how to control the LCD was accomplished by studying the datasheet for the ST7565 which was the LCD’s built in controller. Most of the information was not necessary for the scope of this project and primarily only the information on how to write to the display RAM was important. As mentioned earlier, the RAM uses a page/column based system, which is shown below in Figure 4.

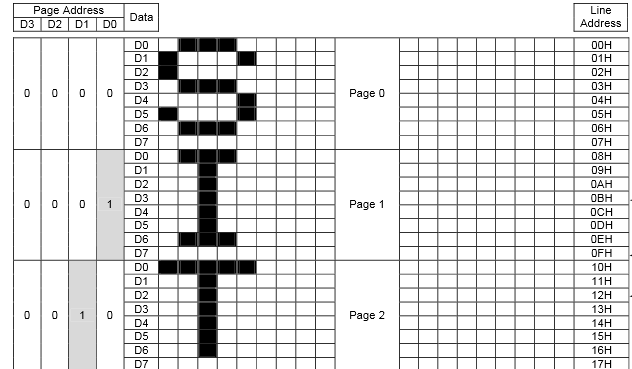


Figure 4: A portion of the diagram that illustrates how the controller's RAM functions

# **III. (PART OF DESIGN APPROACH)**

The code for this lab was created by following the design approach and specifically

setting port values to those used on MSP430 (i.e. the ports for the button, the LCD I/O pins, and the buzzer). Overall, the code can be broken down into several components that each perform the necessary tasks for accomplishing the objectives outlined for this lab. To keep the code organized an easy to follow, all code related to driving the buzzer and LCD were separated into their own C files with headers. The purpose of these segments will be discussed in this section.

The main function of this program, shown in Appenix A, starting on line 23, is responsible for setting up all the code for the rest of the game. First, it sets the used ports to the right functionality, and direction if they are being used as I/O. Second, it initializes the speaker driver and initializes the LCD display. The clock speed for the MSP430 is then increased to 2 MHz All of the variables are then set to their starting position. This was done instead of a planned starting splash screen due to time constraints. The timers needed for ball and paddle movement are then initialized.The msp430 is then put into low power mode waiting for the reset button to be pressed.

Appenix A, starting on line 96 and 111, shows the reset and score functions respectively. The reset function sets all global variables back to the default values and re-initializes the display. The display is reinitialized for two reasons; first to provide feedback to the user that the button had been pressed, and secondly there’s an occasional bug wherein the lcd display would randomly revert to an uninitialized state so adding initialization to the reset button was a quick solution. The reset button also enables the TA0 Capture/Compare interrupt, which is disabled on boot and game end. This interrupt enables ball movement. The score function is similar in how it resets all values (excluding the score) to the default values. It then displays the score screen and emits a noise. If one of the players has reached 5 points the game will terminate by disabling TA0 Capture/Compare interrupt, preventing the ball from moving.

The Port interrupts shown in Appenix A, starting on lines 134 and 168 for players 1 and 2 respectively, relate to the player button inputs. These interrupts are the entirety of the user’s ability to interact with the game. On Port 1 there are the Player 1 paddle movement controls as well as the reset button. The reset button is prioritized in the case statement, thus making it the highest priority function in the program. The Port 2 interrupt only controls the paddle movement for Player 2.

The Timer A0 interrupt begins on line 195 of the main.c file shown in Appendix A. This interrupt starts with a switch governing the ball’s X direction movement. For the vast majority of cases the switch will either move the paddle one pixel to the left or one pixel to the right. This is because the only player interaction with the game system occurs in the edges of the screen. In fact, a sufficiently good player would be able to play the game with only being able to see the ball while it is at these edge cases. First it is determined if the ball hits the paddle. This occurs when the ball is within the paddle length of the center of the paddle. Because the ball is 3 pixels wide and the hit detection is done from the center pixel of the ball and additional pixel had to be added to the paddle length for this calculation to allow for the paddle to hit the corner of the ball. At this the ball’s x direction is flipped. The reflected ball’s then given a new Y velocity. This velocity is calculated by multiplying a curve coefficient multiplied by the distance between the center of the paddle and the center of the ball. The paddle is a flat surface, however curve coefficient is a number that determines how much of a curved surface the paddle should act like during impact. This number is set to .15. This means that if the ball is incoming to the paddle hits at the far extreme top of the paddle, the ball will bounce at angle where it will move .9 pixels vertically for every horizontal pixel of movement. The ball is then pushed 1 pixel back into the normal play zone so that it will correctly move away from the paddle next frame, and a sound is played.

The Timer A0 interrupt slightly further on, on line 229 of main.c, handles the balls y movement. The y axis movement is the only point in the code where code could theoretically hang up if the ball’s y velocity gets too high. This occurred during development due to a casting error. If the ball is not impacting a wall then the current y velocity is added to the subpixel location of the ball. If the subpixel location is larger than a pixel then the ball will be shifted a pixel in the direction required and the subpixel value will be decremented. In order to be compatible with values over 1 y axis pixel per x axis pixel, or 45 degrees, without putting the ball outside of the play area the code will loop, evaluating 1 y pixel of movement per loop until the ball either no longer needs to move or impacts a wall. If the ball is on the same pixel as a wall and is moving toward that wall it will invert its y velocity, make a sound, and then perform the same operations that it would’ve completed were it not impacting said wall. This concludes all of the logic required for the ball to move.

All functionality related to sound was driven by ‘sndmodule.c’, which can be found in Appendix B. First, the ‘fnInitializeSound()’ function, which can be seen in Figure 5, needs to be called in the main function in order to initialize all related timer registers. Two timer modules were used for sound operations in the game because they could easily be used to generate and regulate PWM signals. this allows for the calling of ‘fnPlaySound’ which can be seen in Figure 6.

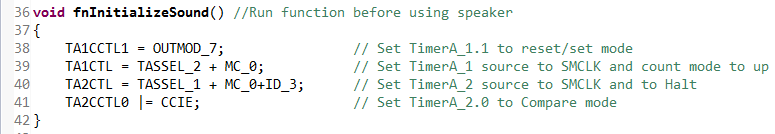


Figure 5: The function for initializing all timers used for sound

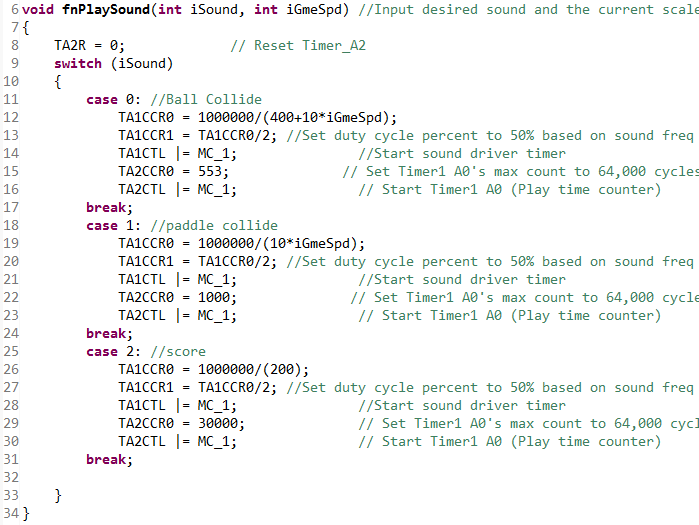


Figure 6 : The function that handles generating PWM signals to play all the games sounds

The function sets the appropriate timer registers so that a PWM signal with a 50% duty cycle, which allows for max volume, at a specified frequency is sent to the piezo buzzer. Because the sounds of the ball bouncing get higher in pitch as the game speed increases the current speed is passed to the function. Case statements were used to allow easy selection of which sound to play. A helper interrupt service routine, shown in Figure 7, is then used to stop the sound after it has played for the selected duration. Stopping the sound using an interrupt is beneficial because it allows the game to keep running during the sound.

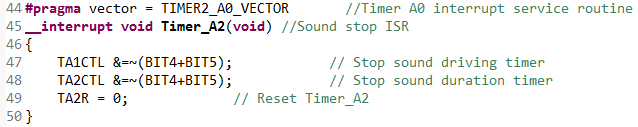


Figure 7: The helper ISR that stops the current sound that is playing.

On the other hand, all screen output is handled by ‘dispmodule.c’, which is found in Appendix C. Similarly to the sound handler, first, ‘fnInitializeDisplay()’ must be run in the ‘main()’ function to initialize the LCD RAM. This is shown in Figure 8.

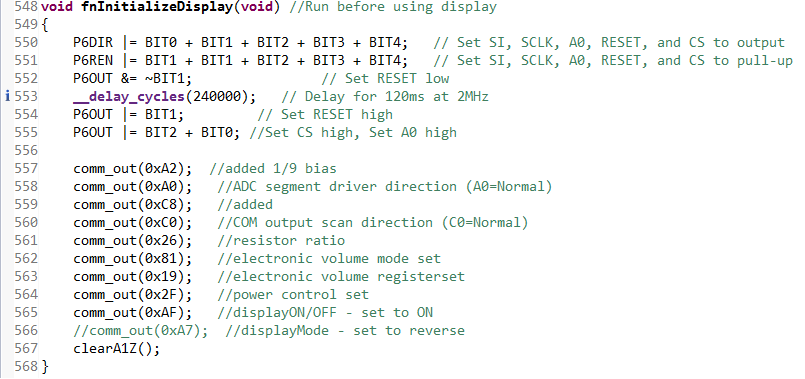


Figure 8: The function that initializes the LCD for writing

After the LCD is initialized the ‘fnRedrawScreen’ function is called everytime a frame is to be drawn at 30 FPS. The function is handed the current position of the paddles and ball and renders them on screen after erasing where they were in the previous screen using the ‘fnClearElements’. This clears the screen by writing zeros to each pixel where the ball and paddles used to be. The LCD is segmented into pages and columns. There are 128 columns and 8 pages with 8 rows each (64 pixel height). When writing to the screen RAM a page and column must be specified so the program calculates what pages and columns currently need pixels that need to be set high each frame and then write to them. There is additional logic in place to prevent writing to an address that would be off screen.

Additionally, the file contains the function ‘fnScoreDisplay()’ which is called whenever a player scores. The paddles and ball are cleared off the screen, a fixed divider and player one and and player two labels are drawn, and then both the players scores, which the function is passed, are written under their respective labels. Due to the size of these functions they are not shown here.

In terms of actually sending data to the LCD the functions ‘comm\_out()’ and ‘data\_out()’ were use. ‘comm\_out()’ was called when sending a command to the LCD such as setting a page or column start address in the display ram, and ‘data\_out()’ is used to send each byte that determines which pixels of each column are on or off. Each function is nearly identical except that ‘A0’ is set low while using ‘comm\_out()’ to signify that a command was being sent. Neither of these functions were called directly, but instead called within the score display and screen redraw functions. ‘comm\_out()’ is shown below in Figure 9.

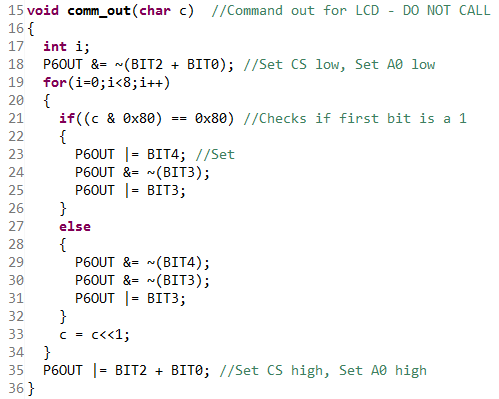


Figure 9: The command for sending commands to the LCD

The function sets the ‘A0’ and chip select lines to the appropriate values and then bit bangs the 8-bit command to the LCD by setting the serial clock line high, sending the most significant bit of the command, setting the serial clock signal low, left shifting the command, and then repeating the process until all bits are sent. The module also contains a ‘clear\_A1Z()’ function that clears the entire display by simply writing a zero to every address in the display RAM. This is used during initialization.

# **IV. Results and Conclusion**

The game was a playable recreation of pong as can be seen in Figure 10. Although not perfectly, all aspects of the original game were successfully emulated and new features were added. There were however a few issues that could be improved with future development. First there is the issue of the LCD display. The LCD wound up having a multitude of unexpected issues, including randomly re-entering an uninitialized state and ghosting. Another issue with the LCD was that the microcontroller could not change the screen fast enough for 60 frames per second. Another physical issue was that the buttons were not very stable in the breadboard. If a more permanent installation were required then this issue would be fixed with glue. With regards to the buttons there is also currently a bouncing issue that was resolved within the software using a delay. This increased computation time and would be better handled as a hardware fix by adding a small capacitor A bug occurred when increasing the clock speed that resulted in the code only working when hooked up to a laptop in the debugging mode of Code Composer Studio. No solution to that bug has been found. outside of these few points the code works as intended.

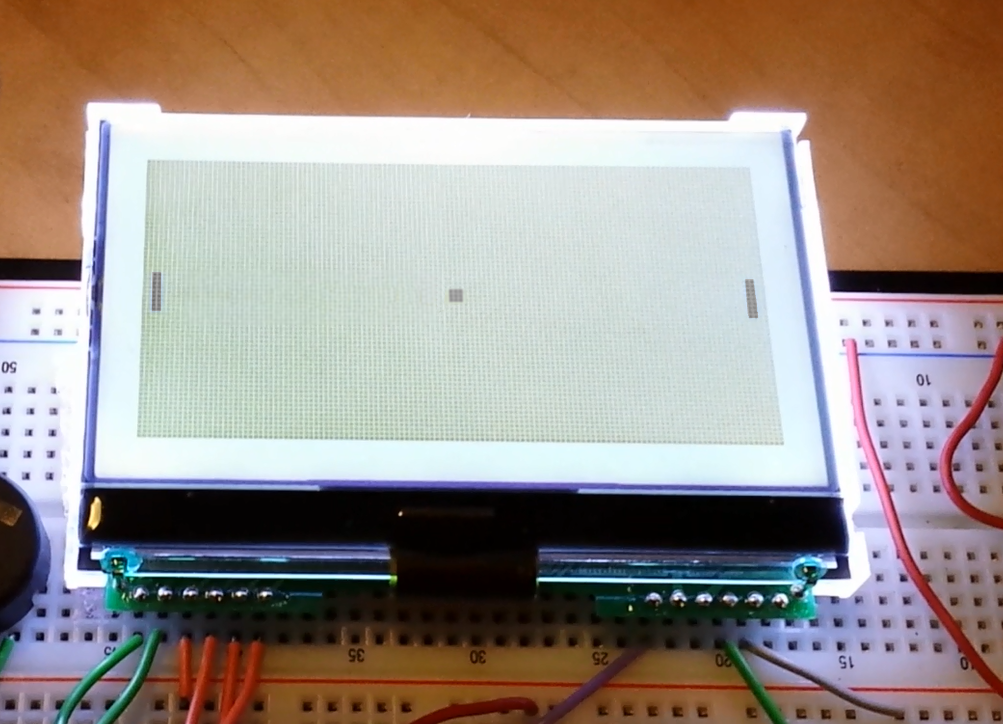


Figure 10: An example of the pong game

Overall the project was a major success. The remaining issues were minimal and most could easily be resolved if more time was allotted. Considering the success of the lab’s outcome and that its procedure covered all key points of recreating pong as laid out in the project proposal, the objective was most definitely completed. The MSP430 continues to be a versatile and powerful tool for learning the C language and the content of this course.

# 

**VI. Appendix**

1. main.c
2. #include <msp430.h>
3. #include <stdio.h>
4. #include <math.h>
5. #include "sndmodule.h"
6. #include "dispmodule.h"
8. #define padlocres 16 //paddle location resolution
9. #define padwidth 4
10. unsigned **int** p1pad,p2pad,iBallX,iBallY,balld,iP1Score,iP2Score,fc,spd;
11. unsigned **int** iPad1Y = 0 ;
12. unsigned **int** iPad2Y = 0 ;
14. unsigned **int** iPad1YL = 0;
15. unsigned **int** iPad2YL = 0;
16. unsigned **int** iBallXL = 0;
17. unsigned **int** iBallYL = 0;
18. **volatile** **float** ballsub,ballud;
19. **const** **float** abspad = 64/(**float**)padlocres;
20. **void** score(**int** player);
21. **void** reset(**void**);

24. **void** main(**void**) {
26. WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
28. //CHANGE CLOCK FREQUENCY TO 2MHz, and restore SMCLK to 1MHz
29. UCSCTL3 |= SELREF\_2; // Set DCO FLL reference = REFO
30. UCSCTL4 |= SELA\_2; // Set ACLK = REFO
31. \_\_bis\_SR\_register(SCG0); // Disable the FLL control loop
32. UCSCTL0 = 0x0000; // Set lowest possible DCOx, MODx
33. UCSCTL1 = DCORSEL\_2; // Select DCO range 24MHz operation
34. UCSCTL2 = FLLD\_1 + 60; // Set DCO Multiplier for 2MHz
35. // (N + 1) \* FLLRef = Fdco
36. // (60 + 1) \* 32768 = 2MHz
37. // Set FLL Div = fDCOCLK/2
38. UCSCTL5 = DIVS\_1;
39. \_\_bic\_SR\_register(SCG0); // Enable the FLL control loop
40. // Worst-case settling time for the DCO when the DCO range bits have been
41. // changed is n x 32 x 32 x f\_MCLK / f\_FLL\_reference. See UCS chapter in 5xx
42. // UG for optimization.
43. // 32 x 32 x 2 MHz / 32,768 Hz = 62500 = MCLK cycles for DCO to settle
44. \_\_delay\_cycles(62500);//
45. // Loop until XT1,XT2 & DCO fault flag is cleared
46. **do**
47. {
48. UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + DCOFFG);
49. // Clear XT2,XT1,DCO fault flags
50. SFRIFG1 &= ~OFIFG; // Clear fault flags
51. }**while** (SFRIFG1&OFIFG); // Test oscillator fault flag
52. // END FREQUENCY CHANGE
54. P1DIR = 0x00; // Set to input direction
55. P2DIR = 0x01;
56. P3DIR = 0xFF; //set output dir

59. P2IE =BIT3+BIT2;// start inturrpt management
60. P1IE = BIT1+BIT2+BIT3;
61. P2IES = BIT3+BIT2;
62. P1IES = BIT3+BIT2;
63. P1REN = BIT1+BIT2+BIT3;
64. P2REN = BIT3+BIT2;
65. P1OUT= BIT1+BIT2+BIT3;
66. P2OUT = BIT3+BIT2;
68. P2IFG =0;
69. P1IFG =0;
70. fnInitializeSound(); //initialize sounds
71. fnInitializeDisplay();


75. p1pad=padlocres >> 1;
76. p2pad=padlocres >> 1;
77. iBallX=64;
78. iBallY=32;
79. balld=0;
80. ballud=0;
81. iP1Score=0;
82. iP2Score=0;
83. P1IV=0;
84. P2IV=0;

87. TA0CCR0 =10000; //start clock management
88. TA0CTL=TASSEL\_2+MC\_1;
89. TA0CCTL0 |= CCIE;//add ccie if start on boot
90. P2SEL |= 0x01; //workaround
91. TB0CTL=TBSSEL\_1+MC\_1;
92. TB0CCR0=1092; // 30fps 32768/fps
93. TB0CCTL0 |= CCIE;
94. \_BIS\_SR(LPM4\_bits + GIE); //Debug LPM0
95. }
97. **void** reset(){
98. p1pad=padlocres >> 1;
99. p2pad=padlocres >> 1;
100. iBallX=64;
101. iBallY=32;
102. balld=0;
103. ballud=0;
104. iP1Score=0;
105. iP2Score=0;
106. P1IV=0;
107. P2IV=0;
108. TA0CCR0 =10000;
109. TA0CTL|=CCIE;
110. fnInitializeDisplay();
111. }
112. **void** score(player){
113. **if** (player)
114. iP2Score++;
115. **else**
116. iP1Score++;
117. fnPlaySound(2,spd);
119. p1pad=padlocres>>1;
120. p2pad=padlocres>>1;
121. iBallX=64;
122. iBallY=32;
123. balld=0;
124. ballud=0;
125. P1IV=0;
126. P2IV=0;
127. TA0CCR0 =10000;
128. spd=0;
129. **if**(iP2Score==5||iP1Score==5)
130. TA0CTL &= ~(CCIE);
131. fnScoreDisplay();
133. }
135. #pragma vector=PORT1\_VECTOR;
136. \_\_interrupt **void** PORT\_1(**void**)
137. {
138. **switch**(P1IFG) //cascading button press for player 1 and reset
139. {
140. **case** 0b00001100 :
141. **case** 0b00000100 : //Player 1 Up
142. **if**(p1pad<padlocres){
143. \_\_delay\_cycles(2000); //1ms delay at 2MHz
144. **if**(!(P1IN & BIT2))
145. p1pad++; //up
146. }
147. P1IFG &=~(BIT2);
148. **break**;
149. **case** 0b00001000 :
150. **if**(p1pad>0){
151. \_\_delay\_cycles(2000); //1ms delay at 2MHz
152. **if**(!(P1IN & BIT3))
153. p1pad--; //down
155. }
156. P1IFG &=~(BIT3);
157. **break**;
158. **case** 0b00000010 :
159. **case** 0b00000110 :
160. **case** 0b00001010 :
161. **case** 0b00001110 : //reset game
162. reset();
163. P1IFG &=~(BIT1);
164. **break**;
165. **default**:
166. **break**;
167. }
168. }
169. #pragma vector=PORT2\_VECTOR;
170. \_\_interrupt **void** PORT\_2(**void**)
171. {
172. **switch**(P2IFG) //cascading button press for player 2
173. {
174. **case** 0b00001100 :
175. **case** 0b00000100 :
176. **if**(p2pad<padlocres){
177. \_\_delay\_cycles(2000); //1ms delay at 2MHz
178. **if**(!(P2IN & BIT2))
179. p2pad++; //up
181. }
182. P2IFG &=~(BIT2);
183. **break**;
184. **case** 0b00001000 :
185. **if**(p2pad>0){
186. \_\_delay\_cycles(2000); //1ms delay at 2MHz
187. **if**(!(P2IN & BIT3))
188. p2pad--; //down
190. }
191. P2IFG &=~(BIT3);
192. **break**;
193. **default**: **break**;
194. }
195. }
196. #pragma vector=TIMER0\_A0\_VECTOR;
197. \_\_interrupt **void** Timer\_A0(**void**) //ball movements
198. {
199. **switch**(iBallX) //start x movement interrupt
200. {
201. **case** 4:
202. **if**(abs(iBallY-(p1pad\*abspad))<=(padwidth+1))
203. { //collison detection player 1 side
204. balld=1;
205. ballud=.15\*(iBallY-(p1pad\*abspad))+ballud;
206. fnPlaySound(1,spd);
207. iBallX++;
208. }
209. **else**
210. score(1);
211. **break**;
212. **case** 124:
213. **if**(abs(iBallY-p2pad\*abspad)<=(padwidth+1))
214. { //collison detection player 1 side
215. balld=0;
216. ballud=.15\*(iBallY-(p2pad\*abspad));
217. fnPlaySound(1,spd);
218. iBallX--;
219. }
220. **else**
221. score(0);
222. **break**;
223. **default**: // not on screen edge
224. **if** (balld)
225. iBallX++;
226. **else**
227. iBallX--;
228. **break**;
229. }
230. **switch**(iBallY)
231. {
232. **case** 0:
233. **if** (ballud<0)
234. {
235. ballud= -ballud;
236. fnPlaySound(0,spd);
237. }
238. ballsub+=ballud;
239. **if** (ballsub>1||ballsub<-1){ //ball y axis subpixel presicion
240. **while**(ballsub > 1 && ~(iBallY==63)){
241. iBallY++;
242. ballsub = ballsub - 1;
243. }
244. **while**(ballsub < -1 && ~(iBallY==0)){
245. iBallY--;
246. ballsub = ballsub + 1;
247. }
249. }
250. **break**;
251. **case** 63:
252. **if** (ballud>0)
253. {
254. ballud= -ballud;
255. fnPlaySound(0,spd);
257. }
258. ballsub+=ballud;
259. **if** (ballsub>1||ballsub<-1){ //ball y axis subpixel presicion
260. **while**(ballsub > 1 && ~(iBallY==63)){
261. iBallY++;
262. ballsub = ballsub - 1;
263. }
264. **while**(ballsub < -1 && ~(iBallY==0)){
265. iBallY--;
266. ballsub = ballsub + 1;
267. }
269. }
270. **break**;
271. **default**:
272. ballsub+=ballud;
273. **if** (ballsub>1||ballsub<-1){ //ball y axis subpixel presicion
274. **while**(ballsub > 1 && ~(iBallY==63)){
275. iBallY++;
276. ballsub = ballsub - 1;
277. }
278. **while**(ballsub < -1 && ~(iBallY==0)){
279. iBallY--;
280. ballsub = ballsub + 1;
281. }
283. }
284. **break**;
285. }
287. }

290. #pragma vector=TIMER0\_B0\_VECTOR;
291. \_\_interrupt **void** Timer\_B0(**void**) //game acceleration
292. {
293. fc++;
294. **if**(p1pad==padlocres)
295. iPad1Y=63;
296. **else**
297. iPad1Y=p1pad\*abspad;
298. **if**(p2pad==padlocres)
299. iPad2Y=63;
300. **else**
301. iPad2Y=p2pad\*abspad;
302. fnRedrawScreen();
303. **if** (fc>200) //number of frames before acceleration
304. {
305. TA0CCR0=TA0CCR0\*.96;
306. fc=0;
307. spd++;
308. }
309. }

B. sndmodule.c

1. #include <msp430f5529.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include "sndmodule.h"
6. **void** fnPlaySound(**int** iSound, **int** iGmeSpd) //Input desired sound and the current scaled game speed to play
7. {
8. TA2R = 0; // Reset Timer\_A2
9. **switch** (iSound)
10. {
11. **case** 0: //Ball Collide
12. TA1CCR0 = 1000000/(400+10\*iGmeSpd);
13. TA1CCR1 = TA1CCR0/2; //Set duty cycle percent to 50% based on sound freq
14. TA1CTL |= MC\_1; //Start sound driver timer
15. TA2CCR0 = 553; // Set Timer1 A0's max count to 64,000 cycles (64 milisec sound length)
16. TA2CTL |= MC\_1; // Start Timer1 A0 (Play time counter)
17. **break**;
18. **case** 1: //paddle collide
19. TA1CCR0 = 1000000/(10\*iGmeSpd);
20. TA1CCR1 = TA1CCR0/2; //Set duty cycle percent to 50% based on sound freq
21. TA1CTL |= MC\_1; //Start sound driver timer
22. TA2CCR0 = 1000; // Set Timer1 A0's max count to 64,000 cycles (64 milisec sound length)
23. TA2CTL |= MC\_1; // Start Timer1 A0 (Play time counter)
24. **break**;
25. **case** 2: //score
26. TA1CCR0 = 1000000/(200);
27. TA1CCR1 = TA1CCR0/2; //Set duty cycle percent to 50% based on sound freq
28. TA1CTL |= MC\_1; //Start sound driver timer
29. TA2CCR0 = 30000; // Set Timer1 A0's max count to 64,000 cycles (64 milisec sound length)
30. TA2CTL |= MC\_1; // Start Timer1 A0 (Play time counter)
31. **break**;
33. }
34. }
36. **void** fnInitializeSound() //Run function before using speaker
37. {
38. TA1CCTL1 = OUTMOD\_7; // Set TimerA\_1.1 to reset/set mode
39. TA1CTL = TASSEL\_2 + MC\_0; // Set TimerA\_1 source to SMCLK and count mode to up
40. TA2CTL = TASSEL\_1 + MC\_0+ID\_3; // Set TimerA\_2 source to SMCLK and to Halt
41. TA2CCTL0 |= CCIE; // Set TimerA\_2.0 to Compare mode
42. }
44. #pragma vector = TIMER2\_A0\_VECTOR //Timer A0 interrupt service routine
45. \_\_interrupt **void** Timer\_A2(**void**) //Sound stop ISR
46. {
47. TA1CTL &=~(BIT4+BIT5); // Stop sound driving timer
48. TA2CTL &=~(BIT4+BIT5); // Stop sound duration timer
49. TA2R = 0; // Reset Timer\_A2
50. }

C. dispmodule.h

1. //DUE TO MSP430's METHOD OF PORT REFERRAL, A PORT CANNOT SIMPLY BE DEFINED.
2. //THERFORE KEEP IN MIND THE FOLLOWING WHEN DOING PORT I/O:
4. //SCLK - P6.3 which is the seriel CLK output UCA0CLK (P2.7 is actual SCLK)
5. //SI - P6.4 which is also UCA0TXBUF (Serial output buffer) (P3.3 is actual SI)
6. //A0 - P6.0
7. //RESET - P6.1
8. //CS - P6.2
10. #include <msp430f5529.h>
11. #include <stdio.h>
12. #include <stdlib.h>
13. #include "dispmodule.h"
15. **void** comm\_out(**char** c) //Command out for LCD - DO NOT CALL
16. {
17. **int** i;
18. P6OUT &= ~(BIT2 + BIT0); //Set CS low, Set A0 low
19. **for**(i=0;i<8;i++)
20. {
21. **if**((c & 0x80) == 0x80) //Checks if first bit is a 1
22. {
23. P6OUT |= BIT4; //Set
24. P6OUT &= ~(BIT3);
25. P6OUT |= BIT3;
26. }
27. **else**
28. {
29. P6OUT &= ~(BIT4);
30. P6OUT &= ~(BIT3);
31. P6OUT |= BIT3;
32. }
33. c = c<<1;
34. }
35. P6OUT |= BIT2 + BIT0; //Set CS high, Set A0 high
36. }
38. **void** data\_out(unsigned **char** d) //Data out for LCD - DO NOT CALL
39. {
40. **int** i;
41. P6OUT &= ~BIT2; //Set CS low
42. **for**(i=0;i<8;i++)
43. {
44. **if**((d & 0x80) == 0x80) // Checks to see if first bit is a 1
45. {
46. P6OUT |= BIT4;
47. P6OUT &= ~(BIT3);
48. P6OUT |= BIT3;
49. }
50. **else**
51. {
52. P6OUT &= ~(BIT4);
53. P6OUT &= ~(BIT3);
54. P6OUT |= BIT3;
55. }
56. d = d<<1;
57. }
58. P6OUT |= BIT2; //Set CS high
59. }
61. **void** clearA1Z(**void**) //Clear for LCD
62. {
63. unsigned **int** a, i;
64. unsigned **char** page = 0xB0;
65. unsigned **int** x = 0;
67. **for**(i=0;i<8;i++)
68. {
69. comm\_out(page);
70. comm\_out(0x10);
71. comm\_out(0x00);
72. **for** (a=0;a<128;a++)
73. {
74. data\_out(0x00);
75. }
76. x+=128;
77. page++;
78. }
79. }
81. **void** fnClearElements(**void**) //Clear elements for LCD - DO NOT CALL
82. {
83. unsigned **int** iPage;
84. **float** fTemp;
85. unsigned **int** iCompNum;
86. unsigned **int** iPix;
88. /////////////////////Clear Ball//////////////////////////////
89. fTemp = iBallYL/(**float**)8;
90. iPage = (**int**)fTemp;
91. iCompNum = (**int**)((fTemp-iPage)\*8);
93. **if**(iCompNum == 0)
94. {
95. comm\_out(iPage+0xB0);
96. comm\_out(0x10+((iBallXL-1) >> 4));
97. comm\_out(0x00+((iBallXL-1) & 0x0F));
98. **for**(iPix = 0; iPix < 3; iPix++)
99. {
100. data\_out(0x00); //2 lines of ball at top of page
101. }
102. **if**(iBallYL != 0)
103. {
104. comm\_out(iPage-1+0xB0);
105. comm\_out(0x10+((iBallXL-1) >> 4));
106. comm\_out(0x00+((iBallXL-1) & 0x0F));
107. **for**(iPix = 0; iPix < 3; iPix++)
108. {
109. data\_out(0x00); //1 lines of ball at bottom of page
110. }
111. }
112. }
113. **else** **if**(iCompNum == 7)
114. {
115. comm\_out(iPage+0xB0);
116. comm\_out(0x10+((iBallXL-1) >> 4));
117. comm\_out(0x00+((iBallXL-1) & 0x0F));
118. **for**(iPix = 0; iPix < 3; iPix++)
119. {
120. data\_out(0x00); //2 lines of ball at bottom of page
121. }
122. **if**(iBallYL != 63)
123. {
124. comm\_out(iPage+1+0xB0);
125. comm\_out(0x10+((iBallXL-1) >> 4));
126. comm\_out(0x00+((iBallXL-1) & 0x0F));
127. **for**(iPix = 0; iPix < 3; iPix++)
128. {
129. data\_out(0x00); //1 lines of ball at top
130. }
131. }
132. }
133. **else**
134. {
135. comm\_out(iPage+0xB0);
136. comm\_out(0x10+((iBallXL-1) >> 4));
137. comm\_out(0x00+((iBallXL-1) & 0x0F));
138. **for**(iPix = 0; iPix < 3; iPix++)
139. {
140. data\_out(0x00); //3 lines of ball in middle of page
141. }
142. }
144. /////////////////////Clear P1 Paddle//////////////////////////////
145. fTemp = iPad1YL/(**float**)8;
146. iPage = (**int**)fTemp;
147. iCompNum = (**int**)((fTemp-iPage)\*8);
149. **if**(iCompNum < 4)
150. {
151. comm\_out(iPage+0xB0);
152. comm\_out(0x10);
153. comm\_out(0x02);
154. **for**(iPix = 0; iPix < 2; iPix++)
155. {
156. data\_out(0x00);
157. }
158. **if**(iPad1YL > 3)
159. {
160. comm\_out(iPage-1+0xB0);
161. comm\_out(0x10);
162. comm\_out(0x02);
163. **for**(iPix = 0; iPix < 2; iPix++)
164. {
165. data\_out(0x00);
166. }
167. }
168. }
169. **else**
170. {
171. comm\_out(iPage+0xB0);
172. comm\_out(0x10);
173. comm\_out(0x02);
174. **for**(iPix = 0; iPix < 2; iPix++)
175. {
176. data\_out(0x00);
177. }
178. comm\_out(iPage+1+0xB0);
179. comm\_out(0x10);
180. comm\_out(0x02);
181. **for**(iPix = 0; iPix < 2; iPix++)
182. {
183. data\_out(0x00);
184. }
185. }
187. ///////////////////////Clear P2 Paddle//////////////////////////////
188. fTemp = iPad2YL/(**float**)8;
189. iPage = (**int**)fTemp;
190. iCompNum = (**int**)((fTemp-iPage)\*8);
192. **if**(iCompNum < 4)
193. {
194. comm\_out(iPage+0xB0);
195. comm\_out(0x17);
196. comm\_out(0x0C);
197. **for**(iPix = 0; iPix < 2; iPix++)
198. {
199. data\_out(0x00);
200. }
201. **if**(iPad1YL > 3)
202. {
203. comm\_out(iPage-1+0xB0);
204. comm\_out(0x17);
205. comm\_out(0x0C);
206. **for**(iPix = 0; iPix < 2; iPix++)
207. {
208. data\_out(0x00);
209. }
210. }
211. }
212. **else**
213. {
214. comm\_out(iPage+0xB0);
215. comm\_out(0x17);
216. comm\_out(0x0C);
217. **for**(iPix = 0; iPix < 2; iPix++)
218. {
219. data\_out(0x00);
220. }
221. comm\_out(iPage+1+0xB0);
222. comm\_out(0x17);
223. comm\_out(0x0C);
224. **for**(iPix = 0; iPix < 2; iPix++)
225. {
226. data\_out(0x00);
227. }
228. }
229. }
231. **void** fnRedrawScreen(**void**) //Redraws screen
232. {
233. unsigned **int** iPage;
234. **float** fTemp;
235. unsigned **int** iCompNum;
236. unsigned **int** iPix;
237. unsigned **int** iST;
238. unsigned **int** iPadOut;


242. /////////////////////Draw Ball//////////////////////////////
243. fTemp = iBallY/(**float**)8;
244. iPage = (**int**)fTemp;
245. iCompNum = (**int**)((fTemp-iPage)\*8);
247. **if**(iCompNum == 0)
248. {
249. comm\_out(iPage+0xB0);
250. comm\_out(0x10+((iBallX-1) >> 4));
251. comm\_out(0x00+((iBallX-1) & 0x0F));
252. **for**(iPix = 0; iPix < 3; iPix++)
253. {
254. data\_out(0x03); //2 lines of ball at top of page
255. }
256. **if**(iBallY != 0)
257. {
258. comm\_out(iPage-1+0xB0);
259. comm\_out(0x10+((iBallX-1) >> 4));
260. comm\_out(0x00+((iBallX-1) & 0x0F));
261. **for**(iPix = 0; iPix < 3; iPix++)
262. {
263. data\_out(0x80); //1 lines of ball at bottom of page
264. }
265. }
266. }
267. **else** **if**(iCompNum == 7)
268. {
269. comm\_out(iPage+0xB0);
270. comm\_out(0x10+((iBallX-1) >> 4));
271. comm\_out(0x00+((iBallX-1) & 0x0F));
272. **for**(iPix = 0; iPix < 3; iPix++)
273. {
274. data\_out(0xC0); //2 lines of ball at bottom of page
275. }
276. **if**(iBallY != 63)
277. {
278. comm\_out(iPage+1+0xB0);
279. comm\_out(0x10+((iBallX-1) >> 4));
280. comm\_out(0x00+((iBallX-1) & 0x0F));
281. **for**(iPix = 0; iPix < 3; iPix++)
282. {
283. data\_out(0x01); //1 lines of ball at top
284. }
285. }
286. }
287. **else**
288. {
289. comm\_out(iPage+0xB0);
290. comm\_out(0x10+((iBallX-1) >> 4));
291. comm\_out(0x00+((iBallX-1) & 0x0F));
292. **for**(iPix = 0; iPix < 3; iPix++)
293. {
294. data\_out(0x07 << (iCompNum -1)); //3 lines of ball in middle of page
295. }
296. }
297. fnClearElements();
298. /////////////////////Draw P1 Paddle//////////////////////////////
299. fTemp = iPad1Y/(**float**)8;
300. iPage = (**int**)fTemp;
301. iCompNum = (**int**)((fTemp-iPage)\*8);
303. **if**(iCompNum < 4)
304. {
305. comm\_out(iPage+0xB0);
306. comm\_out(0x10);
307. comm\_out(0x02);
308. **for**(iPix = 0; iPix < 2; iPix++)
309. {
310. data\_out(0xFF >> (3-iCompNum));
311. }
312. **if**(iPad1Y > 3)
313. {
314. comm\_out(iPage-1+0xB0);
315. comm\_out(0x10);
316. comm\_out(0x02);
317. iPadOut = 0x80;
318. **for**(iST = 0; iST < (3-iCompNum); iST++)
319. {
320. iPadOut>>= 1;
321. iPadOut+=0x80;
322. }
323. **for**(iPix = 0; iPix < 2; iPix++)
324. {
325. data\_out(iPadOut);
326. }
327. }
328. }
329. **else**
330. {
331. comm\_out(iPage+0xB0);
332. comm\_out(0x10);
333. comm\_out(0x02);
334. **for**(iPix = 0; iPix < 2; iPix++)
335. {
336. data\_out(0xFF << (iCompNum-4));
337. }
338. comm\_out(iPage+1+0xB0);
339. comm\_out(0x10);
340. comm\_out(0x02);
341. iPadOut = 0x01;
342. **for**(iST = 0; iST < (iCompNum-4); iST++)
343. {
344. iPadOut<<= 1;
345. iPadOut+=0x01;
346. }
347. **for**(iPix = 0; iPix < 2; iPix++)
348. {
349. data\_out(iPadOut);
350. }
351. }

354. ///////////////////////Draw P2 Paddle//////////////////////////////
355. fTemp = iPad2Y/(**float**)8;
356. iPage = (**int**)fTemp;
357. iCompNum = (**int**)((fTemp-iPage)\*8);
359. **if**(iCompNum < 4)
360. {
361. comm\_out(iPage+0xB0);
362. comm\_out(0x17);
363. comm\_out(0x0C);
364. **for**(iPix = 0; iPix < 2; iPix++)
365. {
366. data\_out(0xFF >> (3-iCompNum));
367. }
368. **if**(iPad1Y > 3)
369. {
370. comm\_out(iPage-1+0xB0);
371. comm\_out(0x17);
372. comm\_out(0x0C);
373. iPadOut = 0x80;
374. **for**(iST = 0; iST < (3-iCompNum); iST++)
375. {
376. iPadOut>>= 1;
377. iPadOut+=0x80;
378. }
379. **for**(iPix = 0; iPix < 2; iPix++)
380. {
381. data\_out(iPadOut);
382. }
383. }
384. }
385. **else**
386. {
387. comm\_out(iPage+0xB0);
388. comm\_out(0x17);
389. comm\_out(0x0C);
390. **for**(iPix = 0; iPix < 2; iPix++)
391. {
392. data\_out(0xFF << (iCompNum-4));
393. }
394. comm\_out(iPage+1+0xB0);
395. comm\_out(0x17);
396. comm\_out(0x0C);
397. iPadOut = 0x01;
398. **for**(iST = 0; iST < (iCompNum-4); iST++)
399. {
400. iPadOut<<= 1;
401. iPadOut+=0x01;
402. }
403. **for**(iPix = 0; iPix < 2; iPix++)
404. {
405. data\_out(iPadOut);
406. }
407. }
409. //////////////////Setup variables for next clear////////////////
410. iPad1YL = iPad1Y;
411. iPad2YL = iPad2Y;
412. iBallXL = iBallX;
413. iBallYL = iBallY;
414. }
416. **void** fnScoreDisplay(**void**) // Shows score for 4 seconds
417. {
418. unsigned **int** iPage;
419. unsigned **int** iPlayer;
420. unsigned **int** iClm;
421. unsigned **int** iCompare;
423. clearA1Z();
424. comm\_out(0xA6); //displayMod - set to normal
426. /////////////////////////Draw P1 Label/////////////////////////
427. comm\_out(0xB7); comm\_out(0x11); comm\_out(0x0A);
428. data\_out(0x3F); data\_out(0x3F);
429. data\_out(0x33); data\_out(0x33); data\_out(0x33);
430. data\_out(0x3F); data\_out(0x3F);
431. data\_out(0x00); data\_out(0x00); data\_out(0x00);
432. data\_out(0x3F); data\_out(0x3F);
433. comm\_out(0xB6); comm\_out(0x11); comm\_out(0x0A);
434. data\_out(0xFC); data\_out(0xFC);
435. data\_out(0x00); data\_out(0x00); data\_out(0x00); data\_out(0x00); data\_out(0x00); data\_out(0x00); data\_out(0x00); data\_out(0x00);
436. data\_out(0xFC); data\_out(0xFC);
438. /////////////////////////Draw Divider/////////////////////////
439. comm\_out(0xB7); comm\_out(0x13); comm\_out(0x0F);
440. data\_out(0x3F); data\_out(0x3F);
441. **for**(iPage = 0xB6; iPage > 0xB0; iPage--)
442. {
443. comm\_out(iPage); comm\_out(0x13); comm\_out(0x0F);
444. data\_out(0xFF); data\_out(0xFF);
445. }
446. comm\_out(0xB0); comm\_out(0x13); comm\_out(0x0F);
447. data\_out(0xFC); data\_out(0xFC);
449. /////////////////////////Draw P2 Label/////////////////////////
450. comm\_out(0xB7); comm\_out(0x15); comm\_out(0x0A);
451. data\_out(0x3F); data\_out(0x3F);
452. data\_out(0x33); data\_out(0x33); data\_out(0x33);
453. data\_out(0x3F); data\_out(0x3F);
454. data\_out(0x00); data\_out(0x00); data\_out(0x00);
455. data\_out(0x31); data\_out(0x31); data\_out(0x31); data\_out(0x31); data\_out(0x31);
456. data\_out(0x3F); data\_out(0x3F);
457. comm\_out(0xB6); comm\_out(0x15); comm\_out(0x0A);
458. data\_out(0xFC); data\_out(0xFC);
459. data\_out(0x00); data\_out(0x00); data\_out(0x00); data\_out(0x00); data\_out(0x00); data\_out(0x00); data\_out(0x00); data\_out(0x00);
460. data\_out(0xFC); data\_out(0xFC);
461. data\_out(0x8C); data\_out(0x8C); data\_out(0x8C); data\_out(0x8C); data\_out(0x8C);
463. /////////////////////////Draw Scores/////////////////////////
464. **for**(iPlayer = 1; iPlayer < 3; iPlayer ++)
465. {
466. iClm = 4\*iPlayer + 13;
467. **switch**(iPlayer)
468. {
469. **case** (1):
470. iCompare = iP1Score;
471. **break**;
473. **case** (2):
474. iCompare = iP2Score;
475. **break**;
476. }
477. comm\_out(0xB4); comm\_out(iClm); comm\_out(0x0A);
478. **switch**(iCompare)
479. {
480. **case** (0):
481. {
482. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
483. data\_out(0xC0); data\_out(0xC0); data\_out(0xC0); data\_out(0xC0); data\_out(0xC0);
484. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
485. comm\_out(0xB3); comm\_out(iClm); comm\_out(0x0A);
486. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
487. data\_out(0x03); data\_out(0x03); data\_out(0x03); data\_out(0x03); data\_out(0x03);
488. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
489. }
490. **break**;
491. **case** (1):
492. {
493. data\_out(0x30);
494. data\_out(0x70); data\_out(0x70);
495. data\_out(0xF0);
496. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
497. comm\_out(0xB3); comm\_out(iClm); comm\_out(0x0A);
498. data\_out(0x03); data\_out(0x03); data\_out(0x03); data\_out(0x03);
499. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
500. data\_out(0x03); data\_out(0x03); data\_out(0x03); data\_out(0x03);
501. }
502. **break**;
503. **case** (2):
504. {
505. data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3);
506. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
507. comm\_out(0xB3); comm\_out(iClm); comm\_out(0x0A);
508. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
509. data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F);
510. }
511. **break**;
512. **case** (3):
513. {
514. data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3);
515. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
516. comm\_out(0xB3); comm\_out(iClm); comm\_out(0x0A);
517. data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F);
518. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
519. }
520. **break**;
521. **case** (4):
522. {
523. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
524. data\_out(0x07); data\_out(0x07); data\_out(0x07); data\_out(0x07); data\_out(0x07);
525. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
526. comm\_out(0xB3); comm\_out(iClm); comm\_out(0x0A);
527. data\_out(0x80); data\_out(0x80); data\_out(0x80); data\_out(0x80); data\_out(0x80); data\_out(0x80); data\_out(0x80); data\_out(0x80); data\_out(0x80);
528. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
529. }
530. **break**;
531. **case** (5):
532. {
533. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
534. data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3); data\_out(0xE3);
535. comm\_out(0xB3); comm\_out(iClm); comm\_out(0x0A);
536. data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F); data\_out(0x8F);
537. data\_out(0xFF); data\_out(0xFF); data\_out(0xFF);
538. }
539. **break**;
540. }
541. }

544. /////////////////////////Book Keeping/////////////////////////
545. \_\_delay\_cycles(4000000); // Delay for 4 seconds at 1MHz
546. comm\_out(0xA7); //displayMode - set to reverse
547. clearA1Z();
548. }

551. **void** fnInitializeDisplay(**void**) //Run before using display
552. {
553. P6DIR |= BIT0 + BIT1 + BIT2 + BIT3 + BIT4; // Set SI, SCLK, A0, RESET, and CS to output
554. P6REN |= BIT1 + BIT1 + BIT2 + BIT3 + BIT4; // Set SI, SCLK, A0, RESET, and CS to pull-up
555. P6OUT &= ~BIT1; // Set RESET low
556. \_\_delay\_cycles(120000); // Delay for 120ms at 1MHz
557. P6OUT |= BIT1; // Set RESET high
558. P6OUT |= BIT2 + BIT0; //Set CS high, Set A0 high
560. comm\_out(0xA2); //added 1/9 bias
561. comm\_out(0xA0); //ADC segment driver direction (A0=Normal)
562. comm\_out(0xC8); //added
563. comm\_out(0xC0); //COM output scan direction (C0=Normal)
564. comm\_out(0x26); //resistor ratio
565. comm\_out(0x81); //electronic volume mode set
566. comm\_out(0x19); //electronic volume registerset
567. comm\_out(0x2F); //power control set
568. comm\_out(0xAF); //displayON/OFF - set to ON
569. comm\_out(0xA7); //displayMode - set to reverse
570. clearA1Z();
571. }