



UNIVERSITY OF LEEDS

**ELEC5620M
Mini Project**

DE1-SoC Pong

Alexander Bolton - 200938078

Sam Wilcock - 201285260

John Jakobsen -

May 2019

The University of Leeds
School of Electronic and Electrical Engineering

Contents

1	Introduction	1
2	Display and Graphics	2
2.1	VGA Driver	2
2.2	Display Driver	3
2.3	Sprites and Text	4
2.4	Game Engine Graphics	5
3	Controls and Menus	6
3.1	Inputs	6
3.2	Menus & Screen Navigation	8
4	Audio Output	10
5	Game Physics	11
5.1	Collisions	11
5.2	Code Architecture	12
6	Problems Encountered	14
6.1	Ball becoming stuck at border/ paddle	14
6.2	Ball moving through paddle at collision	14
6.3	Ball disappearing after paddle collision	14
7	Conclusion	15
8	Appendix	16
8.1	Appendix A - sevenSeg.c	16
8.2	Appendix B - sevenSeg.h	17
8.3	Appendix C - DE1SoC VGA.c	18
8.4	Appendix D - DE1SoC VGA.h	19
8.5	Appendix E - pongDisplay.c	20
8.6	Appendix F - pongDisplay.h	25
8.7	Appendix G - pongEngine.c	26
8.8	Appendix H - pongEngine.h	31
8.9	Appendix I - pongSprites.c	32
8.10	Appendix J - pongSprites.h	34
8.11	Appendix K - pongPhysics.c	35
8.12	Appendix L - pongPhysics.h	37
8.13	Appendix M - pongInputs.c	38
8.14	Appendix N - pongInputs.h	41
8.15	Appendix O - pongSounds.c	42
8.16	Appendix P - pongSounds.h	44
8.17	Appendix Q - pongScreens.c	45
8.18	Appendix R - pongScreens.h	51
8.19	Appendix S - main.c	52

1 Introduction

This report will discuss the group project for the Embedded Microprocessor System Design module. The groups members were Alexander Bolton, Sam Wilcock, and John Jakobsen. The projects aim was to create a game of Pong on the DE1-SoC's microprocessor unit (MPU) which utilised the LT24 LCD Screen, a VGA screen, PS2 keyboard controls, button controls, and have audio output.

This report will be broken down into sections with section 1 being the introduction. Section 2 will discuss the display and graphics side of the project including the VGA driver which controls the monitor, the display driver which controls both LCD and VGA screens with a frame buffer, sprites and text which will go into depth of how the sprites are created, finally game engine graphics which will go into how the game engine uses the sprites including destroying, creating, and moving the sprites. Section 3 will discuss controls and menus and how the screen navigation works. It will also discuss the interrupts and how the PS2 driver is implemented. Section 4 will discuss audio output and the implementation of a fast sine function which avoids float mathematics as well as timer ISR which control sound operation. Section 5 will discuss the physics engine and the implementation of collisions and the code architecture. Section 6 will be the conclusion which will summarise the report and discuss if we have met the aims of the project. It will discuss what could be improved upon and changed. All code will be placed in the end of the report in the appendices.

2 Display and Graphics

2.1 VGA Driver

This subsection discusses the VGA driver and how it was implemented in the project. The VGA video out supports 640x480 however in this project is set to the default value of 320x240 pixels. The image displays from the VGA controller which is addressed from a pixel buffer. Each pixel value is write addressable using equation 1. An example of the pixel at 0,1 is shown in equation 2. The default base address for the pixel buffer is 0xC8000000 as stated in the manual. [1]

$$VGA_{baseaddress} + (pixelX_{coordinates} \text{ } pixelY_{coordinates} \text{ } 0_2) \quad (1)$$

$$C8000000_{16} + (00000001 \text{ } 0000000000 \text{ } 0)_2 = C8000400_{16} \quad (2)$$

The pixels are layed out with the y coordinate starting from the top to bottom of the screen. The x coordinate is from right to left of the screen as shown in figure 1.

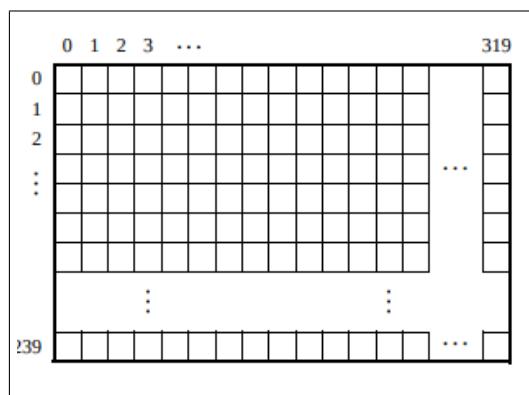


Figure 1: Pixel layout for pixel buffer of VGA controller [1]

Each pixel once addressed can be set to a value of colour with by setting bits for red, green, and blue. Each colour is allocated 5 bits which indicate the strength of colour for the pixel as shown in figure 2.

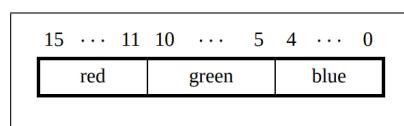


Figure 2: Pixel Colour Layout [1]

This code extracted from the project shows how a pixel is set in C.

```

1 void VGA_drawPixel(int x, int y, short colour){
2     // Call address of pixel
3     // Address base + [Pixel Y][Pixel X]0
4     volatile short *vga_addr=(volatile short*)(0xC8000000 + (y<<10) + (x<<1));
5     *vga_addr=colour; //Set pixel to colour
6 }
```

Figure 3: Code used to set pixel to a colour.

2.2 Display Driver

This subsection discusses the display driver which allows pixels to be set to a frame buffer and refreshed on command. The frame buffer is made up of 2 types of arrays which were a front frame buffer (what's currently on screen) and a rear frame buffer (what wants to be put onto the screen). The advantage of this is that it allows pixels to be checked and allows refresh on command.

Once the screen is desired to be refreshed the memory is compared between the front frame buffer and rear frame buffer to check if they are the same. If not the frame buffers are updated by checking each pixel in the frame buffers values. Any differences then update to display and to the front frame buffer. When first creating this frame buffer a problem with articulating occurred which is believed due to a memory overflow. The decision was made to split the frame buffer into 2 halves which reduced the memory used in each array which alleviated the issue.

Checking the pixels in a frame buffer is a very slow process so multiple frame buffers were created to experiment with. This experimentation involved splitting the screen into multiple sections and finding how many sections would be the fastest. This improved performance as multiple frame buffers were checked in memory and making the changes was done by comparing all pixels in a smaller area when there was a change in that area. In this experimentation frame buffers of 1 section to 8 sections were created. It was found that the 8 section frame buffer yielded the faster rendering of the frame buffer to the screens. All frame buffers were kept in the library which can be set. As it was still a slow process for the game a frame skip feature was also added to which skips a number of frames before refreshing. This speeds up the game dramatically. Also added to reduce errors and prevent system crashing was a 'Displays_setWindow' when set this prevents the driver from trying to address pixels outside of the window.

To compare frame buffers quickly the function 'memcmp' was used which set a variable in the function. If a change was found then the frame buffer would update.

```

1  for(x = 0; x < PixelWidth; x++){
2      short colour = frontFrameBuffer[x][y];
3      if(frontFrameBuffer[x][y] != rearFrameBuffer[x][y]){
4          short colour = frontFrameBuffer[x][y];
5          Displays_drawPixel(x,y,colour);
6          rearFrameBuffer[x][y] = colour;
7      }
8  }
9
10 }
11 //This function fills the displays a set colour.
12 void Displays_fillColour(short colour){
```

Figure 4: Code used to compare and update the buffers (Quad buffer)

A number of functions were created for use by the engine. These are functions such as 'Displays_init' which initialises the displays, 'Displays_mode' to set which buffer to use, 'Displays_setPixel' to set the pixel to a colour in rear buffer, 'Displays_Refresh' to refresh the buffer to hardware, 'Displays_ForceRefresh' to force a display refresh regardless of frame skip count, and finally 'Displays_getPixel' which returns the value of the colour of the pixel requested.

2.3 Sprites and Text

This subsection discusses how sprites are rendered and created in the library and how text is also created. There are 2 main sprites for this game of pong.

The first sprite is the ball. The ball must be initialised into an array before it can be rendered. To initialise the ball Pythagoras theorem was used (as previously completed in the graphics library of a previous assessment in this module). Equation 3 is Pythagoras theorem equation which is used. When the result is equal to or is less than radius squared then the value in the array is set to 1. The code iterates from the centre point for all values of x and y and sets an array. This was created as such to allow the ball to be dynamically set in size. Once rendered the ball can be rendered by providing coordinates of where to render (from centre of ball) and a colour. This will use these values and set the pixels to the frame buffer.

$$\text{Radius}^2 = x^2 + y^2 \quad (3)$$

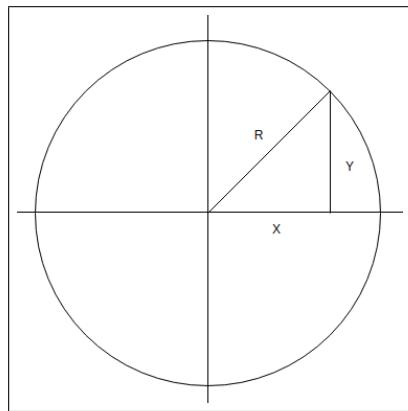


Figure 5: Pythagoras Theorem demonstrated inside circle

The paddle was set by iterating through a square of defined values and setting these pixels on by using a function. This was then called on every change of the paddle.

When generating and rendering text a library of hex values is used. The user inputs the character array of text they wish to use. These hex values are at 90 degree angles so the code first rotates the text 90 degrees to be upright for each letter in the char array into a letter array. A for loop is then used to set the pixels of the letter array to the frame buffer. If small then the is simply reads from the letter array and sets the appropriate pixel. If large 1 pixel in the letter array sets 4 pixels in the frame buffer. This allows large bold text.

2.4 Game Engine Graphics

This section explains how the game engine uses sprites to make a usable resource in the final game engine. In the game engine each sprite (1 ball, 2 paddles) are kept a track of with coordinates set as global values. Each sprite can be created and destroyed using commands such as 'pongEngine_paddleCreate(number of paddle)'.

Once the ball is created it is required to be moved. This is carried out using the function 'pongEngine_moveBall(int angle, int speed)'. To do this code was created to make a ball path. The ball path is saved as a set of instructions in an array.

Firstly to create the set of instructions the angle of the path must be found. This is carried out by finding the angle from the centre of the ball to the outside pixels of the screen. The coordinates are input into function 'pongEngine_calcAngle' from a for loop which outputs an angle between 2 pixel co-ordinates. Once the angle has been found these coordinates are input into the function 'pongEngine_genBallPathInst(ballX, ballY, angleX, angleY)' with ballX, ballY being the current coordinates of the ball and angleX, angleY being the outside coordinates which are at the required angle. This function generates a set of instructions to an array of instructions. The set of instructions is created using Bresenham's line theorem which creates a non-visible line which the ball can follow. If the angle is not changed then the instructions will not be regenerated. Everytime the ball moves it is first destroyed (turns all pixels black) and then redrawn in its new location.

```

1     error += dy;
2     x1 += sx;
3 }
4 //if error2 is smaller than delta x then add 1 to y
5 if (error2 <= dx) {
6     error += dx;
7     y1 += sy;
8 }
```

Figure 6: Code used to calculate angle between 2 co-ordinates

The paddles have 2 functions ('pongEngine_paddleSetYLocation(int player, int y)' and 'pongEngine_paddleSetXLocation(int player, int x)') which set the x and y value, destroys the selected paddle, redraws it, and finally stores the values into memory. These are used on initialisation, whilst in game the function 'pongEngine_paddleMove(int player, int direction, int speed)' is used. This function takes the values of player, direction, and speed. The speed set is how many pixels the paddle must be moved. Direction is either up or down.

To update the score three functions are used, one function is to add a point to a selected player which increases the value of score in a variable. Another function resets the scores to zero. Finally a refresh score function which takes the scores from the stored variables and renders the score on screen. The score values are split down into individual numbers and then sent to the seven segment displays. The numbers are also turned into a char array. Every time the score is refreshed a black rectangle is drawn over the text to make it blank and then it is re-rendered using 'pongSprites_WriteText'. An issue we encountered was random pixels appearing next to the score. The error was spotted by team member Sam Wilcock to be that a char array exit character ('\0') was missing.

3 Controls and Menus

3.1 Inputs

This subsection discusses the implementation of the PS/2 keyboard, pushbutton and slider switch inputs.

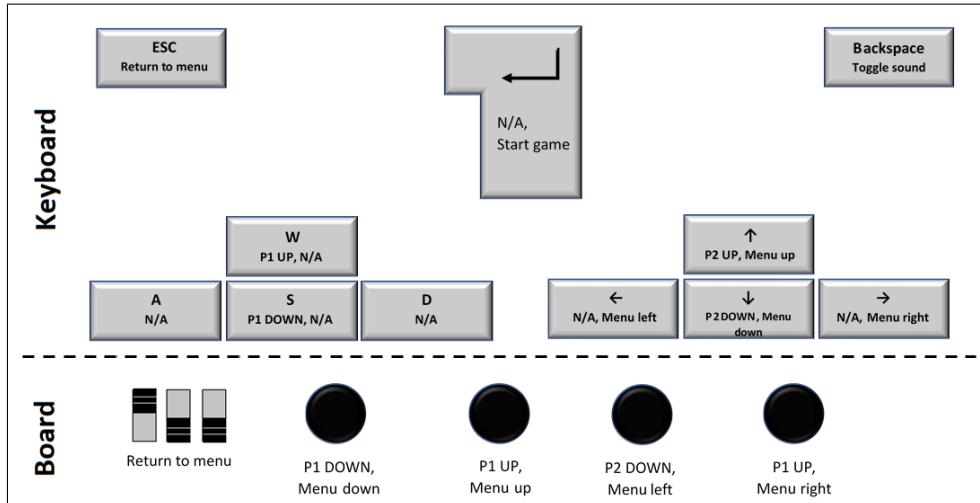


Figure 7: Input functionality

The slider buttons are arguably the simplest to implement. They are not attached to any of the built in IRQ exception handlers, and thus have to be dealt with by continuously checking their values. This is dealt with purely in the 'pongScreens.c' file as part of the gameplay screens' while loops; the pointer to the sliders' base address is defined initially, along with a buffer variable to store the previous value of the sliders as a new screen is formed, as such:

```
1 last_slider = *slider_ptr; // Ensure slider positions are saved
```

Figure 8: Save slider values

As the code for the gameplay screens are run, after the ball has been moved, the slider value is compared to the buffer, and if any of the switches have been moved, the gameplay mode variable is set to MENU (a useful dummy definition in the header file) and the game exits to the main() while loop:

```
1 if ((pongEngine_getBallLocation_y() <= pongEngine_getPaddleY(2) + 30 + 15) && (pongEngine_getBallLocation_y() >=
2 pongEngine_getPaddleY(2) - 30 - 15)) {
3     paddleBeep();
4     arr = pongPhysics_paddleCollision(vel, dir, 2);
```

Figure 9: Check for exiting to menu

The use of the keyboard and the pushbutton switches allowed the use of IRQs as they have interrupt exceptions attached to them, thus avoiding continuously checking inputs unless they are flagged as having changed. In order to enable this, a custom 'DDRRomIRQ.scat' scatter file was written (see **Appendix**). For the keyboard input, a keyboardISR handler

was written in line with the 'HPS_IRQ.c' format. On the keyboard interrupt flagging, the PS/2 port interrupt is first deactivated to prevent further interrupts; the RVALID flag is checked and the PS/2 data FIFO read [1] to accept the incoming characters. The input characters are compared against some scancode definitions in the 'pongInputs.h' header [2]. Make and break codes are discarded and the key input is passed to the 'void Input(unsigned int key, unsigned int speed)' function. If the key is preceded by a break code it is discarded to prevent bounce from single presses; the PS/2 FIFO is finally cleared by reading it until RAVAIL is decremented to 1, before the interrupt is reactivated. [1].

The pushbutton ISR works in a similar way, by first reading the value from its interrupt register and then writing back to it to clear the interrupt flag (preventing infinite interrupts). The value is then compared in a simple state machine to ascertain which button was pressed, before the key is converted to a corresponding keyboard scancode and passed to 'void Input()'.

Additional macros which were written to help the functioning of the inputs include 'enableInputs(int enable)' which switches the input interrupts on and off; 'setInputMode(unsigned int _mode)' which sets the *mode* variable, in order for the screen functions to switch and the Input function to deal with different screens; 'int getInputMode(void)', which returns the current value of *mode*; and, most importantly, the 'Input(unsigned int key, unsigned int speed)' function, which is a state machine dealing with the inputs. The speed input serves a dual purpose. First, it was noted that it was far easier to press the keyboard keys faster than the pushbutton keys, in part due to the short typematic delay [2], and so this allows the function to penalise the paddle speed accordingly. The other use is in detecting whether the input has come from the keyboard or pushbutton for dealing with menu motions. Fig 10 below demonstrates the macro's logic.

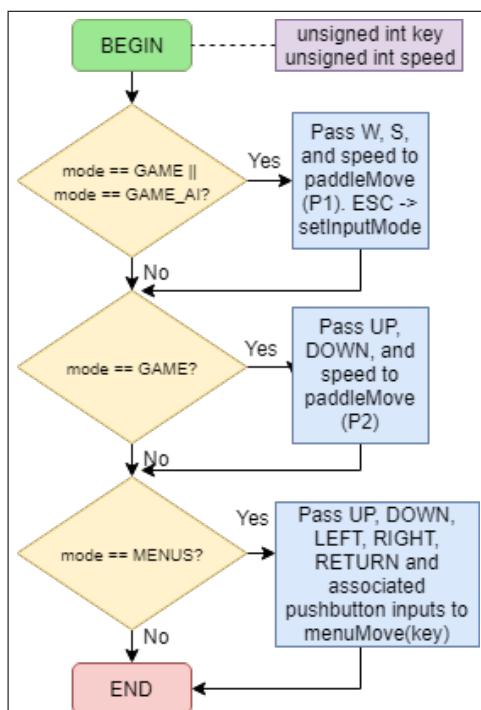


Figure 10: The Input() macro flow

3.2 Menus & Screen Navigation

In this subsection, the menu and screen driver is discussed, along with how they are interfaced with from the main program loop.

There are 3 main types of screen to be reached within the pong game: a start-up screen, loaded once the game is switched on; a menu system, which needed to contain important settings for the game and be reachable from the game; and a game-play screen, where the pong engine is utilised. The screen functions are called from the main() source. After the various libraries have been initialised, the start screen is called, and then the program enters an infinite loop which continuously checks the *mode* variable from pongInputs. Different functions are then called depending on the mode - either MENUS, GAME, or GAME_AI.

The start-up screen was simplest to implement, as it is non-interactive and runs on a timed basis; this meant that it could utilise purely blocking functions, including sound generation. At this point, the inputs are not enabled, so as not to produce unwanted effects for keypresses. A sound is played and a greeting displayed over a colour fill; this gave the added benefit at the time of debugging of being a useful tester for display and sound functions. This screen displays for a total of 4 seconds – 2 for the sounds to play in addition to an extra 2 second sleep.

```

1 void startScreen() {
2     // Clear screen and set input mode
3     Displays_fillColour(_RED); // Fill background
4     setInputMode(MENUS);    // Ensure input mode set to menus
5
6     // Write greeting and refresh
7     pongSprites_writeText(60, 120, 1, "Welcome to armPONG", 0xFFFF);
8     Displays_forceRefresh();
9     ResetWDT();
10
11    startSound();           // Play startup sound
12    usleep(1000000); ResetWDT(); // Sleep for 2s
13    usleep(1000000); ResetWDT();
14    last_slider = *slider_ptr; // Ensure slider positions are saved
15 }
```

Figure 11: Code used to run start-up screen.



Figure 12: Start-up screen

The menu system contained both blocking and non-blocking elements, including input and timer ISRs, sounds and animation, making it more complex. The menu is initialised simply, by using the 'pongSprites_writeText' function to produce a list of items, with the currently selected menu item (dictated by variable *menuSelector*) highlighted in blue, in addition to clearing the 7-segment displays. The setting values for the game mode, paddle size and volume are displayed next to the text, where the mode can be either 2-player or 1 player (AI) mode, and the paddle size and volume are an integer from 1 to 9. The 'pongSprites_renderBall' macro was also used to provide a pointer to the current menu item. While the menu *mode* is set to MENUS, the 'gameMenu(void)' function simply loops, waiting. The motion of the menu and its effects are passed on to the 'menuMove(unsigned int direction)' macro, which the input interrupts interact with through 'Input'; this function stores the previous values of *menuSelector* and each menu option, and if there is a change, it shifts the ball pointer and colour highlighting of the text. It does this simply by writing over the previous text and ball items in the default colours (background white for the ball, black for the text), and then writing over the newly selected locations.



Figure 13: Menus, showing choice of item, option, and the animation which runs on START

The first option, Mode, dictates the game mode, and works simply by changing the *gameMode* variable which is passed to 'setInputMode' on starting a game. Second is the paddle size, which changes the paddle size with 'pongSprites_changePaddleSize'. This function was built especially, and works on the formula $Length_y(px) = 60 + (size - 5) \times 10$ - hence the default option of 5 (out of 10) gives a paddle height of 60 pixels.

The volume setting was slightly more difficult, as it was decided that changing the volume should play a tone at the current amplitude defined by this setting. It was found that the blocking nature of sounds combined with the fact that 'menuMove' is called by interrupts caused the sounds to play forever; hence this option only changes the volume (using 'setVolume') and then the screen's main loop plays sounds if it sees a change in this setting.

In order to reset the score, a function was added to pongEngine, which simply sets the scores for each player to zero. Finally, the start button resets the score if *gameMode* has changed, and then sets the input mode, allowing the menu loop to end and return to *main()*. An animation runs on pressing START, which moves the ball across the start text, erasing it.

It was initially found that if inputs were pressed whilst the menu items were changing, a bug occurred where the display did not accurately depict the current menu items selected. This was remedied by disabling the inputs and re-enabling either side of 'menuMove', using 'enableInputs'.

4 Audio Output

This section explains the method used to implement sounds in the final design. In order to include sounds, the DE1-SoC WM8731 and I2C drivers were utilised in combination with an example code. [Reference T. Carpenter's work from lab 4??]. It was noted initially that the code was slow to perform due to the float arithmetic involved in calculating the sinusoid of an angle, which caused the FIFO buffer of the audio device to overflow. We realised therefore that there were 2 real ways of approaching the problem: either, to utilise the onboard FPU of the DE1-SoC, or to create a faster approximation of the $\sin\theta$ function.

The code below demonstrates the fast sin function that was devised. The values of $\sin\theta$ from 0° to 90° were first calculated in Matlab, multiplied by 10^5 and rounded to whole numbers. These results were then copied into a lookup table within the 'lookupSin(unsigned int degree)' macro. The storing of the numbers as fixed integers as opposed to floating point numbers served to save on memory usage.

```

1 static unsigned int lookup[91] = { 0, 1745, 3490, 5234, 6976, 8716, 10453, 12187, 13917,
2     15643, 17365, 19081, 20791, 22495, 24192, 25882, 27564,
3     ... etc. ...
4     if (degree <= 90){
5         return (float) lookup[degree]/100000;
6     }
7     else if ((degree > 90) && (degree <= 180)){
8         return (float) lookup[180-degree]/100000;
9     }
10    else if ((degree > 180) && (degree <= 270)){
11        return (float) -lookup[degree-180]/100000;
12    }
13    else if ((degree > 270) && (degree <= 360)){
14        return (float) -lookup[360-degree]/100000;
15    } else return 0;
16 }
```

Figure 14: Code used produce fast sin lookup results for whole degrees.

By utilising the characteristics of the $y = \sin\theta$ function over the period 0° to 180° , where $[y]$ is mirrored about $x = 90^\circ$, and the characteristic that the function is mirrored about $y = 0$ at $x = 180^\circ$, it was then possible to find an approximation of $\sin\theta$ for any whole positive integer θ up to 360° . Within the Sound macro, the float input to 'lookupSin', indicating the phase, was wrapped to a maximum of 360° and cast to an integer type. The maximum error in this lookup function at whole integers was approximately 4.9×10^{-6} .

The 'Sound(int _freq, float _duration)' macro first checks if sounds are flagged as on (via the *SoundOn* variable). If so, a timer ISR is created to run for the period length. The phase is then incremented by a set amount determined by the frequency argument. The sounds are pushed to the WM8731 FIFOs and played in a while loop which checks the *sound* variable. When the timer ISR triggers, *sound* is changed to 0 and the Sound function finishes. It was initially noted that during gameplay, key presses caused gaps in sounds, so the 'Sound' macro temporarily disables inputs for the duration it runs. Variable volume was implemented by adding a *VOLUME* variable, which can be changed globally using the 'setVolume(int _volume)' macro. The amplitude sent to the WM8731 is set as $2^{23+VOLUME}$, by bitshifting the value 2^{23} left *VOLUME* bits. In order to make sound generation more intuitive, an additional header file was created containing the frequencies of musical notes from A3 to G6, and then sound macros were created for the various components of the game, including paddle collisions, scoring, and the start sound.

5 Game Physics

This section of the report details the physics engine of the game which is used to regulate the movement of the ball and its interactions with the paddles and the arena borders.

5.1 Collisions

Collisions are very important in a game of Pong as they are the only means by which the player can interact with the ball. The code in this project has been designed so that the ball moves through the game arena with a speed and angle that can be changed by collisions with the border and the paddles. Each paddle is divided into sectors which each produce different outgoing speed and angle values. Sectors that produce a more pronounced outcome, and therefore one more likely to surprise the other player, are smaller and more difficult to hit. A skilful player should be able to anticipate the path of the ball and align their paddle accordingly to gain maximum advantage from the collision.

Figure 15 shows a schematic of the paddle with its collision sectors. Shown in yellow is sector 1, where the ball is perfectly reflected (i.e. angle of reflection = angle of incidence) and the outgoing speed is lowest. This sector is the largest as it gives the least amount of advantage to the player; the rebound angle is predictable and the outgoing speed is relatively easy to manage. Sector 2 is the second largest sector as it gives a 10 degree angle change and a medium outgoing speed. Sector 3 is the smallest and gives the largest change in angle with 20 degrees and the highest outgoing speed.

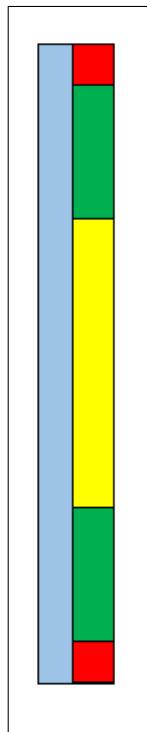


Figure 15: showing location and relative size of each sector on the paddle.

5.2 Code Architecture

The code for the physics of the game engine has its origin in the pongScreens.c file. This is where the game actually runs from and contains the while loop in which the ball is moved, and hence also where the collision checks occur. Six sets of collision checks are included within this while loop; two for the paddle collisions, two for each of the horizontal border collisions and two for the vertical sides of the arena. This latter set is not involved in the game physics as when a collision occurs here the game is reset and a point is added to the opposing player.

The code for the horizontal border collisions is quite simple. Within the while loop in pongScreens, the collision check for the horizontal sides of the arena is triggered when the ball enters a predefined space, defined as 20 pixels from the top of the screen or 10 pixels from the bottom. When the ball enters this area, a new angle is retrieved from the function pongPhysics_borderCollision with the new angle simply calculated as the negative of the incoming angle. For example, an incoming angle of 20 degrees will produce an outgoing angle of negative 20 degrees to the horizontal (left being 0) as shown in figure 16.

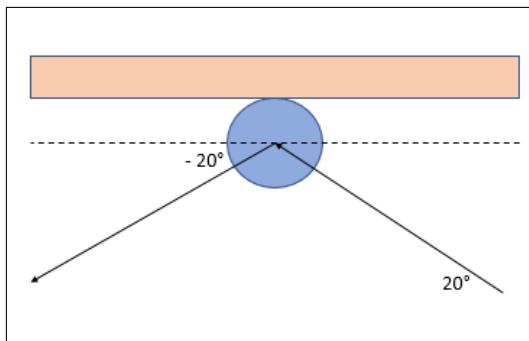
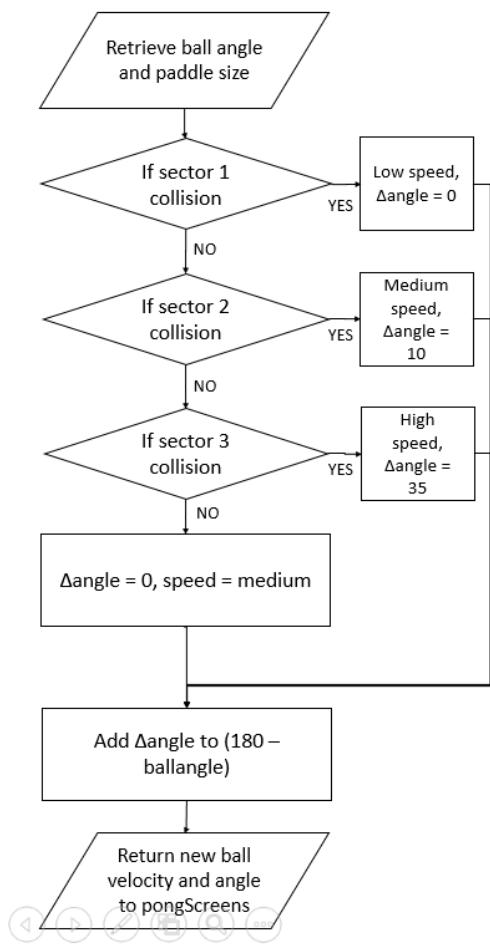


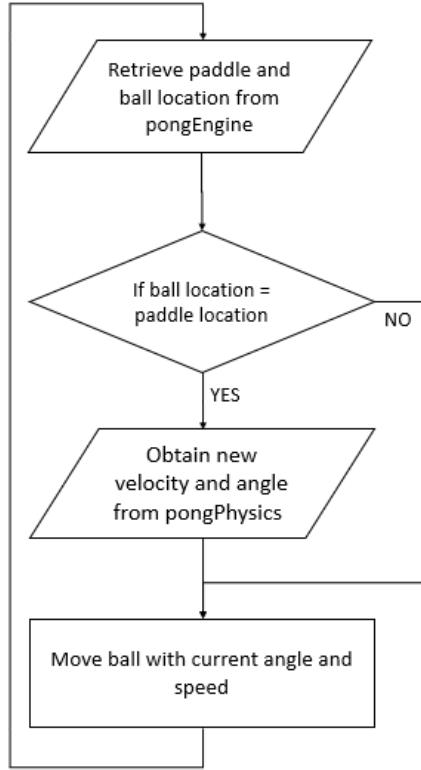
Figure 16: showing 20 degree collision with border and subsequent rebound

The code for the paddle collisions is more complicated than that for border collision although it is similar. In much the same way, the check for the border collision takes place in the while loop within pongScreens. Defined as 15 pixels in front of the paddle (whose length can be varied), when a collision check occurs, a new value for the speed and the angle is obtained from pongPhysics_borderCollision and used to move the ball until a new collision is made. This is done by subtracting the incoming angle from 180 degrees and adding to it a value called deltaAngle. deltaAngle is determined by the point on the paddle with which the ball collides, and is either 0, 10 or 35 (or their negative equivalent) depending on the sector with which the ball collides. A section of code is then used to make sure the angle stays within a constant frame of reference. In much the same way, a new velocity is calculated depending on what sector the ball collides with the velocities from sectors 2 and 3 being increasingly large. The inbound speed has no bearing on the calculated outbound speed. The angle and outbound speed are instantiated as a pointer-array and returned to pongScreens. The array is then read there and the new velocity and angle used to move the ball.

Two flowcharts of the code that regulates the collisions are shown below :



(a) Flowchart for new angle/speed calculation



(b) Flowchart for collision checks

Figure 17: flowcharts of the code that regulates the collisions

6 Problems Encountered

This sections details some of the many bugs that were encountered in the creation of the physics engine and the solutions that were employed in fixing them.

6.1 Ball becoming stuck at border/ paddle

This was one of the earliest bugs encountered in the creation of the physics engine. When the ball collided with the paddle or the game border, it would become stuck, breaking the game and requiring a reset. This was caused by the direction (angle) of the ball constantly changing to the new value and then resetting to the new value as the ball remained within the collision space. This was fixed by introducing the pongEngine_moveBall command into the pongPhysics_borderCollision function. This ‘kicks’ the ball out of the collision space before allowing the pongEngine_moveBall function within the while loop to continue moving the ball afterwards.

6.2 Ball moving through paddle at collision

This bug has arisen several times throughout the development of the physics engine. Simply put, the ball is not affected by the collision boundaries at the paddle wall and instead travels straight through the paddle. Its difficult to predict when this happen and therefore difficult to step through the code and test for it, however it seems to only happen infrequently as of the latest build. Eariler versions of the bug seem to have been caused by the new angle calculations being incorrect as the calculated value exceeded 360. Introducing code that keeps the angle calculations within a constant frame of reference (ie between 0 and 180/-180) seems to have drastically reduced its frequency, although it can still happen on occasion.

6.3 Ball disappearing after paddle collision

his bug was particularly difficult to resolve even though the fix was ultimately quite simple. When the ball collided with a paddle wall it would sometimes randomly disappear without forewarning. Again, this was quite difficult to test for due to the unpredictable nature of the bug. Using breakpoints in the code eventually uncovered the problem. Sometimes when a paddle collision was made, the ball would collide with a part of the paddle not covered by the series of if-else statements in the pongPhysics_paddleCollision code. As such, no value was assigned to the variable ‘outspeed’ when it was returned to the pongScreens file. A nonsensical random value, usually quite large, was instead assigned, causing the ball to have an impossibly high speed and disappear from view. This was fixed by initialising the ‘outspeed’ variable with 0, and introducing an ‘else’ statement that would cover what to do in such a situation.

7 Conclusion

To conclude, a version of the well known game Pong was programmed using the DE1-SoC Development Board from Terasic. Code was written in C and compiled and debugged using the ARM DS-5 suite of software. The design incorporated the use of a range of hardware peripherals and outputs including VGA, LCD, PS/2, audio jack and 7-segment display. The game has been designed to be played either single player against a simple AI or against another player. A menu system has been created allowing the player to select the mode they wish to play in as well as the size of the paddles in the game. The coding architecture and general approach to the project have been described as have the bugs/problems encountered during the course of the project.

The code for the game was extensively tested and debugged through hardware testing. Generally the game is quite robust and functions well when subjected to testing. Nevertheless a few bugs still exist, particularly with regards to the ball's behaviour when collisions occur within the game. Another bug would be screen tearing because of the multiple small frame buffers. Further testing and debugging should be conducted to address these issues.

8 Appendix

8.1 Appendix A - sevenSeg.c

```
1 // This library allows the control of the 7 segment display
2 #include "sevenSeg.h"
3 //Hex numbers in an array
4 int HexSDisplay[16] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x77,0x7C,0x39,0x5E,0x79,0x71};
5
6 //This function sets the 7 segment displays
7 void SDisplay_set(int Display, int HexValue){
8     //Hex memory base
9     volatile int *HEX0 = (int*) 0xFF200020;
10    volatile int *HEX1 = (int*) 0xFF200030;
11    //Values
12    int invClearBits = 0x7F; //inverted bits to put through and bitwise
13    int shiftAmount = 8; //shift multiple amount
14    int hex1Adjust = 4; //adjust amount for second memory address
15    //check each number position to work out between addresses
16    if(Display < 4){
17        *HEX0 &= ~(invClearBits << (Display * shiftAmount)); //clear bits
18        *HEX0 |= (HexSDisplay[HexValue] << (Display * shiftAmount)); //set bits
19    }
20    else{
21        *HEX1 &= ~(invClearBits << ((Display - hex1Adjust) * shiftAmount)); //clear bits
22        *HEX1 |= (HexSDisplay[HexValue] << ((Display - hex1Adjust) * shiftAmount)); //set bits
23    }
24 }
25 //This function clears the seven seg displays
26 void SDisplay_clearAll(){
27     //Hex memory base
28     volatile int *HEX0 = (int*) 0xFF200020;
29     volatile int *HEX1 = (int*) 0xFF200030;
30     int zero = 0x00; //Zero
31     //set displays
32     *HEX0 &= zero; //clear bits
33     *HEX1 &= zero; //clear bits
34 }
35 //This function sets a number per pair of 7 segment displays
36 void SDisplay_PNum(int number, int pair){
37     //Example is 15
38     //Split number in 2
39     int number0 = number/10; //1
40     int number1 = number % 10; //2
41     //calculate pair of displays from the pair set in the function call
42     int pairdisplay0 = 2*pair;
43     int pairdisplay1 = 2*pair+1;
44     //set displays
45     SDisplay_set(pairdisplay0, number1);
46     SDisplay_set(pairdisplay1, number0);
47 }
```

8.2 Appendix B - sevenSeg.h

```
1 #ifndef SEVENSEG_H
2 #define SEVENSEG_H
3
4 void SDISPLAY_PNum(int number, int pair);
5 void SDISPLAY_clearAll( void );
6 void SDISPLAY_set(int Display, int HexValue);
7
8 #endif
```

8.3 Appendix C - DE1SoC VGA.c

```
1 #include "DE1SoC_VGA.h"
2
3 short vga_address = 0x0; //0xC8000000
4 short vga_charaddr = 0x0; //0xC9000000
5 volatile int pixel_buffer_start;
6
7 //Initialise addresses of VGA
8 void VGA_init(short vgaPixelBuffer_address, short vgaCharBuffer_address){
9     //init backbuffer
10    volatile int * pixel_ctrl_ptr = (int *) 0xFF203020;
11    *(pixel_ctrl_ptr + 1) = 0xC8000000;
12    pixel_buffer_start = *pixel_ctrl_ptr;
13    *(pixel_ctrl_ptr + 1) = 0xC0000000;
14    vga_address = vgaPixelBuffer_address;
15    vga_charaddr = vgaCharBuffer_address;
16    //Clear VGA
17    VGA_clearScreen();
18 }
19 //This function gets the address of a pixel and sets it to a colour
20 void VGA_drawPixel(int x, int y, short colour){
21     //Call address of pixel
22     //Address base + [Pixel Y][Pixel X]
23     volatile short *vga_addr=(volatile short*)(0xC8000000 + (y<<10) + (x<<1));
24     *vga_addr=colour; //Set pixel to colour
25 }
26 //This function sets all pixels to black
27 void VGA_clearScreen(){
28     int x, y;
29     for (x = 0; x < 320; x++) {
30         for (y = 0; y < 240; y++) {
31             //set all pixels to black
32             VGA_drawPixel(x,y,0x0000);
33         }
34     }
35 }
36 //This function sets all pixels to the desired colour
37 void VGA_fillColour(short colour){
38     int x, y;
39     for (x = 0; x < 320; x++) {
40         for (y = 0; y < 240; y++) {
41             //set all pixels to colour
42             VGA_drawPixel(x,y,colour);
43         }
44     }
45 }
```

8.4 Appendix D - DE1SoC VGA.h

```
1 #ifndef DE1SOC_VGA_H
2 #define DE1SOC_VGA_H
3
4 void VGA_init( short vgaPixelBuffer_address , short vgaCharBuffer_address );
5 void VGA_drawPixel( int x, int y, short colour );
6 void VGA_drawPixelToHwBuffer( int x, int y, short colour );
7 void VGA_BufferSwap( void );
8 void VGA_clearScreen( void );
9 void VGA_fillColour( short colour );
10
11#endif
```

8.5 Appendix E - pongDisplay.c

```
1 // This is the layer of code between the game and the display devices. It utilises a frame buffer.
2 #include "pongDisplay.h"
3 int PixelHeight = 240; // Screen height
4 int PixelWidth = 320; // Screen Width
5 int totalPixel = 76800; // Total Pixels
6 /*--- Global Variables for frame buffers ---*/
7 // Single framebuffer
8 volatile short frontFrameBuffer[320][240];
9 volatile short rearFrameBuffer[320][240];
10 // Quad framebuffer
11 volatile short frontFrameBuffer1[160][120];
12 volatile short rearFrameBuffer1[160][120];
13 volatile short frontFrameBuffer2[160][120];
14 volatile short rearFrameBuffer2[160][120];
15 volatile short frontFrameBuffer3[160][120];
16 volatile short rearFrameBuffer3[160][120];
17 volatile short frontFrameBuffer4[160][120];
18 volatile short rearFrameBuffer4[160][120];
19 // OCTO framebuffer
20 volatile short OCTOfrontFrameBuffer1[80][120];
21 volatile short OCTOrearFrameBuffer1[80][120];
22 volatile short OCTOfrontFrameBuffer2[80][120];
23 volatile short OCTOrearFrameBuffer2[80][120];
24 volatile short OCTOfrontFrameBuffer3[80][120];
25 volatile short OCTOrearFrameBuffer3[80][120];
26 volatile short OCTOfrontFrameBuffer4[80][120];
27 volatile short OCTOrearFrameBuffer4[80][120];
28 volatile short OCTOfrontFrameBuffer5[80][120];
29 volatile short OCTOrearFrameBuffer5[80][120];
30 volatile short OCTOfrontFrameBuffer6[80][120];
31 volatile short OCTOrearFrameBuffer6[80][120];
32 volatile short OCTOfrontFrameBuffer7[80][120];
33 volatile short OCTOrearFrameBuffer7[80][120];
34 volatile short OCTOfrontFrameBuffer8[80][120];
35 volatile short OCTOrearFrameBuffer8[80][120];
36 // defines
37 int modeSet = 1;
38 int frameskip = 0;
39 int framecount = 0;
40 // Safety Window
41 int minX = 0;
42 int minY = 0;
43 int maxX = 320;
44 int maxY = 240;
45
46 // This function initialises the displays by setting the addresses
47 void Displays_init(volatile int vga_PixelAddress, unsigned volatile int vga_CharacterAddress, unsigned volatile int lcd_pio_base,
48                     unsigned volatile int lcd_hw_base){
49     // init VGA
50     VGA_init(vga_PixelAddress, vga_CharacterAddress);
51     // init LP24
52     LT24_initialise(lcd_pio_base, lcd_hw_base);
53     // Set framebuffers black
54 }
55 // This function sets the frameskip amount.
56 void Displays_frameSkip(int skipamount){
57     frameskip = skipamount;
58 }
59 // This function sets the display mode and then initialises the values.
60 void Displays_mode(int mode){
61     int x = 0; // counter for x
62     int y = 0; // counter for y
63     modeSet = mode; // set global variable
64     // if single software buffer
65     if(modeSet == SOFTWAREFB){
66         // for loops to set values to zero in 2 dimensional array
67         for(y = 0; y < PixelHeight; y++){
68             for(x = 0; x < PixelWidth; x++){
69                 frontFrameBuffer[x][y] = 0x000;
70                 rearFrameBuffer[x][y] = 0x000;
71             }
72         }
73     }
74     // if software buffer split into 4
75     else if(modeSet == SOFTWAREQUADFB || modeSet == FASTFB){
76         // for loops to set values to zero in 2 dimensional array
77         for(y = 0; y < 120; y++){
78             for(x = 0; x < 160; x++){
79                 frontFrameBuffer1[x][y] = 0x000;
80                 rearFrameBuffer1[x][y] = 0x000;
81                 frontFrameBuffer2[x][y] = 0x000;
82                 rearFrameBuffer2[x][y] = 0x000;
83                 frontFrameBuffer3[x][y] = 0x000;
84                 rearFrameBuffer3[x][y] = 0x000;
85                 frontFrameBuffer4[x][y] = 0x000;
86                 rearFrameBuffer4[x][y] = 0x000;
87             }
88         }
89     }
90     // if software buffer split into 8
91     else if (modeSet == SOFTWAREOCTOFB){
92         // for loops to set values to zero in 2 dimensional array
93         for(y = 0; y < 120; y++){
```

```

94     for(x = 0; x < 160; x++){
95         OCTOfrontFrameBuffer1[x][y] = 0x000;
96         OCTOrearFrameBuffer1[x][y] = 0x000;
97         OCTOfrontFrameBuffer2[x][y] = 0x000;
98         OCTOrearFrameBuffer2[x][y] = 0x000;
99         OCTOfrontFrameBuffer3[x][y] = 0x000;
100        OCTOrearFrameBuffer3[x][y] = 0x000;
101        OCTOfrontFrameBuffer4[x][y] = 0x000;
102        OCTOrearFrameBuffer4[x][y] = 0x000;
103        OCTOfrontFrameBuffer5[x][y] = 0x000;
104        OCTOrearFrameBuffer5[x][y] = 0x000;
105        OCTOfrontFrameBuffer6[x][y] = 0x000;
106        OCTOrearFrameBuffer6[x][y] = 0x000;
107        OCTOfrontFrameBuffer7[x][y] = 0x000;
108        OCTOrearFrameBuffer7[x][y] = 0x000;
109        OCTOfrontFrameBuffer8[x][y] = 0x000;
110        OCTOrearFrameBuffer8[x][y] = 0x000;
111    }
112 }
113 }
114 }
115 //This function set the pixel to the hardware
116 void Displays_drawPixel(int x, int y, short colour){
117     VGA_drawPixel(x,y, colour); //Set pixel on VGA hardware
118     LT24_drawPixel(colour,240-y,x); //Set pixel on LT24 hardware
119     ResetWDT(); //reset watchdog
120 }
121
122 // This function clears the screen
123 void Displays_clearScreen(){
124     VGA_clearScreen(); //clear screen VGA
125     LT24_clearDisplay(0x000); //clear screen LT24
126     Displays_fillColour(0); //set frame buffer values to 0
127 }
128
129 // This function refreshes the screens from the framebuffer and checks the mode.
130 void Displays_Refresh(){
131     //check the mode
132     if(frameskip==0){
133         if(modeSet == 0){
134             }
135         else if(modeSet == SOFTWAREFB){
136             DisplaysLocal_singleRefresh();
137         }
138         else if(modeSet == SOFTWAREQUADFB){
139             DisplaysLocal_quadRefresh();
140         }
141         else if(modeSet == SOFTWAREOCTOFB){
142             DisplaysLocal_octoRefresh();
143         }
144         else if(modeSet == HARDWAREFB){
145             }
146     }
147     else{
148         if(frameskip == framecount){
149             framecount = 0;
150             if(modeSet == 0){
151                 }
152             else if(modeSet == SOFTWAREFB){
153                 DisplaysLocal_singleRefresh();
154             }
155             else if(modeSet == SOFTWAREQUADFB){
156                 DisplaysLocal_quadRefresh();
157             }
158             else if(modeSet == SOFTWAREOCTOFB){
159                 DisplaysLocal_octoRefresh();
160             }
161             else if(modeSet == HARDWAREFB){
162                 }
163         }
164     }
165 }
166
167 else{
168     framecount++;
169 }
170 }
171 }
172
173 //This forces a refresh regardless of the frameskip value
174 void Displays_forceRefresh(){
175     if(modeSet == 0){
176         }
177         else if(modeSet == SOFTWAREFB){
178             DisplaysLocal_singleRefresh();
179         }
180         else if(modeSet == SOFTWAREQUADFB){
181             DisplaysLocal_quadRefresh();
182         }
183         else if(modeSet == SOFTWAREOCTOFB){
184             DisplaysLocal_octoRefresh();
185         }
186         else if(modeSet == HARDWAREFB){
187             }
188     }
189     else{
190

```

```

191     if (modeSet == 0){
192     }
193     else if (modeSet == SOFTWAREFB){
194         DisplaysLocal_singleRefresh();
195     }
196     else if (modeSet == SOFTWAREQUADFB){
197         DisplaysLocal_quadRefresh();
198     }
199     else if (modeSet == SOFTWAREOCTOFB){
200         DisplaysLocal_octoRefresh();
201     }
202     else if (modeSet == HARDWAREFB){
203     }
204     }
205 }
206 }
207 }
208
//This sets the display window which makes sure that the display can not set outside of these values which prevents errors
209 void Displays_setWindow(int minx, int miny, int maxx, int maxy){
210     minX = minx;
211     minY = miny;
212     maxX = maxx;
213     maxY = maxy;
214 }
215
//This sets the pixel to the framebuffer
216 void Displays_setPixel(int x, int y, short colour){
217     //check if within window
218     int z;
219     int found = 0;
220     //checks if within window set
221     if(x >= minX && x < maxX && y >= minY && y < maxY){
222         //if no framebuffer draw straight to the display hardware
223         if(modeSet == NOFRAMEBUFFER){
224             Displays_drawPixel(x,y,colour);
225         }
226         //if single framebuffer then set the buffer
227         else if (modeSet == SOFTWAREFB){
228             frontFrameBuffer[x][y] = colour;
229         }
230         //if quad framebuffer then check which segment of screen pixel in and set framebuffer
231         else if (modeSet == SOFTWAREQUADFB){
232             if(x<160 && y<120){
233                 frontFrameBuffer1[x][y] = colour;
234             }
235             else if(x>=160 && y<120){
236                 frontFrameBuffer2[x-160][y] = colour;
237             }
238             else if(x<160 && y>=120){
239                 frontFrameBuffer3[x][y-120] = colour;
240             }
241             else if(x>=160 && y>=120){
242                 frontFrameBuffer4[x-160][y-120] = colour;
243             }
244         }
245     }
246     //if octo framebuffer then check which segment of screen pixel in and set framebuffer
247     else if (modeSet == SOFTWAREOCTOFB){
248         if(x<80 && y<120){
249             OCTOfrontFrameBuffer1[x][y] = colour;
250         }
251         else if(x>=80 && y<120 && x<160){
252             OCTOfrontFrameBuffer2[x-80][y] = colour;
253         }
254         else if(x>=160 && y<120 && x<240){
255             OCTOfrontFrameBuffer3[x-160][y] = colour;
256         }
257         else if(x>=240 && y<120){
258             OCTOfrontFrameBuffer4[x-240][y] = colour;
259         }
260         else if(x<80 && y>=120){
261             OCTOfrontFrameBuffer5[x][y-120] = colour;
262         }
263         else if(x>=80 && y>=120 && x<160){
264             OCTOfrontFrameBuffer6[x-80][y-120] = colour;
265         }
266         else if(x>=160 && y>=120 && x<240){
267             OCTOfrontFrameBuffer7[x-160][y-120] = colour;
268         }
269         else if(x>=240 && y>=120){
270             OCTOfrontFrameBuffer8[x-240][y-120] = colour;
271         }
272     }
273 }
274 }
275 }
276
//This gets the pixel set on the display.
277 short Displays_getPixel(int x, int y){
278     short pixel = 0x000;
279     if(modeSet == 0){
280         pixel = 0x000;
281     }
282     else if (modeSet == SOFTWAREFB){
283         pixel = rearFrameBuffer[x][y];
284     }
285     else if (modeSet == SOFTWAREQUADFB || modeSet == FASTFB){
286         if(x<160 && y<120){

```

```

288     pixel = frontFrameBuffer1[x][y];
289 }
290 else if (x>=160 && y<120){
291     pixel = frontFrameBuffer2[x-160][y];
292 }
293 else if (x<160 && y>=120){
294     pixel = frontFrameBuffer3[x][y-120];
295 }
296 else if (x>=160 && y>=120){
297     pixel = frontFrameBuffer4[x-160][y-120];
298 }
299 }
300 else if (modeSet == SOFTWAREOCTOFB){
301     if(x<80 && y<120){
302         pixel = OCTOfrontFrameBuffer1[x][y];
303     }
304     else if(x>=80 && y<120 && x<160){
305         pixel = OCTOfrontFrameBuffer2[x-80][y];
306     }
307     else if(x>=160 && y<120 && x<240){
308         pixel = OCTOfrontFrameBuffer3[x-160][y];
309     }
310     else if(x>=240 && y<120){
311         pixel = OCTOfrontFrameBuffer4[x-240][y];
312     }
313     else if(x<80 && y>=120){
314         pixel = OCTOfrontFrameBuffer5[x][y-120];
315     }
316     else if(x>=80 && y>=120 && x<160){
317         pixel = OCTOfrontFrameBuffer6[x-80][y-120];
318     }
319     else if(x>=160 && y>=120 && x<240){
320         pixel = OCTOfrontFrameBuffer7[x-160][y-120];
321     }
322     else if(x>=240 && y>=120){
323         pixel = OCTOfrontFrameBuffer8[x-240][y-120];
324     }
325 }
326 return pixel;
327 }
328
329 //FrameBuffer Refreshes
330 //This function refreshes the octo framebuffer
331 void DisplaysLocal_octoRefresh(){
332     int x; //x counter
333     int y; //y counter
334     //compare front and rear buffers using memcmp
335     int o1change = memcmp(OCTOfrontFrameBuffer1,OCTOrearFrameBuffer1, sizeof(OCTOfrontFrameBuffer1));
336     int o2change = memcmp(OCTOfrontFrameBuffer2,OCTOrearFrameBuffer2, sizeof(OCTOfrontFrameBuffer2));
337     int o3change = memcmp(OCTOfrontFrameBuffer3,OCTOrearFrameBuffer3, sizeof(OCTOfrontFrameBuffer3));
338     int o4change = memcmp(OCTOfrontFrameBuffer4,OCTOrearFrameBuffer4, sizeof(OCTOfrontFrameBuffer4));
339     int o5change = memcmp(OCTOfrontFrameBuffer5,OCTOrearFrameBuffer5, sizeof(OCTOfrontFrameBuffer5));
340     int o6change = memcmp(OCTOfrontFrameBuffer6,OCTOrearFrameBuffer6, sizeof(OCTOfrontFrameBuffer6));
341     int o7change = memcmp(OCTOfrontFrameBuffer7,OCTOrearFrameBuffer7, sizeof(OCTOfrontFrameBuffer7));
342     int o8change = memcmp(OCTOfrontFrameBuffer8,OCTOrearFrameBuffer8, sizeof(OCTOfrontFrameBuffer8));
343     //if any changes whilst running through loop then update frame buffer and set on hardware.
344     for(y = 0; y < PixelHeight/2; y++){
345         for(x = 0; x < PixelWidth/4; x++){
346             if(o1change != 0){
347                 if(OCTOfrontFrameBuffer1[x][y] != OCTOrearFrameBuffer1[x][y]){
348                     Displays_drawPixel(x,y,OCTOfrontFrameBuffer1[x][y]); //set on hardware
349                     OCTOrearFrameBuffer1[x][y] = OCTOfrontFrameBuffer1[x][y]; //update FB
350                 }
351             }
352             if(o2change != 0){
353                 if(OCTOfrontFrameBuffer2[x][y] != OCTOrearFrameBuffer2[x][y]){
354                     Displays_drawPixel(x+80,y,OCTOfrontFrameBuffer2[x][y]); //set on hardware
355                     OCTOrearFrameBuffer2[x][y] = OCTOfrontFrameBuffer2[x][y]; //update FB
356                 }
357             }
358             if(o3change != 0){
359                 if(OCTOfrontFrameBuffer3[x][y] != OCTOrearFrameBuffer3[x][y]){
360                     Displays_drawPixel(x+160,y,OCTOfrontFrameBuffer3[x][y]); //set on hardware
361                     OCTOrearFrameBuffer3[x][y] = OCTOfrontFrameBuffer3[x][y]; //update FB
362                 }
363             }
364             if(o4change != 0){
365                 if(OCTOfrontFrameBuffer4[x][y] != OCTOrearFrameBuffer4[x][y]){
366                     Displays_drawPixel(x+240,y,OCTOfrontFrameBuffer4[x][y]); //set on hardware
367                     OCTOrearFrameBuffer4[x][y] = OCTOfrontFrameBuffer4[x][y]; //update FB
368                 }
369             }
370             if(o5change != 0){
371                 if(OCTOfrontFrameBuffer5[x][y] != OCTOrearFrameBuffer5[x][y]){
372                     Displays_drawPixel(x,y+120,OCTOfrontFrameBuffer5[x][y]); //set on hardware
373                     OCTOrearFrameBuffer5[x][y] = OCTOfrontFrameBuffer5[x][y]; //update FB
374                 }
375             }
376             if(o6change != 0){
377                 if(OCTOfrontFrameBuffer6[x][y] != OCTOrearFrameBuffer6[x][y]){
378                     Displays_drawPixel(x+80,y+120,OCTOfrontFrameBuffer6[x][y]); //set on hardware
379                     OCTOrearFrameBuffer6[x][y] = OCTOfrontFrameBuffer6[x][y]; //update FB
380                 }
381             }
382             if(o7change != 0){
383                 if(OCTOfrontFrameBuffer7[x][y] != OCTOrearFrameBuffer7[x][y]){
384                     Displays_drawPixel(x+160,y+120,OCTOfrontFrameBuffer7[x][y]); //set on hardware

```

```

385     OCTOrearFrameBuffer7[x][y] = OCTOfrontFrameBuffer7[x][y]; //update FB
386   }
387 }
388 if(o8change != 0){
389   if(OCTOfrontFrameBuffer8[x][y] != OCTOrearFrameBuffer8[x][y]){
390     Displays_drawPixel(x+240,y+120,OCTOfrontFrameBuffer8[x][y]); //set on hardware
391     OCTOrearFrameBuffer8[x][y] = OCTOfrontFrameBuffer8[x][y]; //update FB
392   }
393 }
394 }
395 }
396 }
397 }
398
399 //This function refreshes the quad framebuffer
400 void DisplaysLocal_quadRefresh(){
401   int x; //x counter
402   int y; //y counter
403   //compare front and rear buffers using memcmp
404   int q1change = memcmp(frontFrameBuffer1,rearFrameBuffer1, sizeof(frontFrameBuffer1));
405   int q2change = memcmp(frontFrameBuffer2,rearFrameBuffer2, sizeof(frontFrameBuffer2));
406   int q3change = memcmp(frontFrameBuffer3,rearFrameBuffer3, sizeof(frontFrameBuffer3));
407   int q4change = memcmp(frontFrameBuffer4,rearFrameBuffer4, sizeof(frontFrameBuffer4));
408   //if any changes whilst running through loop then update frame buffer and set on hardware.
409   for(y = 0; y < PixelHeight/2; y++){
410     for(x = 0; x < PixelWidth/2; x++){
411       if(q1change != 0){
412         if(frontFrameBuffer1[x][y] != rearFrameBuffer1[x][y]){
413           Displays_drawPixel(x,y,frontFrameBuffer1[x][y]); //set on hardware
414           rearFrameBuffer1[x][y] = frontFrameBuffer1[x][y]; //update FB
415         }
416       }
417       if(q2change != 0){
418         if(frontFrameBuffer2[x][y] != rearFrameBuffer2[x][y]){
419           Displays_drawPixel(x+160,y,frontFrameBuffer2[x][y]); //set on hardware
420           rearFrameBuffer2[x][y] = frontFrameBuffer2[x][y]; //update FB
421         }
422       }
423       if(q3change != 0){
424         if(frontFrameBuffer3[x][y] != rearFrameBuffer3[x][y]){
425           Displays_drawPixel(x,y+120,frontFrameBuffer3[x][y]); //set on hardware
426           rearFrameBuffer3[x][y] = frontFrameBuffer3[x][y]; //update FB
427         }
428       }
429       if(q4change != 0){
430         if(frontFrameBuffer4[x][y] != rearFrameBuffer4[x][y]){
431           Displays_drawPixel(x+160,y+120,frontFrameBuffer4[x][y]); //set on hardware
432           rearFrameBuffer4[x][y] = frontFrameBuffer4[x][y]; //update FB
433         }
434       }
435     }
436   }
437 }
438 //This function find changes and sets them in hardware.
439 void DisplaysLocal_singleRefresh(){
440   int x;
441   int y;
442   for(y = 0; y < PixelHeight; y++){
443     for(x = 0; x < PixelWidth; x++){
444       short colour = frontFrameBuffer[x][y];
445       if(frontFrameBuffer[x][y] != rearFrameBuffer[x][y]){
446         short colour = frontFrameBuffer[x][y];
447         Displays_drawPixel(x,y,colour);
448         rearFrameBuffer[x][y] = colour;
449       }
450     }
451   }
452 }
453 }
454 //This function fills the displays a set colour.
455 void Displays_fillColour(short colour){
456   int x;
457   int y;
458   VGA_fillColour(colour);
459   LT24_clearDisplay(colour);
460   for(y = 0; y < 240; y++){
461     for(x = 0; x < 320; x++){
462       Displays_setPixel(x,y,colour);
463     }
464   }
465 }
```

8.6 Appendix F - pongDisplay.h

```
1 #ifndef PONGDISPLAY_H
2 #define PONGDISPLAY_H
3
4 #include "../DE1Soc_LT24/DE1Soc_LT24.h"
5 #include "../DE1Soc_VGA/DE1Soc_VGA.h"
6 #include "../HPS_Watchdog/HPS_Watchdog.h"
7 #include <string.h>
8
9 #define NOFRAMEBUFFER 0
10 #define SOFTWAREFB 1
11 #define SOFTWAREQUADFB 2
12 #define SOFTWAREOCTOFB 3
13 #define HARDWAREFB 4
14 #define FASTFB 5
15
16 void Displays_init(volatile int vga_PixelAddress, unsigned volatile int vga_CharacterAddress, unsigned volatile int lcd_pio_base,
17                     unsigned volatile int lcd_hw_base);
18 void Displays_drawPixel(int x, int y, short colour);
19 void Displays_frameSkip(int skipamount);
20 void Displays_clearScreen( void );
21 void Displays_Refresh( void );
22 void Displays_forceRefresh( void );
23 void Displays_setPixel(int x, int y, short colour);
24 void Displays_setWindow(int minx, int miny, int maxx, int maxy);
25 short Displays_getPixel(int x, int y);
26 void Displays_mode(int mode);
27 void Displays_fillColour(short colour);
28 void Displays_mode(int mode);
29 // Functions not intended to be used outside of pongDisplay.cpp
30
31 void DisplaysLocal_singleRefresh( void );
32 void DisplaysLocal_quadRefresh( void );
33 void DisplaysLocal_octoRefresh( void );
34 #endif
```

8.7 Appendix G - pongEngine.c

```
1 #include "pongEngine.h"
2
3 //320x240
4
5 /*##### Ball Globals #####*/
6 // Calculated ball path
7 struct ballPathInst{
8     int x;
9     int y;
10 };
11 //instruction counter
12 int ballPathInstCounter = 0;
13 //current instruction
14 int ballCurrentPosPath = 0;
15 //array of structs
16 struct ballPathInst BallPath[1000];
17
18 int ballAngle = 361; //Set to impossible angle to ensure on initialisation there
19 short ballColour = 0xFFFF; //Ball colour
20 int ballX = centre_x; //ball defaults
21 int ballY = centre_y; //ball defaults
22 //Thresholds
23 int ballMinX = 8;
24 int ballMaxX = 312;
25 int ballMinY = 23;
26 int ballMaxY = 232;
27
28 /*##### Paddle Globals #####*/
29 //Paddle thresholds
30 int paddleMinY = 50;
31 int paddleMaxY = 210;
32 //Paddle Locations
33 int paddle1Y = centre_y;
34 int paddle1X = 50;
35 int paddle2Y = centre_y;
36 int paddle2X = 270;
37 //Paddle Colours
38 int paddle1Colour = 0xFFFF;
39 int paddle2Colour = 0xFFFF;
40 /*##### Score Keeping Globals#####*/
41 int player1Score = 0;
42 int player2Score = 0;
43 /*##### General Functions #####*/
44 //This function initialises the pong engine
45 void pongEngine_init(){
46     int topAdjust = 15; //VGA Pixel faults from y
47     Displays_clearScreen(); //clear screen
48     pongEngine_resetPaddles(); //reset paddles
49     pongEngine_resetBallLoc(); //reset ball location
50
51     pongEngine_paddleCreate(1); //create paddle
52     pongEngine_paddleCreate(2); //create paddle
53
54     //create arena
55     //create green area
56     pongSprites_renderRectangle(320.0, topAdjust, 0, (0x1F << 11)); //to create red border on LCD
57     paddleMinY = pongSprites_getPaddleSizeY()/2; //set limits
58     paddleMaxY = 240-paddleMinY; //set limits
59
60     pongEngine_paddleSetYLimits(paddleMaxY, paddleMinY+topAdjust+1); //adjust limits
61 }
62 /*##### Score Keeping #####*/
63 //This function adds points to a player
64 void pongEngine_addPoint(int player){
65     //check player and add point to player
66     if(player == 1){
67         player1Score++;
68     }
69     if(player == 2){
70         player2Score++;
71     }
72     //refresh score
73     pongEngine_refreshScore();
74 }
75 //This function resets the score to zero
76 void pongEngine_resetScore(unsigned int _Refresh){
77     player1Score = 0;
78     player2Score = 0;
79     if (_Refresh != 0){
80         pongEngine_refreshScore(); //refresh score
81     }
82 }
83
84 //refresh score to board seven seg and on screen
85 void pongEngine_refreshScore(){
86     //create char
87     char playerscorech[6];
88     //split numbers
89     int P1number0, P1number1, P2number0, P2number1;
90
91     // Fix overflow past 100 - loops to zero
92     player1Score = player1Score %100;
93     player2Score = player2Score %100;
```

```

95 // split numbers into individual characters
96 P1number0 = player1Score/10; //1
97 P1number1 = player1Score % 10; //5
98 P2number0 = player2Score/10; //1
99 P2number1 = player2Score % 10; //5
100 //updates 7 seg
101 SDisplay_PNum(player1Score ,2);
102 SDisplay_PNum(player2Score ,0);
103 //updates score
104 playerscorech[0] = P1number0 + '0';
105 playerscorech[1] = P1number1 + '0';
106 playerscorech[2] = ' ';
107 playerscorech[3] = P2number0 + '0';
108 playerscorech[4] = P2number1 + '0';
109 playerscorech[5] = '0';
110 //clears score by drawing black rectangle on it
111 pongSprites_renderRectangle(centre_x - 30,centre_x+ 30 , 17, 17+14, 0x00);
112 //writes score on screen
113 pongSprites_writeText(centre_x - 28, 17, 1, playerscorech , 0xFFFF);
114 //Displays_forceRefresh();
115 }
116
117 /*##### Ball Functions #####*/
118 //This function moves the pong ball
119 void pongEngine_moveBall(int angle, int speed){
120 ResetWDT(); //reset watchdog
121 pongEngine_destroyBall(); //destroy the ball so its black
122 //Check if at thresholds
123 if(ballX < ballMinX){
124 ballX = ballMinX;
125 ballCurrentPosPath = ballCurrentPosPath-speed;
126 }
127 if(ballX > ballMaxX){
128 ballX = ballMaxX;
129 ballCurrentPosPath = ballCurrentPosPath-speed;
130 }
131 if(ballY < ballMinY){
132 ballY = ballMinY;
133 ballCurrentPosPath = ballCurrentPosPath-speed;
134 }
135 if(ballY > ballMaxY){
136 ballY = ballMaxY;
137 ballCurrentPosPath = ballCurrentPosPath-speed;
138 }
139 //If angle change then calculate new instructions
140 if(angle == ballAngle){
141 //To prevent overflow
142 if(ballCurrentPosPath > ballPathInstCounter){
143 ballCurrentPosPath = ballPathInstCounter;
144 }
145 //render ball and move position
146 ballX = BallPath[ballCurrentPosPath].x;
147 ballY = BallPath[ballCurrentPosPath].y;
148 pongSprites_renderBall(ballX , ballY , ballColour);
149 ResetWDT();
150 //Move ball speed amount
151 ballCurrentPosPath = ballCurrentPosPath+speed;
152 }
153 //if ball is at a different angle
154 else{
155 ballAngle = angle; //set ballAngle to angle
156 ResetWDT();
157 //Calculate new path
158 pongEngine_calcBallPathInst(angle);
159 //render ball and move position
160 ballX = BallPath[ballCurrentPosPath].x;
161 ballY = BallPath[ballCurrentPosPath].y;
162 pongSprites_renderBall(ballX , ballY , ballColour);
163 //Move ball speed amount
164 ballCurrentPosPath = ballCurrentPosPath+speed;
165 }
166 }
167 //reset ball location
168 void pongEngine_resetBallLoc( void ){
169 //pongEngine_setBallLocation(centre_x , centre_y); // Avoiding extra ball on startup
170 ballX = centre_x;
171 ballY = centre_y;
172 }
173 //set ball location
174 void pongEngine_setBallLocation(int x, int y){
175 pongEngine_destroyBall();
176 ResetWDT();
177 ballX = x;
178 ballY = y;
179 pongEngine_calcBallPathInst(ballAngle);
180 pongSprites_renderBall(ballX , ballY , ballColour);
181 ballCurrentPosPath++;
182 }
183 //get ball angle
184 int pongEngine_getBallAngle(void){
185 return ballAngle;
186 }
187 //get ball location x
188 int pongEngine_getBallLocation_x(void){
189 return ballX;
190 }
191 //get ball location y

```

```

192 int pongEngine_getBallLocation_y(void){
193     return ballY;
194 }
195 //create the ball
196 void pongEngine_createBall(void){
197     pongSprites_renderBall(ballX, ballY, ballColour);
198 }
199 //destroy the ball
200 void pongEngine_destroyBall(void){
201     pongSprites_renderBall(ballX, ballY, 0x0000);
202 }
203 //calculate the ball path instructions
204 void pongEngine_calcBallPathInst(int angle){
205     int x;
206     int newangle;
207     int foundAngle = 0;
208     int angleX;
209     int angleY;
210     //Prevent floating point error at right angles.
211     if(angle == 0){
212         foundAngle = 1;
213         angleX = 0;
214         angleY = ballY;
215     }
216     if(angle == 90){
217         foundAngle = 1;
218         angleX = ballX;
219         angleY = 0;
220     }
221     if(angle == 180){
222         foundAngle = 1;
223         angleX = 320;
224         angleY = ballY;
225     }
226     if(angle == 270){
227         foundAngle = 1;
228         angleX = ballX;
229         angleY = 240;
230     }
231     //Finds angle by working out angle between 2 points (one being the centre of ball, one being edge of screen)
232     //if angle found do not calculate the angle
233     if(foundAngle == 0){
234         for(x = 0; x<240; x++){
235             newangle = pongEngine_calcAngle(ballX, ballY, 0, x); //calculate angle
236             if(newangle == angle){
237                 foundAngle = 1;
238                 angleX = 0;
239                 angleY = x;
240             }
241         }
242     }
243     if(foundAngle == 0){
244         for(x = 0; x<240; x++){
245             newangle = pongEngine_calcAngle(ballX, ballY, 320, x); //calculate angle
246             if(newangle == angle){
247                 foundAngle = 1;
248                 angleX = 320;
249                 angleY = x;
250             }
251         }
252     }
253     if(foundAngle == 0){
254         for(x = 0; x<320; x++){
255             newangle = pongEngine_calcAngle(ballX, ballY, x, 0); //calculate angle
256             if(newangle == angle){
257                 foundAngle = 1;
258                 angleX = x;
259                 angleY = 0;
260             }
261         }
262     }
263     if(foundAngle == 0){
264         for(x = 0; x<320; x++){
265             newangle = pongEngine_calcAngle(ballX, ballY, x, 240); //calculate angle
266             if(newangle == angle){
267                 foundAngle = 1;
268                 angleX = x;
269                 angleY = 240;
270             }
271         }
272     }
273     //generate ball instructions
274     pongEngine_genBallPathInst(ballX, ballY, angleX, angleY);
275 }
276
277 //Line drawing code from Assignment 1 – Alexander Bolton
278 void pongEngine_genBallPathInst(int x1, int y1, int x2, int y2){
279     //REFERENCE: drawLine using Bresenham's algorithm. https://rosettacode.org/wiki/Bitmap/Bresenham%27s\_line\_algorithm
280     //reset ballPathCounts
281
282     //calculate deltas
283     int dx = abs(x2 - x1);
284     int dy = -abs(y2 - y1);
285     //calculate error
286     int error = dx + dy;
287     int error2;
288     //Find quadrant

```

```

289 int sy;
290 int sx;
291 ballPathInstCounter = 0;
292 ballCurrentPosPath = 0;
293 if(x1<x2){
294     sx = 1;
295 }
296 else{
297     sx = -1;
298 }
299
300 if(y1<y2){
301     sy = 1;
302 }
303 else{
304     sy = -1;
305 }
306 //Loop though and calculate line pixels
307 while(1){
308     //Generate Instructions
309     BallPath[ballPathInstCounter].x = x1;
310     BallPath[ballPathInstCounter].y = y1;
311     ballPathInstCounter++;
312     if (x1 == x2 && y1 == y2){ break;}
313     error2 = 2 * error;
314     //if error2 is larger than delta y then add 1 to x
315     if (error2 >= dy) {
316         error += dy;
317         x1 += sx;
318     }
319     //if error2 is smaller than delta x then add 1 to y
320     if (error2 <= dx) {
321         error += dx;
322         y1 += sy;
323     }
324 }
325 }
326
327 //This function calculates the angle between 2 points
328 int pongEngine_calcAngle(int x1, int y1, int x2, int y2){
329     //calculate deltas
330     double diff_y = y1 - y2;
331     double diff_x = x1 - x2;
332     //calculate angle
333     double angle = atan2(diff_y, diff_x);
334     //turn to degrees
335     float ang_d = angle * 180/PI;
336     //turn to integer
337     int ang_dint = ang_d;
338     //return
339     return ang_dint;
340 }
341 /*#####
342 //set paddle limits on y axis
343 void pongEngine_paddleSetYLimits(int maxy, int miny){
344     paddleMinY = miny;
345     paddleMaxY = maxy;
346 }
347 //set paddle y location
348 void pongEngine_paddleSetYLocation(int player, int y){
349     pongEngine_paddleDestroy(player); //destroy paddle to turn to black
350     //check thresholds
351     if(y>paddleMaxY){
352         y = paddleMaxY;
353     }
354     if(y<paddleMinY){
355         y = paddleMinY;
356     }
357     //check which player and move appropriate paddle
358     if(player == 1){
359         paddle1Y = y;
360         pongSprites_renderPaddle(paddle1X, y, paddle1Colour); //render paddle
361     }
362     if(player == 2){
363         paddle2Y = y;
364         pongSprites_renderPaddle(paddle2X, y, paddle2Colour); //render paddle
365     }
366 }
367
368 //set paddle x location
369 void pongEngine_paddleSetXLocation(int player, int x){
370     pongEngine_paddleDestroy(player); //destroy paddle
371     //check player and set appropriate paddle
372     if(player == 1){
373         paddle1X = x;
374         pongSprites_renderPaddle(x, paddle1Y, paddle1Colour); //render paddle
375     }
376     if(player == 2){
377         paddle2X = x;
378         pongSprites_renderPaddle(x, paddle1Y, paddle2Colour); //render paddle
379     }
380 }
381 //This function moves the paddle
382 void pongEngine_paddleMove(int player, int direction, int speed){
383     /*
384     UP - 1
385     DOWN - 0

```

```

386 */
387 int moveto;
388 //check player
389 if(player == 1){
390     //check direction
391     if(direction == UP){
392         moveto = paddle1Y+speed; //calculate y to move to
393         pongEngine_paddleSetYLocation(1, moveto); //render
394     }
395     if(direction == DOWN){
396         moveto = paddle1Y-speed; //calculate y to move to
397         pongEngine_paddleSetYLocation(1, moveto); //render
398     }
399 }
400 //check player
401 if(player == 2){
402     //check direction
403     if(direction == UP){
404         moveto = paddle2Y+speed; //calculate y to move to
405         pongEngine_paddleSetYLocation(2, moveto); //render
406     }
407     if(direction == DOWN){
408         moveto = paddle2Y-speed; //calculate y to move to
409         pongEngine_paddleSetYLocation(2, moveto); //render
410     }
411 }
412 }
413 //This function creates the paddle
414 void pongEngine_paddleCreate(int player){
415     //check player
416     if(player == 1){
417         pongSprites_renderPaddle(paddle1X, paddle1Y, paddle1Colour); //render paddle
418     }
419     if(player == 2){
420         pongSprites_renderPaddle(paddle2X, paddle2Y, paddle2Colour); //render paddle
421     }
422 }
423 //This function destroys the paddle
424 void pongEngine_paddleDestroy(int player){
425     //check player
426     if(player == 1){
427         pongSprites_renderPaddle(paddle1X, paddle1Y, 0x0000); //render paddle black
428     }
429     if(player == 2){
430         pongSprites_renderPaddle(paddle2X, paddle2Y, 0x0000); //render paddle black
431     }
432 }
433 //This function resets the paddles
434 void pongEngine_resetPaddles(){
435     pongEngine_paddleDestroy(1); //destroy paddles
436     pongEngine_paddleDestroy(2);
437     pongEngine_paddleSetYLocation(1, centre_y); //set paddle location
438     pongEngine_paddleSetYLocation(2, centre_y);
439 }
440 //This function gets the paddle locations
441 unsigned int pongEngine_getPaddleY(unsigned int paddle){
442     if (paddle == 1){
443         return paddle1Y;
444     } else {
445         return paddle2Y;
446     }
447 }
```

8.8 Appendix H - pongEngine.h

```
1 #ifndef PONGENGINE_H
2 #define PONGENGINE_H
3
4 #include <stdlib.h>
5 #include <cmath>
6 #include "pongSprites.h"
7 #include "../DE1SoC_SevenSeg/sevenSeg.h"
8 #include "../HPS_Watchdog/HPS_Watchdog.h"
9 #include "../pongDisplay/pongDisplay.h"
10 #define PI 3.14159265359
11 #define centre_x 160
12 #define centre_y 120
13 #define UP 0
14 #define DOWN 1
15
16
17 // Publicly used
18 void pongEngine_init( void );
19 /*##### Ball Functions ####*/
20 void pongEngine_moveBall(int angle, int speed);
21 void pongEngine_createBall(void);
22 void pongEngine_destroyBall(void);
23 void pongEngine_setBallLocation(int x, int y);
24 int pongEngine_getBallAngle(void);
25 int pongEngine_getBallLocation_x(void);
26 int pongEngine_getBallLocation_y(void);
27 void pongEngine_resetBallLoc( void );
28 /*##### Paddle Functions ####*/
29 void pongEngine_paddleMove(int player, int direction, int speed);
30 void pongEngine_paddleSetYLimits(int maxy, int miny);
31 void pongEngine_paddleSetYLocation(int player, int y);
32 void pongEngine_paddleSetXLocation(int player, int x);
33 void pongEngine_paddleCreate(int player);
34 void pongEngine_paddleDestroy(int player);
35 unsigned int pongEngine_getPaddleY(unsigned int paddle);
36 /*##### Score Keeping ####*/
37 void pongEngine_addPoint(int player);
38 void pongEngine_resetScore( unsigned int _Refresh );
39
40 //used within library
41 /*##### Ball Functions ####*/
42 int pongEngine_calcAngle(int x1, int y1, int x2, int y2);
43 void pongEngine_calcBallPathInst(int angle);
44 void pongEngine_genBallPathInst(int x1, int y1, int x2, int y2);
45 /*##### Paddle Functions ####*/
46 /*##### Score Keeping ####*/
47 void pongEngine_refreshScore( void );
48 void pongEngine_resetPaddles( void );
49 #endif
```

8.9 Appendix I - pongSprites.c

```
1 #include "pongSprites.h"
2
3 volatile int sprite_Ball[17][17];
4
5 int paddlesize_x = 5;
6 int paddlesize_y = 60;
7
8 //This function initialises the ball
9 void pongSprites_initBall(void){
10    int x = 8;
11    int y = 8;
12    //Radius as signed int
13    int signedr = 8;
14    //Radius squared
15    int rad2 = signedr * signedr;
16    //Outline threshold
17    //Go through x's
18    int xc = 0;
19    int yc = 0;
20    //Loop through all X and Y of square the size of radius squared
21    for (xc = -signedr; xc <= signedr; xc++) {
22        for (yc = -signedr; yc <= signedr; yc++) {
23            //radius squared = yc^2 + xc^2
24            int pyr = (yc*yc) + (xc*xc);
25            //If no fill then draw outline
26            if(pyr <= rad2){
27                sprite_Ball[xc+x][yc+y] = 1;
28            }
29            else{
30                sprite_Ball[xc+x][yc+y] = 0;
31            }
32        }
33    }
34 }
35
36 //This function renders the ball
37 void pongSprites_renderBall(int x, int y, short colour){
38    int xl;
39    int yl;
40    //adjust so called at centre of ball
41    x = x-8;
42    y = y-8;
43    //loop through values and check if they equal radius
44    for(yl = 0; yl < 17; yl++){
45        for(xl = 0; xl < 17; xl++){
46            if(sprite_Ball[yl][xl] == 1){
47                Displays_setPixel(x+xl, y+yl, colour); //set pixels
48            }
49        }
50    }
51 }
52
53 //This function renders a paddle
54 void pongSprites_renderPaddle(int x, int y, short colour){
55    //taken from assignment 1 - graphics library and modified for paddle
56    int lly;
57    int llx;
58    //adjust for centre
59    x = x - (paddlesize_x/2);
60    y = y - (paddlesize_y/2);
61    //loop through values and set pixels
62    for(lly=0; lly <= paddlesize_y; lly++){
63        for(llx=0; llx <=paddlesize_x; llx++){
64            Displays_setPixel(x+llx ,y+lly ,colour); //set pixels
65        }
66    }
67 }
68
69 //This function renders a rectangle
70 void pongSprites_renderRectangle(int x1,int x2, int y1, int y2, int colour){
71    //taken from assignment 1 - graphics library and modified for paddle
72    int height = abs(y2-y1);
73    int width = abs(x1-x2);
74    int y=0;
75    int x=0;
76
77    //find bottom left value
78    int llx = 0;
79    int lly = 0;
80    if(x1<x2){
81        llx = x1;
82    }
83    else{
84        llx = x2;
85    }
86    if(y1<y2){
87        lly = y1;
88    }
89    else{
90        lly = y2;
91    }
92    //set pixels
93    for(y=0; y <= height; y++){
94        for(x=0; x<=width; x++) {
```

```

95     Displays_setPixel(x+llx ,y+lly ,colour);
96 }
97 }
98 }
99
100 //This function writes text to the screen
101 void pongSprites_writeText(int x, int y, int size, char *text, short colour){
102     int letter[5][7]; //letter 2 dimensional array
103     int stringLength = strlen(text); //string length
104     int character; //current character
105     int line; //current line
106     int pixel; //current pixel
107     int xp , yp; //counter for rendering of character
108 //for each character
109     for(character = 0; character < stringLength; character++){
110         int charX; //char value
111         if(size == 0){charX = x+((character)*6);} //if small
112         if(size == 1){charX = x+((character)*12);} //if large
113         //This code essentially turns the array 90 degrees and stores value in the letter 2-D array
114         //for each line
115         for(line = 0; line < 5; line++){
116             //for each pixel
117             for(pixel = 0; pixel < 7; pixel++){
118                 //check if pixel is on for the current character
119                 int pixelon = BF_fontMap[text[character] - ' '][line];
120                 int shiftedbit = 0;
121                 //shift down for each pixel to go down the line of pixels
122                 shiftedbit = (pixelon >> pixel) & 1;
123                 //output to letter array
124                 letter[line][pixel] = shiftedbit;
125             }
126         }
127         //render character
128         for(yp = 0; yp < 7; yp++){
129             for(xp = 0; xp < 5; xp++){
130                 //if pixel is on
131                 if(letter[xp][yp] == 1){
132                     if(size == 0){Displays_setPixel(charX+xp, y+yp, colour);} //if small set 1 pixel
133                     if(size == 1){ //if large set 4 pixels per pixel
134                         Displays_setPixel(charX+(xp*2), y+(yp*2), colour);
135                         Displays_setPixel(charX+(xp*2)+1, y+(yp*2), colour);
136                         Displays_setPixel(charX+(xp*2), y+(yp*2)+1, colour);
137                         Displays_setPixel(charX+(xp*2)+1, y+(yp*2)+1, colour);
138                     }
139                 }
140             }
141         }
142     }
143 }
144
145 //This function changes the paddles size
146 void pongSprites_changePaddleSize(unsigned int size){
147     paddlesize_y = 60+(size - 5)*10;
148 }
149
150 //This function gets the paddle size
151 unsigned int pongSprites_getPaddleSizeY( void ){
152     return paddlesize_y;
153 }
```

8.10 Appendix J - pongSprites.h

```
1 #ifndef PONGSPRITES_H
2 #define PONGSPRITES_H
3
4 #include "../pongDisplay/pongDisplay.h"
5 #include "pongEngine.h"
6 #include "../BasicFont/BasicFont.h"
7 #include <string.h>
8
9 #define SMALL 0
10 #define LARGE 1
11
12 void pongSprites_initBall( void );
13 void pongSprites_renderBall(int x, int y, short colour);
14 void pongSprites_renderPaddle(int x, int y, short colour);
15 void pongSprites_writeText(int x, int y, int size, char *text, short colour);
16 void pongSprites_renderRectangle(int x1,int x2, int y1, int y2, int colour);
17 void pongSprites_changePaddleSize(unsigned int size);
18 unsigned int pongSprites_getPaddleSizeY( void );
19
20 #endif
```

8.11 Appendix K - pongPhysics.c

```
1 /*  
2  * pongPhysics.c  
3  *  
4  * Created on: 24 Apr 2019  
5  * Author: John  
6  */  
7  
8 #include "pongPhysics.h"  
9  
10 int angle;  
11 int speed;  
12 int rounds; //Number of rounds elapsed  
13  
14 //  
15 // Code to regulate serving rules  
16 //  
17  
18 int pongPhysics_serve (void) {  
19  
20     int r = rand() % 1; //Choose random number between 0 and 9  
21     int serveSpeed = 2 ;  
22     int pr = 0; // Previous receiver (Player 1 or player 2)  
23  
24     if (rounds < 1) { //First serve  
25         if (r <= 4) { //Player 1 receives  
26             angle = 0;  
27             pr = 1;  
28             rounds++;  
29         } else if (r > 5) { //Player 2 receives  
30             angle = 180;  
31             pr = 2;  
32             rounds++;  
33         }  
34     } else { // Alternate thereafter  
35         if (pr == 1) {  
36             angle = 180;  
37             pr = 2 ;  
38             rounds++;  
39         } else if (pr == 2) {  
40             angle = 0 ;  
41             pr = 1 ;  
42             rounds++;  
43         }  
44     }  
45 }  
46  
47 return angle;  
48 }  
49  
50 //  
51 // Code to regulate horizontal border collisions  
52 //  
53  
54 int pongPhysics_borderCollision (int speed, int angle) {  
55  
56     int outangle;  
57  
58     outangle = -angle;  
59  
60     pongEngine_moveBall(outangle, speed); //Used to avoid ball getting stuck at the border  
61     pongEngine_moveBall(outangle, speed);  
62     pongEngine_moveBall(outangle, speed);  
63  
64     return outangle;  
65 }  
66  
67 // Code to regulate paddle collisions  
68  
69 int* pongPhysics_paddleCollision (int speed, int angle, int player) {  
70  
71     int outangle = 0;  
72     int outspeed = 0;  
73     int deltaAngle;  
74     float sector1 = (0.167 * pongSprites_getPaddleSizeY());  
75     float sector2 = (0.417 * pongSprites_getPaddleSizeY());  
76     float sector3 = (0.083 * pongSprites_getPaddleSizeY());  
77  
78     int output[2];  
79  
80     outangle = 180 - angle;  
81  
82     // Paddle sector 1 collision / no change in angle or speed  
83     if ((pongEngine_getBallLocation_y() < pongEngine_getPaddleY(player) + (int)(sector1)) && (pongEngine_getBallLocation_y() > pongEngine_getPaddleY(player) - (int)(sector1))) {  
84  
85         deltaAngle = 0;  
86         outspeed = 4;  
87  
88         // Paddle sector 2 collision  
89     } else if ((pongEngine_getBallLocation_y() >= pongEngine_getPaddleY (player) + (int)(sector1)) && (pongEngine_getBallLocation_y() <= pongEngine_getPaddleY (player) + (int)(sector2))) {  
90         deltaAngle = -10;  
91  
92 }
```

```

93     outspeed = 6;
94
95 } else if ((pongEngine_getBallLocation_y() < pongEngine_getPaddleY(player) - (int)(sector1)) && (pongEngine_getBallLocation_y() >=
96     pongEngine_getPaddleY(player) - (int)(sector2))) {
97
98     deltaAngle = 10;
99     outspeed = 6;
100
101 // Paddle sector 3 collision
102 } else if ((pongEngine_getBallLocation_y() > pongEngine_getPaddleY(player) + (int)(sector2)) && pongEngine_getBallLocation_y() <=
103     pongEngine_getPaddleY(player) + (int)(sector3))) {
104
105     deltaAngle = -20;
106     outspeed = 8;
107
108 } else if ((pongEngine_getBallLocation_y() < pongEngine_getPaddleY(player) - (int)(sector2)) && pongEngine_getBallLocation_y() >=
109     pongEngine_getPaddleY(player) - (int)(sector3))) {
110
111     deltaAngle = 20;
112     outspeed = 8;
113
114 } else {
115
116     outspeed = speed;
117
118     if ((pongEngine_getBallLocation_x() <= 50 + 15) && (pongEngine_getBallLocation_x() >= 50 + 10)) {
119
120         outangle = 135;
121
122     } else if ((pongEngine_getBallLocation_x() >= 270 - 15) && (pongEngine_getBallLocation_x() <= 270 - 10)) {
123
124         outangle = 45;
125
126     }
127
128 /*
129     if (outangle > 180) { //outangle < 360
130         outangle = -(360 - outangle);
131     } else if (outangle < -180) {
132         outangle = 360 + outangle;
133     } else if (outangle == 360) { // == 0 = 180 / ==180 = 0
134         outangle = 0;
135     }
136 */
137
138     outangle = outangle + deltaAngle;
139
140
141     if (outangle > 180) { //outangle < 360
142         outangle = -(360 - outangle);
143     } else if (outangle < -180) {
144         outangle = 360 + outangle;
145     } else if (outangle == 360) { // == 0 = 180 / ==180 = 0
146         outangle = 0;
147     }
148
149     output[0] = outangle;
150     output[1] = outspeed;
151
152
153
154     pongEngine_moveBall(output[0], output[1]);
155     pongEngine_moveBall(output[0], output[1]);
156     pongEngine_moveBall(output[0], output[1]);
157
158
159
160     return output;
161
162 }

```

8.12 Appendix L - pongPhysics.h

```
1  /*
2   * pongPhysics.h
3   *
4   * Created on: 24 Apr 2019
5   * Author: John
6   */
7
8 #ifndef PONGENGINE_PONGPHYSICS_H_
9 #define PONGENGINE_PONGPHYSICS_H_
10
11 #include <cmath>
12 #include <stdlib.h>
13 #include "pongSprites.h"
14 #include "pongEngine.h"
15 #include "../HPS_Watchdog/HPS_Watchdog.h"
16 #include "../pongDisplay/pongDisplay.h"
17
18
19 int pongPhysics_serve (void);
20 int pongPhysics_borderCollision (int speed, int angle);
21 int* pongPhysics_paddleCollision (int speed, int angle, int player);
22
23
24
25
26
27
28 #endif /* PONGENGINE_PONGPHYSICS_H_ */
```

8.13 Appendix M - pongInputs.c

```
1 #include "pongInputs.h"
2 // #include "../HPS IRQ/HPS IRQ.h"
3 // #include "../pongEngine/pongEngine.h"
4 // #include "../pongDisplay/pongDisplay.h"
5
6
7 /* Notes: Add audio output
8 * */
9
10 /* To set function running, simply include PS2_Keyboard.h and then call inputsInitialise */
11 /* Requires scatter file change to FPGARomRamIRQ.scat */
12
13
14 volatile unsigned int *PS2_DATA = (unsigned int *) 0xFF200100; // PS2 base address
15 volatile unsigned int *PS2_CONTROL = (unsigned int *) 0xFF200104; // PS2 control address
16 volatile unsigned int *KEY_ptr = (unsigned int *) 0xFF200050; // Keys
17
18 volatile unsigned int keyBuffer[3] = {0,0,0};
19
20 bool inputsIsInit = false;
21
22 volatile unsigned int mode;
23
24 // https://www.avrfreaks.net/sites/default/files/PS2%20Keyboard.pdf
25
26
27 /* IRQ handlers */
28 void keyboardISR(HPSIRQSource interruptID, bool isInit, void* initParams){
29     if (!isInit){
30         *PS2_CONTROL |= 0x0;
31         inputKeyboard();
32         // *PS2_DATA |= 0x00;
33         *PS2_CONTROL |= 0x1;
34         emptyFIFO();
35     }
36     HPS_ResetWatchdog();
37 }
38
39 void pushbuttonISR(HPSIRQSource interruptID, bool isInit, void* initParams) {
40     if (!isInit) {
41         unsigned int press;
42         // Read the Push-button interrupt register
43         press = KEY_ptr[3];
44         // Then clear the interrupt flag by writing the value back
45         KEY_ptr[3] = press;
46
47         if (press == 1){
48             Input(_UP, keySpeed);
49         }
50         else if (press == 2){
51             Input(_DOWN, keySpeed);
52         }
53         else if (press == 4){
54             Input(_W, keySpeed);
55         }
56         else if (press == 8){
57             Input(_S, keySpeed);
58         }
59         else if (press == 10){
60             Input(99, keySpeed);
61         }
62     }
63     // Reset watchdog.
64     HPS_ResetWatchdog();
65 }
66
67
68 /* Actual functions */
69
70 void inputsInitialise(void) {
71     if (!inputsIsInit){
72         // Register IRQs
73         HPS_IRQ_registerHandler(IRQ_LSC_PS2_PRIMARY, keyboardISR);
74         HPS_ResetWatchdog();
75         HPS_IRQ_registerHandler(IRQ_LSC_KEYS, pushbuttonISR);
76         HPS_ResetWatchdog();
77
78         // Flag keyboard as initialised
79         inputsIsInit = true;
80         HPS_ResetWatchdog();
81     }
82 }
83
84 char PS2Scan( void ) {
85     int keyboard_data, RVALID;
86     char inputKey;
87
88     keyboard_data = *(PS2_DATA); // Read Data register
89     RVALID = keyboard_data & 0x8000; // Extract RVALID
90     if (RVALID){
91         inputKey = keyboard_data & 0xFF; HPS_ResetWatchdog();
92     } else { inputKey = 0; } // Handle any odd errors
93     HPS_ResetWatchdog();
94     // printf("%X \n", inputKey); // Debug for checking scancodes
```

```

95     return inputKey;
96 }
97
98 void inputKeyboard( void ) {
99     char key = PS2Scan();
100    keyBuffer[0] = keyBuffer[1];
101    keyBuffer[1] = keyBuffer[2];
102    keyBuffer[2] = PS2Scan();
103
104    if (key == _W) {
105        keyBuffer[2] = 1; Input(_W, keyBSpeed);
106    } else if (key == _S) {
107        keyBuffer[2] = 2; Input(_S, keyBSpeed);
108    } else if (key == _UP) {
109        keyBuffer[2] = 3; Input(_UP, keyBSpeed);
110    } else if (key == _DOWN) {
111        keyBuffer[2] = 4; Input(_DOWN, keyBSpeed);
112    } else if (key == _ESC) {
113        keyBuffer[2] = 5; Input(_ESC, keyBSpeed);
114    } else if (key == _RETURN) {
115        keyBuffer[2] = 6; Input(_RETURN, keyBSpeed);
116    } else if ((key == _BKSPACE) && (keyBuffer[0] != 7)) {
117        keyBuffer[2] = 7; toggleSound(); // Input(7, keyBSpeed);
118    } else if (key == _LEFT) {
119        keyBuffer[2] = 8; Input(_LEFT, keyBSpeed);
120    } else if (key == _RIGHT) {
121        keyBuffer[2] = 9; Input(_RIGHT, keyBSpeed);
122    }
123
124 // Ignore make/break signals
125 if ((key == 0xF0) || (key == 0xE0)) keyBuffer[2] = 0;
126
127 // Avoid doubling motions on single press due to make/break
128 if ((keyBuffer[2] == keyBuffer[0]) && (keyBuffer[1] == 0)) keyBuffer[2] = 0;
129
130 HPS_ResetWatchdog();
131 }
132
133 void emptyFIFO( void ){
134     int RAVAIL = *(PS2_DATA) & 0xFFFF0000;
135     while (RAVAIL > 0x1000){
136         char temp;
137         temp = *(PS2_DATA) & 0xFF; // Read necessary to clear FIFO
138         RAVAIL = *(PS2_DATA) & 0xFFFF0000;
139         temp &= temp; // Get rid of annoying warning
140     }
141 }
142
143 void Input(unsigned int key, unsigned int speed){
144     if ((mode == GAME) || (mode == GAME_AI)){ // Common to games
145         if (key == _W){
146             pongEngine_paddleMove(1, DOWN, speed);
147             //pongEngine_destroyBall(); // Partially repairs artifacts - constrains them to path of ball
148         }
149         else if (key == _S){
150             pongEngine_paddleMove(1, UP, speed);
151             //pongEngine_destroyBall();
152         } else if (key == _ESC){ // Escape
153             setInputMode(MENUS);
154         }
155     }
156
157     if (mode == GAME){ // 2P only
158         if (key == _UP){
159             pongEngine_paddleMove(2, DOWN, speed);
160             //pongEngine_destroyBall();
161         }
162         else if (key == _DOWN){
163             pongEngine_paddleMove(2, UP, speed);
164             //pongEngine_destroyBall();
165         }
166     }
167
168 //menuMove(unsigned int direction)
169     if (mode == MENUS){
170         if (speed == keyBSpeed){
171             if (key == _UP){
172                 menuMove(_UP);
173             } else if (key == _DOWN){
174                 menuMove(_DOWN);
175             } else if (key == _LEFT){
176                 menuMove(_LEFT);
177             } else if (key == _RIGHT){
178                 menuMove(_RIGHT);
179             } else if (key == _RETURN){
180                 setMenu(4, 1); menuMove(0);
181             }
182         } else {
183             if (key == _UP){
184                 menuMove(_RIGHT);
185             } else if (key == _DOWN){
186                 menuMove(_LEFT);
187             } else if (key == _W){
188                 menuMove(_UP);
189             } else if (key == _S){
190                 menuMove(_DOWN);
191             }
192     }
193 }

```

```
192     }
193 }
194
195 if (key == _BKSPACE){ // Never reached
196     toggleSound();
197 }
198 ResetWDT();
200 }
201
202 void enableInputs(int enable){
203     if (enable == 1){
204         // Enable interrupts
205         KEY_ptr[2] = 0xF; // Set key interrupts
206         //PS2_CONTROL |= 0x1; // Set interrupt on PS2
207         *PS2_CONTROL = (1<<0);
208     } else {
209         // Disable interrupts
210         KEY_ptr[2] = 0x00; // Set key interrupts
211         //PS2_CONTROL &= 0x0; // Set interrupt on PS2
212         *PS2_CONTROL = (0<<0);
213     }
214 }
215
216 void setInputMode(unsigned int _mode){
217     //if (_mode == MENUS) { mode = MENUS; } else if (_mode == GAME){ mode = GAME; } else if (_mode == GAME_AI){ mode = GAME; }
218     mode = _mode;
219     if (mode == MENUS){ Displays_mode(NOFRAMEBUFFER); } else { Displays_mode(SOFTWAREOCTOFB); }
220 }
221
222 int getInputMode( void ){
223     return mode;
224 }
```

8.14 Appendix N - pongInputs.h

```
1 /* To set function running, simply include PS2_Keyboard.h and then call inputsInitialise */
2 /* Requires scatter file change to an IRQ ready file */
3 // http://www-ug.eecg.utoronto.ca/msl/nios_devices_SoC/datasheets/PS2%20Keyboard%20Protocol.htm
4
5 #ifndef PONGINPUTS_PONGINPUTS_H_
6 #define PONGINPUTS_PONGINPUTS_H_
7
8 #include <stdbool.h>
9 #include "../pongEngine/pongEngine.h"
10 #include "../pongDisplay/pongDisplay.h"
11 #include "../pongSound/pongSound.h"
12 #include "../pongScreens/pongScreens.h"
13 #include <string.h>
14 #include "../HPS_IRQ/HPS_IRQ.h"
15 #include "../HPS_Watchdog/HPS_Watchdog.h"
16 #include <stdio.h>
17 #include <stdlib.h>
18
19
20 // Keyboard scancodes for scan set 2
21 // From https://techdocs.altium.com/display/FPGA/PS2+Keyboard+Scan+Codes
22 #define _ESC 0x76
23 #define _RETURN 0x5A
24 #define _BSPACE 0x66
25 #define _W 0x1D
26 #define _A 0x1C
27 #define _S 0x1B
28 #define _D 0x23
29 #define _LEFT 0x6B
30 #define _RIGHT 0x74
31 #define _UP 0x75
32 #define _DOWN 0x72
33
34 // Definitions for modes
35 #define MENUS 0
36 #define GAME 1
37 #define GAME_AI 2
38
39 // Set paddle speeds
40 #define keySpeed 20
41 #define keyBSpeed 10
42
43
44 /* These are all pretty much used exclusively by this library. Interrupts handle everything and pass to InputControl. */
45
46 // Initialise the inputs
47 void inputsInitialise(void);
48
49 // Get raw keyboard scan codes, including make/break codes
50 char PS2Scan(void);
51
52 // Convert keyboard input to usable inputs and pass to Input()
53 void inputKeyboard( void );
54
55 // Clears FIFO on PS/2
56 void emptyFIFO( void );
57
58 // Takes input from keyboard/keys and acts on it depending on screen mode
59 void Input(unsigned int key, unsigned int speed);
60
61 // Set screen mode. Add switching between menus?
62 void setInputMode(unsigned int _mode);
63
64 // Get screen mode
65 int getInputMode( void );
66
67 // Enable/Disable inputs
68 void enableInputs(int enable);
69
70#endif /* PONGINPUTS_PONGINPUTS_H_ */
```

8.15 Appendix O - pongSounds.c

```
1 /*  
2  * pongSound.c  
3  *  
4  * Created on: 20 Apr 2019  
5  * Author: Sam  
6  */  
7  
8 #include "pongSound.h"  
9 #include "notes.h"  
10  
11 volatile unsigned int SoundOn = 1;  
12  
13 // Variables  
14 volatile int sound = 1;  
15 int freq = 1000;  
16 float duration = 1;  
17 volatile unsigned VOLUME = 4;  
18  
19 // Pointers  
20 volatile unsigned char* fifospace_ptr;  
21 volatile unsigned int* audio_left_ptr;  
22 volatile unsigned int* audio_right_ptr;  
23 volatile unsigned int* HPS_timer_ptr = (unsigned int *) 0xFFC08000;  
24  
25  
26 // Variables  
27 //Phase Accumulator  
28 double phase = 0.0; // Phase accumulator  
29 double inc = 0.0; // Phase increment  
30 double baseampl = 0.0; // Tone amplitude (i.e. volume)  
31 signed int audio_sample = 0;  
32  
33  
34 void timerISR(HPSIRQSource interruptID , bool isInit , void* initParams) {  
35     if (!isInit) {  
36         volatile unsigned int * HPS_timer_ptr = (unsigned int *) 0xFFC08000;  
37         //Clear the Timer Interrupt Flag  
38         //By reading timer end of interrupt register  
39         HPS_timer_ptr[3];  
40         // Turn sound off  
41         sound = 0;  
42     }  
43     //Reset watchdog.  
44     HPS_ResetWatchdog();  
45 }  
46  
47  
48  
49 void pongSound_Init(){  
50     //Initialise the Audio Codec.  
51     WM8731_initialise(0xFF203040);  
52 }  
53  
54 void Sound(int _freq, float _duration){  
55     //Duration in milliseconds  
56     if (SoundOn){  
57         freq = _freq; duration = _duration;  
58         sound = 1;  
59  
60         HPS_timer_ptr[2] = 0; // write to control register to stop timer  
61         // Set the timer period  
62         HPS_timer_ptr[0] = duration * 100000; // period = 1/(100 MHz) x (100 x 10^6) = 1 sec  
63         // Write to control register to start timer, with interrupts  
64         HPS_timer_ptr[2] = 0x03; // mode = 1, enable = 1  
65         // Register interrupt handler for timer  
66         HPS IRQ_registerHandler(IRQ_TIMER_L4SP_0, timerISR);  
67         HPS_ResetWatchdog();  
68  
69         //Clear both FIFOs  
70         WM8731_clearFIFO(true ,true );  
71         //Grab the FIFO Space and Audio Channel Pointers  
72         fifospace_ptr = WM8731_getFIFOSpacePtr();  
73         audio_left_ptr = WM8731_getLeftFIFOPtr();  
74         audio_right_ptr = WM8731_getRightFIFOPtr();  
75         //Initialise Phase Accumulator  
76         inc = ((float) freq) * 360.0 / F_SAMPLE; // Calculate the phase increment based on desired frequency - e.g. 440Hz  
77         baseampl = 838608.0; // Pick desired amplitude (e.g. 2^23). WARNING: If too high = deafening!  
78         phase = 0.0;  
79         // Primary function while loop  
80         while(sound == 1){  
81             //Always check the FIFO Space before writing or reading left/right channel pointers  
82             if ((fifospace_ptr[2] > 0) && (fifospace_ptr[3] > 0)) {  
83                 //If there is space in the write FIFO for both channels:  
84                 //Increment the phase  
85                 phase = phase + inc;  
86  
87                 //Ensure phase is wrapped to range 0 to 360* (range of lookupSin function)  
88                 while (phase >= 360.0){  
89                     phase = phase - 360.0;  
90                 }  
91                 // Calculate next sample of the output tone.  
92                 audio_sample = (signed int)( (1<<VOLUME)*baseampl * lookupSin((int) phase) );  
93             }  
94         }
```

```

95     // Output tone to left and right channels.
96     *audio_left_ptr = audio_sample;
97     *audio_right_ptr = audio_sample;
98   }
99   ResetWDT();
100 }
101 // Disable timer ISR
102 HPS_timer_ptr[2] = 0x02; // mode = 1, enable = 0
103 // Finally reset the watchdog.
104 HPS_ResetWatchdog();
105 }
106 }
107 }
108
109 void enableSound(unsigned int _onoff){
110   SoundOn = _onoff;
111 }
112
113 void toggleSound(){
114   if (SoundOn == 1) { SoundOn = 0; } else SoundOn = 1;
115 }
116
117 void setVolume(unsigned int _volume){
118   VOLUME = _volume;
119 }
120
121 // Set sounds here
122 void startSound(){
123   unsigned int i;
124   unsigned int sequence[] = {C4, D4, E4, F4, G4, 0, G4};
125   unsigned int durations[] = {200, 200, 200, 200, 500, 200, 500};
126   for (i=0; i<=6; i++){
127     Sound(sequence[i], durations[i]);
128     ResetWDT();
129   }
130 }
131
132 void paddleBeep(){
133   Sound(B4, 100);
134 }
135
136 void ballOutBeep(){
137   Sound(B3, 100);
138 }
139
140 float lookupSin(unsigned int degree){
141   // Angles found with matlab, x10^5 and rounded
142   static unsigned int lookup[91] = { 0, 1745, 3490, 5234, 6976, 8716, 10453, 12187, 13917,
143     15643, 17365, 19081, 20791, 22495, 24192, 25882, 27564,
144     29237, 30902, 32557, 34202, 35837, 37461, 39073, 40674,
145     42262, 43837, 45399, 46947, 48481, 50000, 51504, 52992,
146     54464, 55919, 57358, 58779, 60182, 61566, 62932, 64279,
147     65606, 66913, 68200, 69466, 70711, 71934, 73135, 74314,
148     75471, 76604, 77715, 78801, 79864, 80902, 81915, 82904,
149     83867, 84805, 85717, 86603, 87462, 88295, 89101, 89879,
150     90631, 91355, 92050, 92718, 93358, 93969, 94552, 95106,
151     95630, 96126, 96593, 97030, 97437, 97815, 98163, 98481,
152     98769, 99027, 99255, 99452, 99619, 99756, 99863, 99939,
153     99985, 100000 };
154
155   if (degree <= 90){
156     return (float) lookup[degree]/100000;
157   }
158   else if ((degree > 90) && (degree <= 180)){
159     return (float) lookup[180-degree]/100000;
160   }
161   else if ((degree > 180) && (degree <= 270)){
162     return (float) -lookup[degree-180]/100000;
163   }
164   else if ((degree > 270) && (degree <= 360)){
165     return (float) -lookup[360-degree]/100000;
166   }
167   else return 0;
}

```

8.16 Appendix P - pongSounds.h

```
1  /*
2   * pongSound.h
3   *
4   * Created on: 20 Apr 2019
5   * Author: Sam
6   */
7
8  /* Warning – sounds are BLOCKING      */
9
10 #include "../DE1SoC_WM8731/DE1SoC_WM8731.h"
11 #include "../HPS_Watchdog/HPS_Watchdog.h"
12 #include "../HPS_IRQ/HPS_IRQ.h"
13
14 #ifndef _PONGSOUND_PONGSOUND_H
15 #define _PONGSOUND_PONGSOUND_H
16
17 // Define some useful constants
18 #define F_SAMPLE 48000.0           // Sampling rate of WM8731 Codec (Do not change)
19
20 // Initialise driver
21 void pongSound_Init( void );
22
23 // Play a sound of frequency _freq Hz for _duration milliseconds
24 void Sound(int _freq, float _duration);
25
26 // Enable (1) or disable (0) sounds
27 void enableSound(unsigned int _onoff);
28
29 // Set volume, 1–9
30 void setVolume(unsigned int _volume);
31
32 // Toggle sound
33 void toggleSound( void );
34
35 // Fast sin implementation using lookup table(0–90°), characteristics of sin and fixed point(ish) storage
36 float lookupSin(unsigned int degree);
37
38 // Should probably add volume control too
39
40
41 // Custom sounds here
42 void startSound( void );
43
44 void paddleBeep( void );
45
46 void ballOutBeep( void );
47
48
49 #endif /* PONGSOUND_PONGSOUND_H */
```

8.17 Appendix Q - pongScreens.c

```
1 /*  
2  * pongScreens.c  
3  *  
4  * Created on: 20 Apr 2019  
5  * Author: Sam  
6  */  
7  
8 #include "pongScreens.h"  
9 // #include <string.h>  
10  
11 volatile unsigned int menuSelector = 0;  
12 volatile unsigned int settings[] = {0,5,4,0,0};  
13 volatile unsigned int menuSelectorOld = 0;  
14 volatile unsigned int settingsOld[] = {0,5,4,0,0};  
15 short menuColours[] = {_BLUE, _BLACK, _BLACK, _BLACK, _BLACK};  
16  
17 unsigned int settingsMax[] = {1, 9, 9, 1, 1};  
18  
19 unsigned int numMenuItems = sizeof(settings)/sizeof(int);  
20 volatile unsigned int gameMode = GAME_AI;  
21 volatile unsigned int gameModeOld = GAME_AI;  
22 int n = 0;  
23 volatile unsigned int *slider_ptr = (unsigned int *)0xFF200040;  
24 unsigned int last_slider;  
25  
26 void setMenu(unsigned int _menuSelector, unsigned int _setting){  
27     menuSelectorOld = menuSelector;  
28     settingsOld[_menuSelector] = settings[_menuSelector];  
29  
30     menuSelector = _menuSelector;  
31     settings[_menuSelector] = _setting;  
32 }  
33  
34 void menuMove(unsigned int direction){  
35     unsigned int i;  
36     char str_txt[30];  
37     enableInputs(0);  
38  
39     // Save old settings and reset menu colours  
40     for (i = 0; i<numMenuItems; i++){  
41         settingsOld[i] = settings[i];  
42         menuColours[i] = _BLACK;  
43     }  
44  
45     menuSelectorOld = menuSelector;  
46  
47     if (direction == _DOWN){  
48         if (menuSelector == 4){ menuSelector = 0; } else menuSelector++;  
49         //menuSelector++;  
50     } else if (direction == _UP){  
51         if (menuSelector == 0){ menuSelector = 4; } else menuSelector--;  
52         //menuSelector--;  
53     } else if (direction == _LEFT){  
54         if (settings[menuSelector] == 0) { settings[menuSelector] = settingsMax[menuSelector]; } else settings[menuSelector]--;  
55         //settings[menuSelector]--;  
56     } else if (direction == _RIGHT){  
57         if (settings[menuSelector] == settingsMax[menuSelector]) { settings[menuSelector] = 0; } else settings[menuSelector]++;  
58         //settings[menuSelector]++;  
59     }  
60     //printf("%d %d\n", menuSelector, settings[menuSelector]); // For debug only  
61  
62  
63     /* Set menu item colours */  
64     if (menuSelector != menuSelectorOld){  
65         menuColours[menuSelector] = _BLUE;  
66         pongSprites_renderBall(50, (71+(int)25+menuSelectorOld), _WHITE); ResetWDT(); // Get rid of old ball  
67         pongSprites_renderBall(50, (71+(int)25+menuSelector), _BLACK); ResetWDT(); // Replace  
68         // Menu items  
69         pongSprites_writeText(100, 20, LARGE, "Main menu", _RED); ResetWDT();  
70         pongSprites_writeText(75, 65, LARGE, "Mode: ", menuColours[0]); ResetWDT();  
71         pongSprites_writeText(75, 90, LARGE, "Paddle size: ", menuColours[1]); ResetWDT();  
72         pongSprites_writeText(75, 115, LARGE, "Volume: ", menuColours[2]); ResetWDT();  
73         pongSprites_writeText(75, 140, LARGE, "Reset score > ", menuColours[3]); ResetWDT();  
74         pongSprites_writeText(75, 165, LARGE, "START >", menuColours[4]); ResetWDT();  
75     }  
76  
77     /* Menu effects */  
78     if (menuSelector == 0){ // Game mode  
79         if (settings[0] == 1){ // If press right on Reset  
80             pongSprites_writeText(240, 65, 1, "<AI>", _WHITE); ResetWDT();  
81             pongSprites_writeText(240, 65, 1, "<2P>", _MAGENTA); ResetWDT();  
82             gameMode = GAME;  
83         } else {  
84             pongSprites_writeText(240, 65, 1, "<2P>", _WHITE); ResetWDT();  
85             pongSprites_writeText(240, 65, 1, "<AI>", _MAGENTA); ResetWDT();  
86             gameMode = GAME_AI;  
87         }  
88     }  
89  
90     if ((menuSelector == 1) && (settings[menuSelector] != settingsOld[menuSelector])){ // Paddle size  
91         sprintf(str_txt, "%d", settingsOld[1]);  
92         pongSprites_writeText(242, 90, LARGE, str_txt, _WHITE); ResetWDT();  
93         sprintf(str_txt, "%d", settings[1]);  
94         pongSprites_writeText(242, 90, LARGE, str_txt, _MAGENTA); ResetWDT();
```

```

95     pongSprites_changePaddleSize(settings[1]);
96 }
97
98 if ((menuSelector == 2) && (settings[menuSelector] != settingsOld[menuSelector])){ // Volume
99     sprintf(str_txt, "%d", settingsOld[2]);
100    pongSprites_writeText(242, 115, LARGE, str_txt, _WHITE); ResetWDT();
101    sprintf(str_txt, "%d", settings[2]);
102    pongSprites_writeText(242, 115, LARGE, str_txt, _MAGENTA); ResetWDT();
103
104    if (settings[2] != 0){
105        enableSound(1);
106        setVolume(settings[2]);
107    } else {
108        enableSound(0); // Disable sound at volume 0
109    }
110
111 //Sound(G4, 50); ResetWDT();
112 //paddleBeep();
113 }
114
115 if (menuSelector == 3){
116     if (settings[3] == 1){ // If press right on Reset
117         settings[3] = 0; // Reset setting
118         pongEngine_resetScore(0); // Reset scores
119     }
120 }
121
122 if (menuSelector == 4){ // Start game
123     if (settings[4] == 1){ // If press right on start
124         settings[4] = 0; // Reset start setting
125         if (gameModeOld != gameMode){ // Reset score for new game modes
126             pongEngine_resetScore(0);
127         }
128         // Ensure last ball is gone for animation
129         pongSprites_renderRectangle(50-8, 50+8, 71-8, 180, _WHITE);
130         for (i = 50; i < 260; i++){
131             pongSprites_renderBall(i, (71+(int)25*menuSelector), _WHITE); ResetWDT(); // Get rid of old ball
132             pongSprites_renderBall(i+1, (71+(int)25*menuSelector), _BLACK); ResetWDT(); // Replace
133         }
134         setInputMode(gameMode); // Begin playing
135     }
136 }
137
138 usleep(150000); //paddleBeep(); // Prevent keyboard input bounce
139 emptyFIFO();
140 ResetWDT();
141 enableInputs(1);
142 }
143
144 void startScreen(){
145     // Clear screen and set input mode
146     Displays_fillColour(_RED); // Fill background
147     setInputMode(MENUS); // Ensure input mode set to menus
148
149     // Write greeting and refresh
150     pongSprites_writeText(60, 120, 1, "Welcome to armPONG", 0xFFFF);
151     Displays_forceRefresh();
152     ResetWDT();
153
154     startSound(); // Play startup sound
155     usleep(1000000); ResetWDT(); // Sleep for 2s
156     usleep(1000000); ResetWDT();
157     last_slider = *slider_ptr; // Ensure slider positions are saved
158 }
159
160 void gameMenu(){
161     unsigned int i;
162     char str_txt[30];
163
164     setInputMode(MENUS);
165     gameModeOld = gameMode;
166     SDisplay_clearAll();
167
168     // Reset menu
169     menuSelector = 0; menuSelectorOld = 0;
170     for (i = 0; i < numMenuItems; i++){
171         menuColours[i] = _BLACK;
172     }
173     menuColours[0] = _BLUE;
174
175     // Displays_clearScreen();
176     Displays_fillColour(_WHITE);
177
178     // Displays_forceRefresh();
179     // Menu items
180     pongSprites_renderBall(50, 71/*+25*menuSelector*/ , _BLACK); ResetWDT();
181     pongSprites_writeText(100, 20, LARGE, "Main menu", _RED); ResetWDT();
182     pongSprites_writeText(75, 65, LARGE, "Mode: ", menuColours[0]); ResetWDT();
183     pongSprites_writeText(75, 90, LARGE, "Paddle size: ", menuColours[1]); ResetWDT();
184     pongSprites_writeText(75, 115, LARGE, "Volume: ", menuColours[2]); ResetWDT();
185     pongSprites_writeText(75, 140, LARGE, "Reset score > ", menuColours[3]); ResetWDT();
186     pongSprites_writeText(75, 165, LARGE, "START >", menuColours[4]); ResetWDT();
187
188     // Options (Defaults)
189     if (settings[0] == 0){
190         pongSprites_writeText(240, 65, LARGE, "<AI>", _MAGENTA); ResetWDT();
191     }

```

```

192 } else {
193     pongSprites_writeText(240, 65, LARGE, "<2P>", _MAGENTA); ResetWDT();
194 }
195
196 sprintf(str_txt, "<%d>", settings[1]); // Paddle size options
197 pongSprites_writeText(242, 90, LARGE, str_txt, _MAGENTA); ResetWDT();
198
199 sprintf(str_txt, "<%d>", settings[2]); // Volume options
200 pongSprites_writeText(242, 115, LARGE, str_txt, _MAGENTA); ResetWDT();
201
202 ResetWDT();
203 // Displays_forceRefresh(); // Hmm that frame buffer's being odd again
204 Displays_Refresh();
205 enableInputs(1);
206 while (getInputMode() == MENUS){
207     /* Most of this can probably be pushed to menuMove */
208
209     if ((menuSelector != menuSelectorOld) || (settings[menuSelector] != settingsOld[menuSelector])){
210         // As sounds are blocking, and rely on a timer ISR, they need to be called outside menuMove,
211         // as it is essentially an extension of the input ISRs
212         if (settings[2] != settingsOld[2]){
213             Sound(G4, 50);
214         }
215
216         // Displays_forceRefresh();
217
218         menuSelectorOld = menuSelector;
219         for (i = 0; i<numMenuItems; i++){
220             settingsOld[i] = settings[i];
221             menuColours[i] = _BLACK;
222         }
223     }
224     ResetWDT();
225 }
226 last_slider = *slider_ptr;
227 Displays_clearScreen();
228 }
229
230 void testScreen_AI( void ){
231     int dir;
232     int vel;
233     int *arr;
234     int x;
235     int variable = 240;
236
237     // Clear screen and set input mode
238     Displays_clearScreen();
239     setInputMode(GAME_AI);
240
241     // Initialise engine
242     pongEngine_init();
243     ResetWDT();
244
245     pongEngine_createBall();
246     // pongSprites_writeText(96, 60, 1, "AI MODE", 0xFFFF);
247
248     Displays_forceRefresh(); pongEngine_refreshScore();
249     n = 0;
250     while (getInputMode() == GAME_AI){
251
252         ResetWDT();
253         Displays_forceRefresh();
254
255         if (n < 1) {
256
257             n++;
258             dir = pongPhysics_serve();
259             vel = 2;
260         }
261
262         if ((pongEngine_getBallLocation_x() <= 50 + 15) && (pongEngine_getBallLocation_x() >= 50 + 10)) {
263
264             if ((pongEngine_getBallLocation_y() <= pongEngine_getPaddleY(1) + 30 + 15) && (pongEngine_getBallLocation_y() >= pongEngine_getPaddleY(1) - 30 - 15)) {
265
266                 paddleBeep();
267                 arr = pongPhysics_paddleCollision(vel, dir, 1);
268                 dir = arr[0];
269                 vel = arr[1];
270             }
271
272         } else if ((pongEngine_getBallLocation_x() >= 270 - 15) && (pongEngine_getBallLocation_x() <= 270 - 10)) { // -10
273
274             if ((pongEngine_getBallLocation_y() <= pongEngine_getPaddleY(2) + 30 + 15) && (pongEngine_getBallLocation_y() >= pongEngine_getPaddleY(2) - 30 - 15)) {
275
276                 paddleBeep();
277                 arr = pongPhysics_paddleCollision(vel, dir, 2);
278                 dir = arr[0];
279                 vel = arr[1];
280
281             }
282         }
283     }
284
285     // collisions
286 }
```

```

287 if ( pongEngine_getBallLocation_y() <= 23 + 20 ) {
288     enableInputs(0);
289     paddleBeep();
290     enableInputs(1);
291
292     dir = pongPhysics_borderCollision(vel, dir);
293
294 }
295 } else if ( pongEngine_getBallLocation_y() >= 230 - 10){
296     enableInputs(0);
297     ballOutBeep();
298     enableInputs(1);
299
300     dir = pongPhysics_borderCollision(vel, dir);
301 }
302
303
304
305 // AI bit
306 if ( pongEngine_getBallLocation_y() > pongEngine_getPaddleY(2)){
307
308     pongEngine_paddleMove(2, UP, 2);
309
310 } else if ( pongEngine_getBallLocation_y() < pongEngine_getPaddleY(2)){
311
312     pongEngine_paddleMove(2, DOWN, 2);
313 }
314
315 pongEngine_moveBall(dir, vel);
316
317 if (*slider_ptr != last_slider) {
318
319     setInputMode(MENUS);
320     last_slider = *slider_ptr;
321 }
322 //This function adds points
323 if ( pongEngine_getBallLocation_x() <= 10 ) { //+20
324     pongEngine_destroyBall(); //destroy ball
325     Displays_forceRefresh(); //force refresh for buffers prior to points
326     enableInputs(0); //disable inputs
327     paddleBeep(); //play sound
328     enableInputs(1); //enable inputs
329     pongEngine_resetPaddles(); //reset paddles
330     pongEngine_resetBallLoc(); //reset ball location
331     pongEngine_createBall(); //create ball
332     pongEngine_paddleCreate(1); //create paddle
333     pongEngine_paddleCreate(2); //create paddle
334     pongEngine_moveBall(1, 0); //reset angle instructions
335     pongEngine_moveBall(0, 0);
336     Displays_forceRefresh(); pongEngine_refreshScore(); //force refresh
337     variable = 240; //reset var
338     n = 0; //reset var
339     pongEngine_addPoint(2); //add point
340 } else if ( pongEngine_getBallLocation_x() >= 320 - 10){
341     pongEngine_destroyBall(); //destroy ball
342     enableInputs(0); //disable inputs
343     paddleBeep(); //play sound
344     enableInputs(1); //enable inputs
345     pongEngine_resetPaddles(); //reset paddles
346     pongEngine_resetBallLoc(); //reset ball location
347     pongEngine_createBall(); //create ball
348     pongEngine_paddleCreate(1); //create paddle
349     pongEngine_paddleCreate(2); //create paddle
350     pongEngine_moveBall(1, 0); //reset angle instructions
351     pongEngine_moveBall(0, 0);
352     Displays_forceRefresh(); pongEngine_refreshScore(); //force refresh
353     variable = 240; //reset var
354     n = 0; //reset var
355     pongEngine_addPoint(1); //add point
356
357 }
358 }
359
360 //gameEngine_paddleDestroy(1);
361 //gameEngine_paddleDestroy(2);
362 Displays_clearScreen();
363 }
364
365
366 void testScreen( void ){
367     int dir = 0;
368     //int x;
369     //int variable = 240; //240
370
371     int vel;
372     int* arr;
373     int x;
374     int variable = 240;
375     // Clear screen and set input mode
376     Displays_clearScreen();
377     setInputMode(GAME);
378
379     // Initialise engine
380     pongEngine_init();
381     ResetWDT();
382
383     pongEngine_createBall();

```

```

384 //pongSprites_writeText(96, 60, 1, "2P MODE", 0xFFFF);
385 Displays_forceRefresh(); pongEngine_refreshScore();
386 n = 0;
387 while (getInputMode() == GAME){
388     ResetWDT();
389     Displays_forceRefresh();
390
391     if (n < 1) {
392         n++;
393         dir = pongPhysics_serve ();
394         vel = 3;
395     }
396
397     if ((pongEngine_getBallLocation_x() <= 50 + 15) && (pongEngine_getBallLocation_x() >= 50 + 10)) {
398
399     if ((pongEngine_getBallLocation_y() <= pongEngine_getPaddleY (1) + 30) && (pongEngine_getBallLocation_y() >=
400 pongEngine_getPaddleY (1) - 30)) {
401
402         paddleBeep();
403         arr = pongPhysics_paddleCollision(vel ,dir ,1);
404         dir = arr[0];
405         vel = arr[1];
406
407     } else if ((pongEngine_getBallLocation_x() >= 270 - 15) && (pongEngine_getBallLocation_x() <= 270 - 10)) { // -10
408
409     if ((pongEngine_getBallLocation_y() <= pongEngine_getPaddleY (2) + 30) && (pongEngine_getBallLocation_y() >=
410 pongEngine_getPaddleY (2) - 30)) {
411
412         paddleBeep();
413         arr = pongPhysics_paddleCollision(vel ,dir ,2);
414         dir = arr[0];
415         vel = arr[1];
416
417     }
418 }
419
420 // collisions
421 if ( pongEngine_getBallLocation_y() <= 23 + 20 ) {
422     enableInputs(0);
423     paddleBeep();
424     enableInputs(1);
425
426     dir = pongPhysics_borderCollision(vel , dir);
427
428 } else if ( pongEngine_getBallLocation_y() >= 230 - 10){
429     enableInputs(0);
430     ballOutBeep();
431     enableInputs(1);
432
433     dir = pongPhysics_borderCollision(vel , dir);
434 }
435
436 // This function adds points
437 if ( pongEngine_getBallLocation_x() <= 10 ) { //+20
438     pongEngine_destroyBall(); //destroy ball
439     Displays_forceRefresh(); //force refresh for buffers prior to points
440     enableInputs(0); //disable inputs
441     paddleBeep(); //play sound
442     enableInputs(1); //enable inputs
443     pongEngine_resetPaddles(); //reset paddles
444     pongEngine_resetBallLoc(); //reset ball location
445     pongEngine_createBall(); //create ball
446     pongEngine_paddleCreate(1); //create paddle
447     pongEngine_paddleCreate(2); //create paddle
448     pongEngine_moveBall(1, 0); //reset angle instructions
449     pongEngine_moveBall(0, 0);
450     Displays_forceRefresh(); pongEngine_refreshScore(); //force refresh
451     variable = 240; //reset var
452     n = 0; //reset var
453     pongEngine_addPoint(2); //add point
454 } else if ( pongEngine_getBallLocation_x() >= 320 - 10){
455     pongEngine_destroyBall(); //destroy ball
456     enableInputs(0); //disable inputs
457     paddleBeep(); //play sound
458     enableInputs(1); //enable inputs
459     pongEngine_resetPaddles(); //reset paddles
460     pongEngine_resetBallLoc(); //reset ball location
461     pongEngine_createBall(); //create ball
462     pongEngine_paddleCreate(1); //create paddle
463     pongEngine_paddleCreate(2); //create paddle
464     pongEngine_moveBall(1, 0); //reset angle instructions
465     pongEngine_moveBall(0, 0);
466     Displays_forceRefresh(); pongEngine_refreshScore(); //force refresh
467     variable = 240; //reset var
468     n = 0; //reset var
469     pongEngine_addPoint(1); //add point
470 }
471
472 pongEngine_moveBall(dir , vel);
473
474 if (*slider_ptr != last_slider){
475     setInputMode(MENUS);
476     last_slider = *slider_ptr;
477 }

```

```
479 }  
480 //gameEngine_paddleDestroy(1);  
481 //gameEngine_paddleDestroy(2);  
482 Displays_clearScreen();  
483 }
```

8.18 Appendix R - pongScreens.h

```
1  /*
2   * pongScreens.h
3   *
4   * Created on: 20 Apr 2019
5   * Author: Sam
6   */
7
8 #ifndef PONGSCREENS_H_
9 #define PONGSCREENS_H_
10
11 #include "../pongInputs/pongInputs.h"
12 #include "../pongSound/pongSound.h"
13 #include "../pongSound/notes.h"
14 #include "../pongEngine/pongSprites.h"
15 #include "../pongDisplay/pongDisplay.h"
16 #include "../DE1SoC_SevenSeg/sevenSeg.h"
17 #include "../HPS_Watchdog/HPS_Watchdog.h"
18 #include "../HPS_usleep/HPS_usleep.h"
19 #include "../pongEngine/pongPhysics.h"
20 #include <stdio.h>
21 #include <stdbool.h>
22
23 #define _BLACK (0x0000)
24 #define _WHITE (0xFFFF)
25 #define _RED (0xF800)
26 #define _GREEN (0x07C0)
27 #define _BLUE (0x001F)
28 #define _YELLOW (_RED | _GREEN)
29 #define _CYAN (_GREEN | _BLUE)
30 #define _MAGENTA (_BLUE | _RED)
31
32 void menuMove(unsigned int direction);
33
34 void setMenu(unsigned int _menuSelector, unsigned int _setting);
35
36 void startScreen( void );
37
38 void testScreen( void );
39
40 void testScreen_AI( void );
41
42 void gameMenu( void );
43
44#endif /* PONGSCREENS_H_ */
```

8.19 Appendix S - main.c

```
1 #include "DE1Soc_VGA/DE1SoC_VGA.h"
2 #include "pongDisplay/pongDisplay.h"
3 #include "pongEngine/pongSprites.h"
4 #include "pongEngine/pongEngine.h"
5 #include "pongScreens/pongScreens.h"
6 #include "pongInputs/pongInputs.h"
7 #include "pongSound/pongSound.h"
8 #include "HPS_Watchdog/HPS_Watchdog.h"
9
10
11 int main(void) {
12     int FS = 3;
13     // int paddleX;
14
15     // Initialise displays
16     Displays_init(0xC8000000,0xC9000000,0xFF200060,0xFF200080);
17     Displays_frameSkip(FS);
18     ResetWDT();
19
20     // Initialise interrupts
21     HPS_IRQ_initialise(NULL);
22     HPS_ResetWatchdog();
23
24     // Initialise keyboard/pushbuttons
25     inputsInitialise();
26
27     // Initialise sounds
28     pongSound_Init();
29
30     // Init ball for menu
31     pongSprites_initBall();
32
33     // Run loading screen
34     startScreen();
35
36     // Turn on inputs
37     enableInputs(1);
38     while(1){
39         if (getInputMode() == MENUS){
40             // Run start menu
41             gameMenu();
42         } else if (getInputMode() == GAME){
43             // Run 2P test screen
44             testScreen();
45         } else if (getInputMode() == GAME_AI){
46             // Run test screen
47             testScreen_AI();
48         }
49     }
50 }
```

References

- [1] Altera, *DE1-SoC Computer System with Nios II*. Altera, 2014.
- [2] Altium, “Ps2 keyboard scan codes,” [Online: Accessed 29-Apr-2019]. [Online]. Available: <https://techdocs.altium.com/display/FPGA/PS2+Keyboard+Scan+Codes>