UNIVERSITY OF LEEDS

ELEC5620M
Mini Project

DE1-SoC Pong

Alexander Bolton - 200938078
Sam Wilcock - 201285260
John Jakobsen - 201291841

May 2019

The University of Leeds
School of Electronic and Electrical Engineering

Contents

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 is a brief description of the game AI, and section 7 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]

$$VGA_{baseaddress} + (pixelX_{coordinates}\, pixelY_{coordinates}\, 0_2) \tag{1}$$

$$C8000000_{16} + (00000001\,000000000\,0)_2 = C8000400_{16} \tag{2}$$

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.

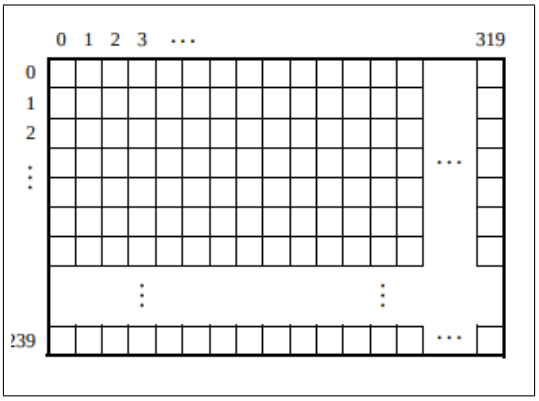


Figure 1: Pixel layout for pixel buffer of VGA controller [1]

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.

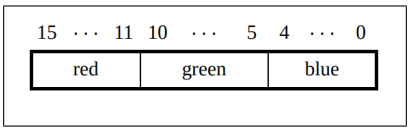


Figure 2: Pixel Colour Layout [1]

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

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

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  int q1change = memcmp(frontFrameBuffer1,rearFrameBuffer1, sizeof(frontFrameBuffer1));
2  int q2change = memcmp(frontFrameBuffer2,rearFrameBuffer2, sizeof(frontFrameBuffer2));
3  int q3change = memcmp(frontFrameBuffer3,rearFrameBuffer3, sizeof(frontFrameBuffer3));
4  int q4change = memcmp(frontFrameBuffer4,rearFrameBuffer4, sizeof(frontFrameBuffer4));
5  //if any changes whilst running through loop then update frame buffer and set on hardware.
6  for(y = 0; y < PixelHeight/2; y++){
7      for(x = 0; x < PixelWidth/2; x++){
8        if(q1change != 0){
9          if(frontFrameBuffer1[x][y] != rearFrameBuffer1[x][y]){
10           Displays_drawPixel(x,y,frontFrameBuffer1[x][y]); //set on hardware
11           rearFrameBuffer1[x][y] = frontFrameBuffer1[x][y]; //update FB
```

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

A number of function 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.

$$Radius^2 = x^2 + y^2 \tag{3}$$

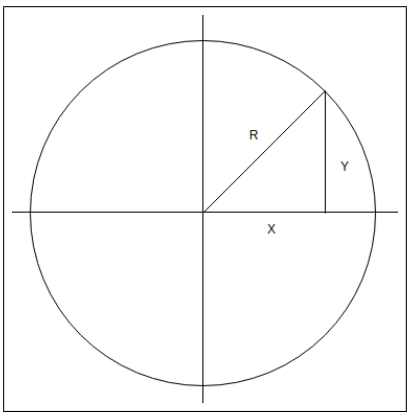


Figure 5: Pythagoras Theorem demonstrated inside circle

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.

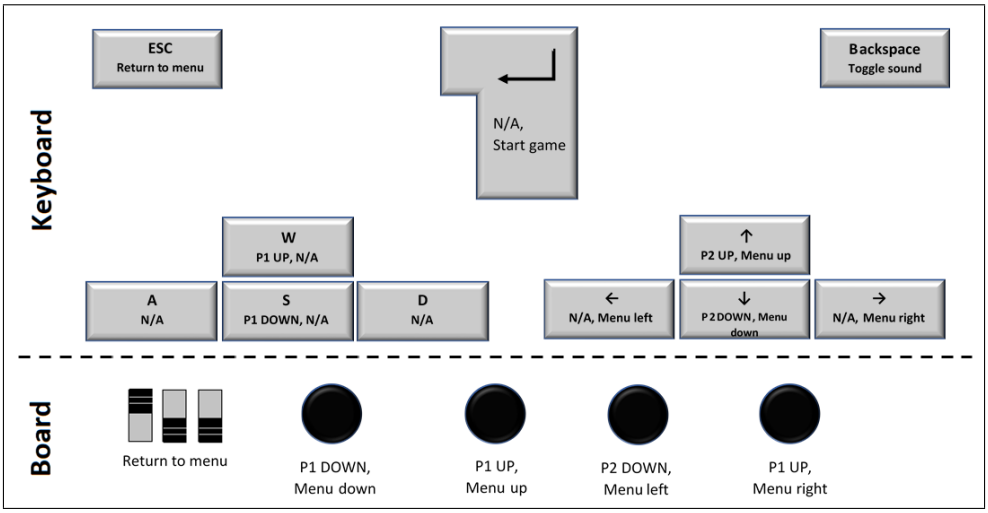


Figure 7: Input functionality

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 MENUS (a useful dummy definition in the header file) and the game exits to the main() while loop:

```
1
2      paddleBeep();
3      arr = pongPhysics_paddleCollision(vel,dir,2);
4      dir = arr[0];
```

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 preceeded 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 pusbhutton 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 macross 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.

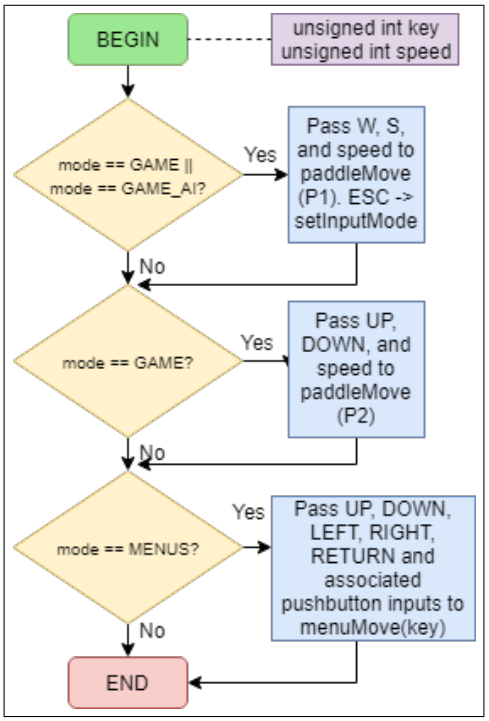


Figure 10: The Input() macro flow

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.

```
static unsigned int lookup[91] = { 0, 1745, 3490, 5234, 6976, 8716, 10453, 12187, 13917,
           15643, 17365, 19081, 20791, 22495, 24192, 25882, 27564,
           ... etc. ...
  if (degree <= 90){
    return (float) lookup[degree]/100000;
  }
  else if ((degree > 90) && (degree <= 180)){
    return (float) lookup[180-degree]/100000;
  }
  else if ((degree > 180) && (degree <= 270)){
    return (float) -lookup[degree-180]/100000;
  }
  else if ((degree > 270) && (degree <= 360)){
    return (float) -lookup[360-degree]/100000;
  } else return 0;
}
```

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 $x = 90°$, and the characteristic that the function is mirrored about $y = 0$ at $x = 180°$, 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.

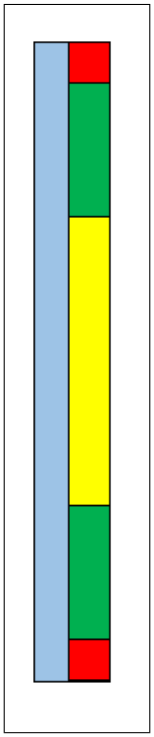


Figure 15: showing location and relative size of each sector on the paddle.

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.

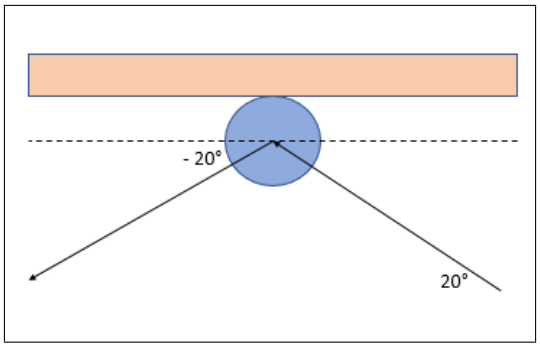
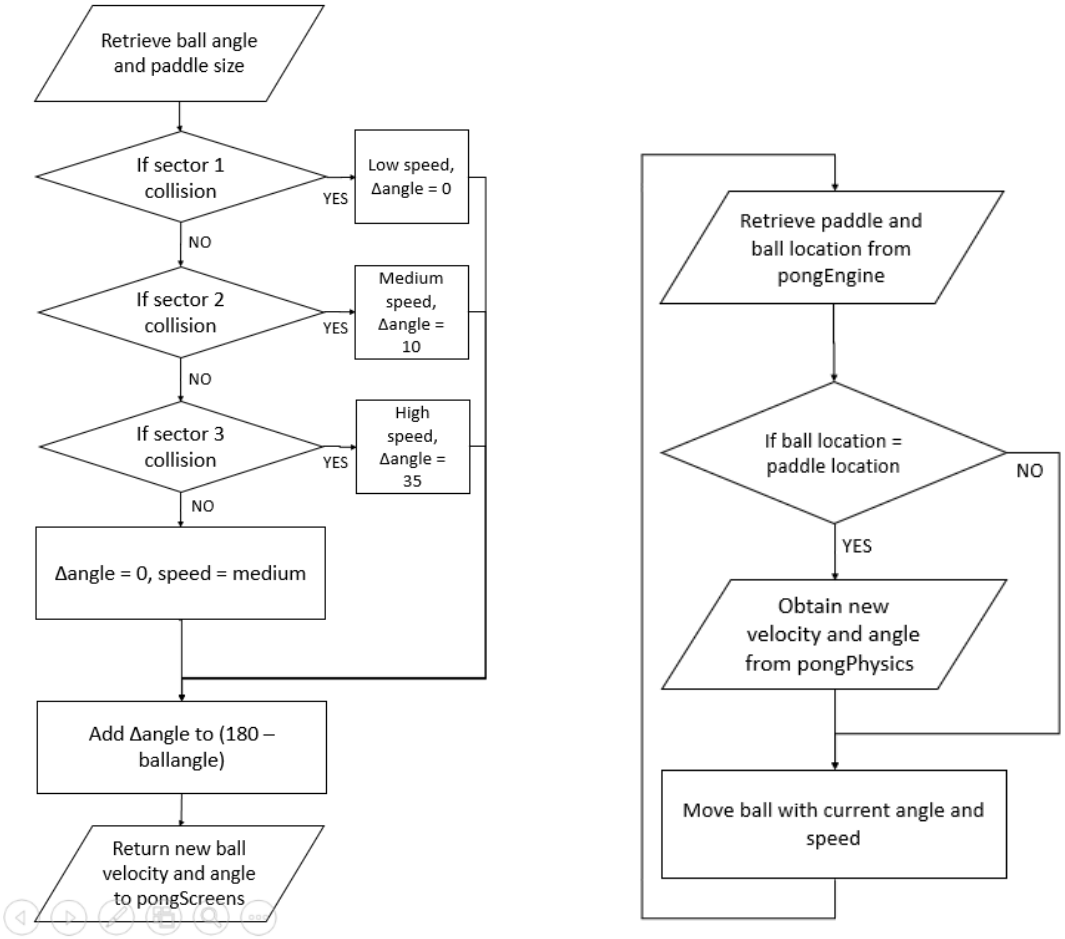


Figure 16: showing 20 degree collision with border and subsequent rebound

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 AI

The AI is extremely basic, and works by comparing the P2 paddle Y location and the ball Y location. At each iteration of the game's while loop, the paddle moves in the direction of the ball. It was found initially that this caused the paddle to jerk, as the paddle was making movements larger than the ball was; this was remedied by adding an extra region about the paddle's centre in the Y direction, hence it only moves when the ball is more than 10 px above or below from the centre of the paddle.

7 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.

7.1 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.

7.2 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.

7.3 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.

8 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.

9 Appendix

9.1 Appendix A - sevenSeg.c

```
//This library allows the control of the 7 segment display
#include "sevenSeg.h"
//Hex numbers in an array
int HexSDisplay[16] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x77,0x7C,0x39,0x5E,0x79,0x71};

//This function sets the 7 segment displays
void SDisplay_set(int Display, int HexValue){
  //Hex memory base
  volatile int *HEX0 = (int* ) 0XFF200020;
  volatile int *HEX1 = (int* ) 0XFF200030;
  //Values
  int invClearBits = 0x7F; //inverted bits to put through and bitwise
  int shiftAmount = 8; //shift multiple amount
  int hex1Adjust = 4; //adjust amount for second memory address
  //check each number position to work out between addresses
  if(Display < 4){
    *HEX0 &= ~(invClearBits << (Display * shiftAmount)); //clear bits
    *HEX0 |= (HexSDisplay[HexValue] << (Display * shiftAmount)); //set bits
  }
  else{
    *HEX1 &= ~(invClearBits << ((Display - hex1Adjust) * shiftAmount)); //clear bits
    *HEX1 |= (HexSDisplay[HexValue] << ((Display - hex1Adjust) * shiftAmount)); //set bits
  }
}
//This function clears the seven seg displays
void SDisplay_clearAll(){
  //Hex memory base
  volatile int *HEX0 = (int* ) 0XFF200020;
  volatile int *HEX1 = (int* ) 0XFF200030;
  int zero = 0x00;//Zero
  //set displays
  *HEX0 &= zero; //clear bits
  *HEX1 &= zero; //clear bits
}
//This function sets a number per pair of 7 segment displays
void SDisplay_PNum(int number, int pair){
  //Example is 15
  //Split number in 2
  int number0 = number/10; //1
  int number1 = number % 10; //2
  //calculate pair of displays from the pair set in the function call
  int pairdisplay0 = 2*pair;
  int pairdisplay1 = 2*pair+1;
  //set displays
  SDisplay_set(pairdisplay0,number1);
  SDisplay_set(pairdisplay1,number0);
}
```

9.2 Appendix B - sevenSeg.h

```
#ifndef SEVENSEG_H
#define SEVENSEG_H

void SDisplay_PNum(int number, int pair);
void SDisplay_clearAll( void );
void SDisplay_set(int Display, int HexValue);

#endif
```

9.3 Appendix C - DE1SoC VGA.c

```c
#include "DE1SoC_VGA.h"

short vga_address = 0x0;  //0xC8000000
short vga_charaddr = 0x0;   //0xC9000000
volatile int pixel_buffer_start;

//Initialise addresses of VGA
void VGA_init(short vgaPixelBuffer_address, short vgaCharBuffer_address){
  //init backbuffer
  volatile int * pixel_ctrl_ptr = (int *) 0xFF203020;
  *(pixel_ctrl_ptr + 1) = 0xC8000000;
  pixel_buffer_start = *pixel_ctrl_ptr;
  *(pixel_ctrl_ptr + 1) = 0xC0000000;
  vga_address = vgaPixelBuffer_address;
  vga_charaddr = vgaCharBuffer_address;
  //Clear VGA
  VGA_clearScreen();
}
//This function gets the address of a pixel and sets it to a colour
void VGA_drawPixel(int x, int y, short colour){
  //Call address of pixel
    //Address base + [Pixel Y][Pixel X]0
  volatile short *vga_addr=(volatile short*)(0xC8000000 + (y<<10) + (x<<1));
  *vga_addr=colour; //Set pixel to colour
}
//This function sets all pixels to black
void VGA_clearScreen(){
    int x, y;
    for (x = 0; x < 320; x++) {
      for (y = 0; y < 240; y++) {
        //set all pixels to black
        VGA_drawPixel(x,y,0x0000);
    }
    }
}
//This function sets all pixels to the desired colour
void VGA_fillColour(short colour){
    int x, y;
    for (x = 0; x < 320; x++) {
      for (y = 0; y < 240; y++) {
        //set all pixels to colour
        VGA_drawPixel(x,y,colour);
    }
    }
}
```

9.4 Appendix D - DE1SoC VGA.h

```h
#ifndef DE1SOC_VGA_H
#define DE1SOC_VGA_H

void VGA_init(short vgaPixelBuffer_address, short vgaCharBuffer_address);
void VGA_drawPixel(int x, int y, short colour);
void VGA_drawPixelToHwBuffer(int x, int y, short colour);
void VGA_BufferSwap( void );
void VGA_clearScreen( void );
void VGA_fillColour(short colour);

#endif
```

9.5 Appendix E - pongDisplay.c

```
//This is the layer of code between the game and the display devices. It utalises a frame buffer.
#include "pongDisplay.h"
int PixelHeight = 240; //Screen height
int PixelWidth = 320; //Screen Width
int totalPixel = 76800; //Total Pixels
/*--- Global Variables for frame buffers ---*/
//Single framebuffer
volatile short frontFrameBuffer[320][240];
volatile short rearFrameBuffer[320][240];
//Quad framebuffer
volatile short frontFrameBuffer1[160][120];
volatile short rearFrameBuffer1[160][120];
volatile short frontFrameBuffer2[160][120];
volatile short rearFrameBuffer2[160][120];
volatile short frontFrameBuffer3[160][120];
volatile short rearFrameBuffer3[160][120];
volatile short frontFrameBuffer4[160][120];
volatile short rearFrameBuffer4[160][120];
//OCTO framebuffer
volatile short OCTOfrontFrameBuffer1[80][120];
volatile short OCTOrearFrameBuffer1[80][120];
volatile short OCTOfrontFrameBuffer2[80][120];
volatile short OCTOrearFrameBuffer2[80][120];
volatile short OCTOfrontFrameBuffer3[80][120];
volatile short OCTOrearFrameBuffer3[80][120];
volatile short OCTOfrontFrameBuffer4[80][120];
volatile short OCTOrearFrameBuffer4[80][120];
volatile short OCTOfrontFrameBuffer5[80][120];
volatile short OCTOrearFrameBuffer5[80][120];
volatile short OCTOfrontFrameBuffer6[80][120];
volatile short OCTOrearFrameBuffer6[80][120];
volatile short OCTOfrontFrameBuffer7[80][120];
volatile short OCTOrearFrameBuffer7[80][120];
volatile short OCTOfrontFrameBuffer8[80][120];
volatile short OCTOrearFrameBuffer8[80][120];
//defines
int modeSet = 1;
int frameskip = 0;
int framecount = 0;
//Safety Window
int minX = 0;
int minY = 0;
int maxX = 320;
int maxY = 240;

//This function initialises the displays by setting the addresses
void Displays_init(volatile int vga_PixelAddress, unsigned volatile int vga_CharacterAddress, unsigned volatile int lcd_pio_base,
     unsigned volatile int lcd_hw_base){
  //init VGA
  VGA_init(vga_PixelAddress,vga_CharacterAddress);
  //init LP24
  LT24_initialise(lcd_pio_base,lcd_hw_base);
  //Set framebuffers black

}
//This function sets the frameskip amount.
void Displays_frameSkip(int skipamount){
  frameskip = skipamount;
}
//This function sets the display mode and then initialises the values.
void Displays_mode(int mode){
  int x = 0; //counter for x
  int y = 0; //counter for y
  modeSet = mode; //set global variable
  //if single software buffer
  if(modeSet == SOFTWAREFB){
    //for loops to set values to zero in 2 dimensional array
    for(y = 0; y < PixelHeight; y++){
      for(x = 0; x < PixelWidth; x++){
       frontFrameBuffer[x][y] = 0x000;
       rearFrameBuffer[x][y] = 0x000;
      }
    }
  }
  //if software buffer split into 4
   else if(modeSet == SOFTWAREQUADFB || modeSet == FASTFB){
    //for loops to set values to zero in 2 dimensional array
    for(y = 0; y < 120; y++){
      for(x = 0; x < 160; x++){
        frontFrameBuffer1[x][y] = 0x000;
        rearFrameBuffer1[x][y] = 0x000;
        frontFrameBuffer2[x][y] = 0x000;
        rearFrameBuffer2[x][y] = 0x000;
        frontFrameBuffer3[x][y] = 0x000;
        rearFrameBuffer3[x][y] = 0x000;
        frontFrameBuffer4[x][y] = 0x000;
        rearFrameBuffer4[x][y] = 0x000;
      }
    }
  }
  //if software buffer split into 8
   else if (modeSet == SOFTWAREOCTOFB){
    //for loops to set values to zero in 2 dimensional array
    for(y = 0; y < 120; y++){
```

```
        for(x = 0; x < 160; x++){
          OCTOfrontFrameBuffer1[x][y] = 0x000;
          OCTOrearFrameBuffer1[x][y] = 0x000;
          OCTOfrontFrameBuffer2[x][y] = 0x000;
          OCTOrearFrameBuffer2[x][y] = 0x000;
          OCTOfrontFrameBuffer3[x][y] = 0x000;
          OCTOrearFrameBuffer3[x][y] = 0x000;
          OCTOfrontFrameBuffer4[x][y] = 0x000;
          OCTOrearFrameBuffer4[x][y] = 0x000;
          OCTOfrontFrameBuffer5[x][y] = 0x000;
          OCTOrearFrameBuffer5[x][y] = 0x000;
          OCTOfrontFrameBuffer6[x][y] = 0x000;
          OCTOrearFrameBuffer6[x][y] = 0x000;
          OCTOfrontFrameBuffer7[x][y] = 0x000;
          OCTOrearFrameBuffer7[x][y] = 0x000;
          OCTOfrontFrameBuffer8[x][y] = 0x000;
          OCTOrearFrameBuffer8[x][y] = 0x000;
        }
      }
    }
}
//This function set the pixel to the hardware
void Displays_drawPixel(int x, int y, short colour){
  VGA_drawPixel(x,y, colour); //Set pixel on VGA hardware
  LT24_drawPixel(colour,240-y,x); //Set pixel on LT24 hardware
  ResetWDT(); //reset watchdog
}

//This function clears the screen
void Displays_clearScreen(){
  VGA_clearScreen(); //clear screen VGA
  LT24_clearDisplay(0x000); //clear screen LT24
  Displays_fillColour(0); //set frame buffer values to 0
}

//This function refreshes the screens from the framebuffer and checks the mode.
void Displays_Refresh(){
  //check the mode
  if(frameskip==0){
    if(modeSet == 0){
    }
    else if(modeSet == SOFTWAREFB){
      DisplaysLocal_singleRefresh();
    }
    else if(modeSet == SOFTWAREQUADFB){
      DisplaysLocal_quadRefresh();
    }
    else if(modeSet == SOFTWAREOCTOFB){
      DisplaysLocal_octoRefresh();
    }
    else if(modeSet == HARDWAREFB){

    }
  }
  else{
    if(frameskip == framecount){
      framecount = 0;
      if(modeSet == 0){

      }
      else if(modeSet == SOFTWAREFB){
        DisplaysLocal_singleRefresh();
      }
      else if(modeSet == SOFTWAREQUADFB){
        DisplaysLocal_quadRefresh();
      }
      else if(modeSet == SOFTWAREOCTOFB){
        DisplaysLocal_octoRefresh();
      }
      else if(modeSet == HARDWAREFB){

      }
    }
    else{
      framecount++;
    }
  }
}

//This forces a refresh regardless of the frameskip value
void Displays_forceRefresh(){
  if(modeSet == 0){

    }
    else if(modeSet == SOFTWAREFB){
      DisplaysLocal_singleRefresh();
    }
    else if(modeSet == SOFTWAREQUADFB){
      DisplaysLocal_quadRefresh();
    }
    else if(modeSet == SOFTWAREOCTOFB){
      DisplaysLocal_octoRefresh();
    }
    else if(modeSet == HARDWAREFB){

    }
    else{
```

```
      if(modeSet == 0){

      }
      else if(modeSet == SOFTWAREFB){
        DisplaysLocal_singleRefresh();
      }
      else if(modeSet == SOFTWAREQUADFB){
        DisplaysLocal_quadRefresh();
      }
      else if(modeSet == SOFTWAREOCTOFB){
        DisplaysLocal_octoRefresh();
      }
      else if(modeSet == HARDWAREFB){

      }
    }
}

//This sets the display window which makes sure that the display can not set outside of these values which prevents errors
void Displays_setWindow(int minx, int miny, int maxx, int maxy){
  minX = minx;
  minY = miny;
  maxX = maxx;
  maxY = maxy;
}

//This sets the pixel to the framebuffer
void Displays_setPixel(int x, int y, short colour){
  //check if within window
  //int z;
  //int found = 0;
  //checks if within window set
  if(x >= minX && x < maxX && y >= minY && y < maxY){
    //if no framebuffer draw straight to the display hardware
    if(modeSet == NOFRAMEBUFFER){
      Displays_drawPixel(x,y,colour);
    }
    //if single framebuffer then set the buffer
    else if(modeSet == SOFTWAREFB){
      frontFrameBuffer[x][y] = colour;
    }
    //if quad framebuffer then check which segment of screen pixel in and set framebuffer
    else if(modeSet == SOFTWAREQUADFB){
      if(x<160 && y<120){
        frontFrameBuffer1[x][y] = colour;
      }
      else if(x>=160 && y<120){
        frontFrameBuffer2[x-160][y] = colour;
      }
      else if(x<160 && y>=120){
        frontFrameBuffer3[x][y-120] = colour;
      }
      else if(x>=160 && y>=120){
        frontFrameBuffer4[x-160][y-120] = colour;
      }
    }
    //if octo framebuffer then check which segment of screen pixel in and set framebuffer
    else if(modeSet == SOFTWAREOCTOFB){
      if(x<80 && y<120){
        OCTOfrontFrameBuffer1[x][y] = colour;
      }
      else if(x>=80 && y<120 && x<160){
        OCTOfrontFrameBuffer2[x-80][y] = colour;
      }
      else if(x>=160 && y<120 && x<240){
        OCTOfrontFrameBuffer3[x-160][y] = colour;
      }
      else if(x>=240 && y<120){
        OCTOfrontFrameBuffer4[x-240][y] = colour;
      }
      else if(x<80 && y>=120){
        OCTOfrontFrameBuffer5[x][y-120] = colour;
      }
      else if(x>=80 && y>=120 && x<160){
        OCTOfrontFrameBuffer6[x-80][y-120] = colour;
      }
      else if(x>=160 && y>=120 && x<240){
        OCTOfrontFrameBuffer7[x-160][y-120] = colour;
      }
      else if(x>=240 && y>=120){
        OCTOfrontFrameBuffer8[x-240][y-120] = colour;
      }
    }
  }
}

//This gets the pixel set on the display.
short Displays_getPixel(int x, int y){
  short pixel = 0x000;
  if(modeSet == 0){
    pixel = 0x000;
  }
  else if(modeSet == SOFTWAREFB){
    pixel = rearFrameBuffer[x][y];
  }
  else if(modeSet == SOFTWAREQUADFB || modeSet == FASTFB){
    if(x<160 && y<120){
```

```
      pixel = frontFrameBuffer1[x][y];
    }
    else if(x>=160 && y<120){
      pixel = frontFrameBuffer2[x-160][y];
    }
    else if(x<160 && y>=120){
      pixel = frontFrameBuffer3[x][y-120];
    }
    else if(x>=160 && y>=120){
      pixel = frontFrameBuffer4[x-160][y-120];
    }
  }
  else if(modeSet == SOFTWAREOCTOFB){
    if(x<80 && y<120){
      pixel = OCTOfrontFrameBuffer1[x][y];
    }
    else if(x>=80 && y<120 && x<160){
      pixel = OCTOfrontFrameBuffer2[x-80][y];
    }
    else if(x>=160 && y<120 && x<240){
      pixel = OCTOfrontFrameBuffer3[x-160][y];
    }
    else if(x>=240 && y<120){
      pixel = OCTOfrontFrameBuffer4[x-240][y];
    }
    else if(x<80 && y>=120){
      pixel = OCTOfrontFrameBuffer5[x][y-120];
    }
    else if(x>=80 && y>=120 && x<160){
      pixel = OCTOfrontFrameBuffer6[x-80][y-120];
    }
    else if(x>=160 && y>=120 && x<240){
      pixel = OCTOfrontFrameBuffer7[x-160][y-120];
    }
    else if(x>=240 && y>=120){
      pixel = OCTOfrontFrameBuffer8[x-240][y-120];
    }
  }
  return pixel;
}

//FrameBuffer Refreshes
//This function refreshes the octo framebuffer
void DisplaysLocal_octoRefresh(){
  int x; //x counter
  int y; //y counter
  //compare front and rear buffers using memcmp
  int o1change = memcmp(OCTOfrontFrameBuffer1,OCTOrearFrameBuffer1, sizeof(OCTOfrontFrameBuffer1));
  int o2change = memcmp(OCTOfrontFrameBuffer2,OCTOrearFrameBuffer2, sizeof(OCTOfrontFrameBuffer2));
  int o3change = memcmp(OCTOfrontFrameBuffer3,OCTOrearFrameBuffer3, sizeof(OCTOfrontFrameBuffer3));
  int o4change = memcmp(OCTOfrontFrameBuffer4,OCTOrearFrameBuffer4, sizeof(OCTOfrontFrameBuffer4));
  int o5change = memcmp(OCTOfrontFrameBuffer5,OCTOrearFrameBuffer5, sizeof(OCTOfrontFrameBuffer5));
  int o6change = memcmp(OCTOfrontFrameBuffer6,OCTOrearFrameBuffer6, sizeof(OCTOfrontFrameBuffer6));
  int o7change = memcmp(OCTOfrontFrameBuffer7,OCTOrearFrameBuffer7, sizeof(OCTOfrontFrameBuffer7));
  int o8change = memcmp(OCTOfrontFrameBuffer8,OCTOrearFrameBuffer8, sizeof(OCTOfrontFrameBuffer8));
  //if any changes whilst running through loop then update frame buffer and set on hardware.
  for(y = 0; y < PixelHeight/2; y++){
    for(x = 0; x < PixelWidth/4; x++){
      if(o1change != 0){
        if(OCTOfrontFrameBuffer1[x][y] != OCTOrearFrameBuffer1[x][y]){
          Displays_drawPixel(x,y,OCTOfrontFrameBuffer1[x][y]); //set on hardware
          OCTOrearFrameBuffer1[x][y] = OCTOfrontFrameBuffer1[x][y]; //update FB
        }
      }
      if(o2change != 0){
        if(OCTOfrontFrameBuffer2[x][y] != OCTOrearFrameBuffer2[x][y]){
          Displays_drawPixel(x+80,y,OCTOfrontFrameBuffer2[x][y]); //set on hardware
          OCTOrearFrameBuffer2[x][y] = OCTOfrontFrameBuffer2[x][y]; //update FB
        }
      }
      if(o3change != 0){
        if(OCTOfrontFrameBuffer3[x][y] != OCTOrearFrameBuffer3[x][y]){
          Displays_drawPixel(x+160,y,OCTOfrontFrameBuffer3[x][y]); //set on hardware
          OCTOrearFrameBuffer3[x][y] = OCTOfrontFrameBuffer3[x][y]; //update FB
        }
      }
      if(o4change != 0){
        if(OCTOfrontFrameBuffer4[x][y] != OCTOrearFrameBuffer4[x][y]){
          Displays_drawPixel(x+240,y,OCTOfrontFrameBuffer4[x][y]); //set on hardware
          OCTOrearFrameBuffer4[x][y] = OCTOfrontFrameBuffer4[x][y];  //update FB
        }
      }
      if(o5change != 0){
        if(OCTOfrontFrameBuffer5[x][y] != OCTOrearFrameBuffer5[x][y]){
          Displays_drawPixel(x,y+120,OCTOfrontFrameBuffer5[x][y]); //set on hardware
          OCTOrearFrameBuffer5[x][y] = OCTOfrontFrameBuffer5[x][y]; //update FB
        }
      }
      if(o6change != 0){
        if(OCTOfrontFrameBuffer6[x][y] != OCTOrearFrameBuffer6[x][y]){
          Displays_drawPixel(x+80,y+120,OCTOfrontFrameBuffer6[x][y]); //set on hardware
          OCTOrearFrameBuffer6[x][y] = OCTOfrontFrameBuffer6[x][y]; //update FB
        }
      }
      if(o7change != 0){
        if(OCTOfrontFrameBuffer7[x][y] != OCTOrearFrameBuffer7[x][y]){
          Displays_drawPixel(x+160,y+120,OCTOfrontFrameBuffer7[x][y]); //set on hardware
```

```
          OCTOrearFrameBuffer7[x][y] = OCTOfrontFrameBuffer7[x][y]; //update FB
        }
      }
      if(o8change != 0){
        if(OCTOfrontFrameBuffer8[x][y] != OCTOrearFrameBuffer8[x][y]){
          Displays_drawPixel(x+240,y+120,OCTOfrontFrameBuffer8[x][y]); //set on hardware
          OCTOrearFrameBuffer8[x][y] = OCTOfrontFrameBuffer8[x][y]; //update FB
        }
      }
    }
    }

}

//This function refreshes the quad framebuffer
void DisplaysLocal_quadRefresh(){
  int x; //x counter
  int y; //y counter
  //compare front and rear buffers using memcmp
  int q1change = memcmp(frontFrameBuffer1,rearFrameBuffer1, sizeof(frontFrameBuffer1));
  int q2change = memcmp(frontFrameBuffer2,rearFrameBuffer2, sizeof(frontFrameBuffer2));
  int q3change = memcmp(frontFrameBuffer3,rearFrameBuffer3, sizeof(frontFrameBuffer3));
  int q4change = memcmp(frontFrameBuffer4,rearFrameBuffer4, sizeof(frontFrameBuffer4));
  //if any changes whilst running through loop then update frame buffer and set on hardware.
  for(y = 0; y < PixelHeight/2; y++){
      for(x = 0; x < PixelWidth/2; x++){
        if(q1change != 0){
          if(frontFrameBuffer1[x][y] != rearFrameBuffer1[x][y]){
            Displays_drawPixel(x,y,frontFrameBuffer1[x][y]); //set on hardware
            rearFrameBuffer1[x][y] = frontFrameBuffer1[x][y]; //update FB
          }
        }
        if(q2change != 0){
          if(frontFrameBuffer2[x][y] != rearFrameBuffer2[x][y]){
            Displays_drawPixel(x+160,y,frontFrameBuffer2[x][y]); //set on hardware
            rearFrameBuffer2[x][y] = frontFrameBuffer2[x][y]; //update FB
          }
        }
        if(q3change != 0){
          if(frontFrameBuffer3[x][y] != rearFrameBuffer3[x][y]){
            Displays_drawPixel(x,y+120,frontFrameBuffer3[x][y]); //set on hardware
            rearFrameBuffer3[x][y] = frontFrameBuffer3[x][y]; //update FB
          }
        }
        if(q4change != 0){
          if(frontFrameBuffer4[x][y] != rearFrameBuffer4[x][y]){
            Displays_drawPixel(x+160,y+120,frontFrameBuffer4[x][y]); //set on hardware
            rearFrameBuffer4[x][y] = frontFrameBuffer4[x][y]; //update FB
          }
        }
      }
  }
}
//This function find changes and sets them in hardware.
void DisplaysLocal_singleRefresh(){
  int x;
  int y;
  for(y = 0; y < PixelHeight; y++){
    for(x = 0; x < PixelWidth; x++){
      short colour = frontFrameBuffer[x][y];
      if(frontFrameBuffer[x][y] != rearFrameBuffer[x][y]){
        short colour = frontFrameBuffer[x][y];
        Displays_drawPixel(x,y,colour);
        rearFrameBuffer[x][y] = colour;
      }
    }

  }
}
//This function fills the displays a set colour.
void Displays_fillColour(short colour){
  int x;
  int y;
  VGA_fillColour(colour);
  LT24_clearDisplay(colour);
  for(y = 0; y < 240; y++){
    for(x = 0; x < 320; x++){
      Displays_setPixel(x,y,colour);
    }
  }
}
```

9.6 Appendix F - pongDisplay.h

```
#ifndef PONGDISPLAY_H
#define PONGDISPLAY_H

#include "../DE1Soc_LT24/DE1Soc_LT24.h"
#include "../DE1Soc_VGA/DE1Soc_VGA.h"
#include "../HPS_Watchdog/HPS_Watchdog.h"
#include <string.h>

#define NOFRAMEBUFFER 0
#define SOFTWAREFB 1
#define SOFTWAREQUADFB 2
#define SOFTWAREOCTOFB 3
#define HARDWAREFB 4
#define FASTFB 5

void Displays_init(volatile int vga_PixelAddress, unsigned volatile int vga_CharacterAddress, unsigned volatile int lcd_pio_base,
     unsigned volatile int lcd_hw_base);
void Displays_drawPixel(int x, int y, short colour);
void Displays_frameSkip(int skipamount);
void Displays_clearScreen( void );
void Displays_Refresh( void );
void Displays_forceRefresh( void );
void Displays_setPixel(int x, int y, short colour);
void Displays_setWindow(int minx, int miny, int maxx, int maxy);
short Displays_getPixel(int x, int y);
void Displays_mode(int mode);
void Displays_fillColour(short colour);
void Displays_mode(int mode);

//Functions not intended to be used outside of pongDisplay.cpp

void DisplaysLocal_singleRefresh( void );
void DisplaysLocal_quadRefresh( void );
void DisplaysLocal_octoRefresh( void );
#endif
```

9.7 Appendix G - pongEngine.c

```
#include "pongEngine.h"

//320x240

/*############ Ball Globals ############*/
//Calculated ball path
struct ballPathInst{
  int x;
  int y;
};
//instruction counter
int ballPathInstCounter = 0;
//current instruction
int ballCurrentPosPath = 0;
//array of structs
struct ballPathInst BallPath[1000];

int ballAngle = 361; //Set to impossible angle to ensure on initialisation there
short ballColour = 0xFFFF; //Ball colour
int ballX = centre_x; //ball defaults
int ballY = centre_y; //ball defaults
//Thresholds
int ballMinX = 8;
int ballMaxX = 312;
int ballMinY = 23;
int ballMaxY = 232;

/*############ Paddle Globals ############*/
//Paddle thresholds
int paddleMinY = 50;
int paddleMaxY = 210;
//Paddle Locations
int paddle1Y = centre_y;
int paddle1X = 50;
int paddle2Y = centre_y;
int paddle2X = 270;
//Paddle Colours
int paddle1Colour = 0xFFFF;
int paddle2Colour = 0xFFFF;
/*############ Score Keeping Globals############*/
int player1Score = 0;
int player2Score = 0;
/*############ General Functions ############*/
//This function initialises the pong engine
void pongEngine_init(){
  int topAdjust = 15; //VGA Pixel faults from y
  Displays_clearScreen(); //clear screen
  pongEngine_resetPaddles(); //reset paddles
  pongEngine_resetBallLoc(); //reset ball location

  pongEngine_paddleCreate(1); //create paddle
  pongEngine_paddleCreate(2); //create paddle


  //create arena
  //create green area
  pongSprites_renderRectangle(320,0, topAdjust, 0, (0x1F << 11)); //to create red border on LCD
  paddleMinY = pongSprites_getPaddleSizeY()/2;//set limits
  paddleMaxY = 240-paddleMinY; //set limits

  pongEngine_paddleSetYLimits(paddleMaxY, paddleMinY+topAdjust+1); //adjust limits
}
/*############ Score Keeping ############*/
//This function adds points to a player
void pongEngine_addPoint(int player){
  //check player and add point to player
  if(player == 1){
    player1Score++;
  }
  if(player == 2){
    player2Score++;
  }
  //refresh score
  pongEngine_refreshScore();
}
//This function resets the score to zero
void pongEngine_resetScore(unsigned int _Refresh){
  player1Score = 0;
  player2Score = 0;
  if (_Refresh != 0){
    pongEngine_refreshScore(); //refresh score
  }
}

//refresh score to board seven seg and on screen
void pongEngine_refreshScore(){
  //create char
  char playerscorech[6];
  //split numbers
  int P1number0, P1number1, P2number0, P2number1;

  // Fix overflow past 100 - loops to zero
  player1Score = player1Score %100;
  player2Score = player2Score %100;
```

```
  //split numbers into individual characters
  P1number0 = player1Score/10; //1
  P1number1 = player1Score % 10; //5
  P2number0 = player2Score/10; //1
  P2number1 = player2Score % 10; //5
  //updates 7 seg
  SDisplay_PNum(player1Score,2);
  SDisplay_PNum(player2Score,0);
  //updates score
  playerscorech[0] = P1number0 + '0';
  playerscorech[1] = P1number1 + '0';
  playerscorech[2] = ' ';
  playerscorech[3] = P2number0 + '0';
  playerscorech[4] = P2number1 + '0';
  playerscorech[5] = '\0';
  //clears score by drawing black rectangle on it
  pongSprites_renderRectangle(centre_x - 30,centre_x+ 30 , 17, 17+14, 0x00);
  //writes score on screen
  pongSprites_writeText(centre_x - 28, 17, 1, playerscorech, 0xFFFF);
  //Displays_forceRefresh();
}

/*############ Ball Functions ############*/
//This function moves the pong ball
void pongEngine_moveBall(int angle, int speed){
  ResetWDT(); //reset watchdog
  pongEngine_destroyBall(); //destroy the ball so its black
  //Check if at thresholds
  if(ballX < ballMinX){
    ballX = ballMinX;
    ballCurrentPosPath = ballCurrentPosPath-speed;
  }
  if(ballX > ballMaxX){
    ballX = ballMaxX;
    ballCurrentPosPath = ballCurrentPosPath-speed;
  }
  if(ballY < ballMinY){
    ballY = ballMinY;
    ballCurrentPosPath = ballCurrentPosPath-speed;
  }
  if(ballY > ballMaxY){
    ballY = ballMaxY;
    ballCurrentPosPath = ballCurrentPosPath-speed;
  }
  //If angle change then calculate new instructions
  if(angle == ballAngle){
    //To prevent overflow
    if(ballCurrentPosPath > ballPathInstCounter){
      ballCurrentPosPath = ballPathInstCounter;
    }
    //render ball and move position
    ballX = BallPath[ballCurrentPosPath].x;
    ballY = BallPath[ballCurrentPosPath].y;
    pongSprites_renderBall(ballX, ballY, ballColour);
    ResetWDT();
    //Move ball speed amount
    ballCurrentPosPath = ballCurrentPosPath+speed;
  }
  //if ball is at a different angle
  else{
    ballAngle = angle; //set ballAngle to angle
    ResetWDT();
    //Calculate new path
    pongEngine_calcBallPathInst(angle);
    //render ball and move position
    ballX = BallPath[ballCurrentPosPath].x;
    ballY = BallPath[ballCurrentPosPath].y;
    pongSprites_renderBall(ballX, ballY, ballColour);
    //Move ball speed amount
    ballCurrentPosPath = ballCurrentPosPath+speed;
  }
}
//reset ball location
void pongEngine_resetBallLoc( void ){
  //pongEngine_setBallLocation(centre_x, centre_y); // Avoiding extra ball on startup
  ballX = centre_x;
  ballY = centre_y;
}
//set ball location
void pongEngine_setBallLocation(int x, int y){
  pongEngine_destroyBall();
  ResetWDT();
  ballX = x;
  ballY = y;
  pongEngine_calcBallPathInst(ballAngle);
  pongSprites_renderBall(ballX, ballY, ballColour);
  ballCurrentPosPath++;
}
//get ball angle
int pongEngine_getBallAngle(void){
  return ballAngle;
}
//get ball location x
int pongEngine_getBallLocation_x(void){
  return ballX;
}
//get ball location y
```

```
int pongEngine_getBallLocation_y(void){
  return ballY;
}
//create the ball
void pongEngine_createBall(void){
  pongSprites_renderBall(ballX, ballY, ballColour);
}
//destroy the ball
void pongEngine_destroyBall(void){
  pongSprites_renderBall(ballX, ballY, 0x0000);
}
//calculate the ball path instructions
void pongEngine_calcBallPathInst(int angle){
  int x;
  int newangle;
  int foundAngle = 0;
  int angleX;
  int angleY;
  //Prevent floating point error at right angles.
  if(angle == 0){
    foundAngle = 1;
    angleX = 0;
    angleY = ballY;
  }
  if(angle == 90){
    foundAngle = 1;
    angleX = ballX;
    angleY = 0;
  }
  if(angle == 180){
    foundAngle = 1;
    angleX = 320;
    angleY = ballY;
  }
  if(angle == 270){
    foundAngle = 1;
    angleX = ballX;
    angleY = 240;
  }
  //Finds angle by working out angle between 2 points (one being the centre of ball, one being edge of screen)
  //if angle found do not calculate the angle
  if(foundAngle == 0){
    for(x = 0; x<240; x++){
      newangle = pongEngine_calcAngle(ballX,ballY,0,x); //calculate angle
      if(newangle == angle){
        foundAngle = 1;
        angleX = 0;
        angleY = x;
      }
    }
  }
  if(foundAngle == 0){
    for(x = 0; x<240; x++){
      newangle = pongEngine_calcAngle(ballX,ballY,320,x); //calculate angle
      if(newangle == angle){
        foundAngle = 1;
        angleX = 320;
        angleY = x;
      }
    }
  }
  if(foundAngle == 0){
    for(x = 0; x<320; x++){
      newangle = pongEngine_calcAngle(ballX,ballY,x,0); //calculate angle
      if(newangle == angle){
        foundAngle = 1;
        angleX = x;
        angleY = 0;
      }
    }
  }
  if(foundAngle == 0){
    for(x = 0; x<320; x++){
      newangle = pongEngine_calcAngle(ballX,ballY,x,240); //calculate angle
      if(newangle == angle){
        foundAngle = 1;
        angleX = x;
        angleY = 240;
      }
    }
  }
  //generate ball instructions
  pongEngine_genBallPathInst(ballX, ballY, angleX, angleY);
}

//Line drawing code from Assignement 1 - Alexander Bolton
void pongEngine_genBallPathInst(int x1, int y1, int x2, int y2){
  //REFERENCE: drawLine using Bresenhams algorithm. https://rosettacode.org/wiki/Bitmap/Bresenham%27s_line_algorithm
  //reset ballPathCounts

  //calculate deltas
  int dx =  abs (x2 - x1);
  int dy = -abs (y2 - y1);
  //calculate error
  int error = dx + dy;
  int error2;
  //Find quadrant
```

```
  int sy;
  int sx;
  ballPathInstCounter = 0;
  ballCurrentPosPath = 0;
  if(x1<x2){
    sx = 1;
  }
  else{
    sx = -1;
  }

  if(y1<y2){
    sy = 1;
  }
  else{
    sy = -1;
  }
  //Loop though and calculate line pixels
  while(1){
    //Generate Instructions
    BallPath[ballPathInstCounter].x = x1;
    BallPath[ballPathInstCounter].y = y1;
    ballPathInstCounter++;
    if (x1 == x2 && y1 == y2){ break;}
    error2 = 2 * error;
    //if error2 is larger than delta y then add 1 to x
    if (error2 >= dy) {
      error += dy;
      x1 += sx;
    }
    //if error2 is smaller than delta x then add 1 to y
    if (error2 <= dx) {
      error += dx;
      y1 += sy;
    }
  }
}

//This function calculates the angle between 2 points
int pongEngine_calcAngle(int x1, int y1, int x2, int y2){
  //calculate deltas
  double diff_y = y1 - y2;
  double diff_x = x1 - x2;
  //calculate angle
  double angle = atan2(diff_y, diff_x);
  //turn to degrees
  float ang_d = angle * 180/PI;
  //turn to integer
  int ang_dint = ang_d;
  //return
  return ang_dint;
}
/*############ Paddle Functions ############*/
//set paddle limits on y axis
void pongEngine_paddleSetYLimits(int maxy, int miny){
  paddleMinY = miny+15;
  paddleMaxY = maxy;
}
//set paddle y location
void pongEngine_paddleSetYLocation(int player, int y){
  pongEngine_paddleDestroy(player); //destroy paddle to turn to black
  //check thresholds
  if(y>paddleMaxY){
    y = paddleMaxY;
  }
  if(y<paddleMinY){
    y = paddleMinY;
  }
  //check which player and move appropriate paddle
  if(player == 1){
    paddle1Y = y;
    pongSprites_renderPaddle(paddle1X, y, paddle1Colour); //render paddle
  }
  if(player == 2){
    paddle2Y = y;
    pongSprites_renderPaddle(paddle2X, y, paddle2Colour); //render paddle
  }
}

//set paddle x location
void pongEngine_paddleSetXLocation(int player, int x){
  pongEngine_paddleDestroy(player); //destroy paddle
  //check player and set appropriate paddle
  if(player == 1){
    paddle1X = x;
    pongSprites_renderPaddle(x, paddle1Y, paddle1Colour); //render paddle
  }
  if(player == 2){
    paddle2X = x;
    pongSprites_renderPaddle(x, paddle1Y, paddle2Colour); //render paddle
  }
}
//This function moves the paddle
void pongEngine_paddleMove(int player, int direction, int speed){
  /*
  UP - 1
  DOWN - 0
```

```
  */
  int moveto;
  //check player
  if(player == 1){
    //check direction
    if(direction == UP){
      moveto = paddle1Y+speed; //calculate y to move to
      pongEngine_paddleSetYLocation(1, moveto); //render
    }
    if(direction == DOWN){
      moveto = paddle1Y-speed; //calculate y to move to
      pongEngine_paddleSetYLocation(1, moveto); //render
    }
  }
  //check player
  if(player == 2){
    //check direction
    if(direction == UP){
      moveto = paddle2Y+speed; //calculate y to move to
      pongEngine_paddleSetYLocation(2, moveto); //render
    }
    if(direction == DOWN){
      moveto = paddle2Y-speed; //calculate y to move to
      pongEngine_paddleSetYLocation(2, moveto); //render
    }
  }
}
//This function creates the paddle
void pongEngine_paddleCreate(int player){
  //check player
  if(player == 1){
    pongSprites_renderPaddle(paddle1X, paddle1Y, paddle1Colour); //render paddle
  }
  if(player == 2){
    pongSprites_renderPaddle(paddle2X, paddle2Y, paddle2Colour); //render paddle
  }
}
//This function destroys the paddle
void pongEngine_paddleDestroy(int player){
  //check player
  if(player == 1){
    pongSprites_renderPaddle(paddle1X, paddle1Y, 0x0000); //render paddle black
  }
  if(player == 2){
    pongSprites_renderPaddle(paddle2X, paddle2Y, 0x0000); //render paddle black
  }
}
//This function resets the paddles
void pongEngine_resetPaddles(){
  pongEngine_paddleDestroy(1); //destroy paddles
  pongEngine_paddleDestroy(2);
  pongEngine_paddleSetYLocation(1, centre_y); //set paddle location
  pongEngine_paddleSetYLocation(2, centre_y);
}
//This function gets the paddle locations
unsigned int pongEngine_getPaddleY(unsigned int paddle){
  if (paddle == 1){
    return paddle1Y;
  } else {
    return paddle2Y;
  }
}
```

9.8 Appendix H - pongEngine.h

```
#ifndef PONGENGINE_H
#define PONGENGINE_H

#include <stdlib.h>
#include <cmath>
#include "pongSprites.h"
#include "../DE1SoC_SevenSeg/sevenSeg.h"
#include "../HPS_Watchdog/HPS_Watchdog.h"
#include "../pongDisplay/pongDisplay.h"
#define PI 3.14159265359
#define centre_x 160
#define centre_y 120
#define UP 0
#define DOWN 1


//Publicly used
void pongEngine_init( void );
/*############ Ball Functions ############*/
void pongEngine_moveBall(int angle, int speed);
void pongEngine_createBall(void);
void pongEngine_destroyBall(void);
void pongEngine_setBallLocation(int x, int y);
int pongEngine_getBallAngle(void);
int pongEngine_getBallLocation_x(void);
int pongEngine_getBallLocation_y(void);
void pongEngine_resetBallLoc( void );
/*############ Paddle Functions ############*/
void pongEngine_paddleMove(int player, int direction, int speed);
void pongEngine_paddleSetYLimits(int maxy, int miny);
void pongEngine_paddleSetYLocation(int player, int y);
void pongEngine_paddleSetXLocation(int player, int x);
void pongEngine_paddleCreate(int player);
void pongEngine_paddleDestroy(int player);
unsigned int pongEngine_getPaddleY(unsigned int paddle);
/*############ Score Keeping ############*/
void pongEngine_addPoint(int player);
void pongEngine_resetScore( unsigned int _Refresh );

//used within library
/*############ Ball Functions ############*/
int pongEngine_calcAngle(int x1, int y1, int x2, int y2);
void pongEngine_calcBallPathInst(int angle);
void pongEngine_genBallPathInst(int x1, int y1, int x2, int y2);
/*############ Paddle Functions ############*/
/*############ Score Keeping ############*/
void pongEngine_refreshScore( void );
void pongEngine_resetPaddles( void );
#endif
```

9.9 Appendix I - pongSprites.c

```
#include "pongSprites.h"

volatile int sprite_Ball[17][17];

int paddlesize_x = 5;
int paddlesize_y = 60;

//This function initialises the ball
void pongSprites_initBall(void){
  int x = 8;
  int y = 8;
  //Radius as signed int
  int signedr = 8;
  //Radius squared
  int rad2 = signedr * signedr;
  //Outline threshold
  //Go through x's
  int xc = 0;
  int yc = 0;
  //Loop through all X and Y of square the size of radius squared
  for (xc = -signedr; xc <= signedr; xc++) {
    for (yc = -signedr; yc <= signedr; yc++) {
      //radius squared = yc^2 + xc^2
      int pyr = (yc*yc) + (xc*xc);
      //If no fill then draw outline
      if(pyr <= rad2){
        sprite_Ball[xc+x][yc+y] = 1;
      }
      else{
        sprite_Ball[xc+x][yc+y] = 0;
      }
    }
  }
}

//This function renders the ball
void pongSprites_renderBall(int x, int y, short colour){
  int yl;
  int xl;
  //adjust so called at centre of ball
  x = x-8;
  y = y-8;
  //loop through values and check if they equal radius
  for(yl = 0; yl < 17; yl++){
    for(xl = 0; xl < 17; xl++){
      if(sprite_Ball[yl][xl] == 1){
        Displays_setPixel(x+xl, y+yl, colour); //set pixels
      }
    }
  }
}

//This function renders a paddle
void pongSprites_renderPaddle(int x, int y, short colour){
  //taken from assignment 1 - graphics library and modified for paddle
  int lly;
  int llx;
  //adjust for centre
  x = x - (paddlesize_x/2);
  y = y - (paddlesize_y/2);
  //loop through values and set pixels
  for(lly=0; lly <= paddlesize_y; lly++){
    for(llx=0; llx<=paddlesize_x; llx++){
      Displays_setPixel(x+llx,y+lly,colour); //set pixels
    }
  }
}

//This function renders a rectangle
void pongSprites_renderRectangle(int x1,int x2, int y1, int y2, int colour){
  //taken from assignment 1 - graphics library and modified for paddle
  int height = abs(y2-y1);
  int width = abs(x1-x2);
  int y=0;
  int x=0;

  //find bottom left value
  int llx = 0;
  int lly = 0;
  if(x1<x2){
    llx = x1;
  }
  else{
    llx = x2;
  }
  if(y1<y2){
    lly = y1;
  }
  else{
    lly = y2;
  }
  //set pixels
  for(y=0; y <= height; y++){
    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 }
```

9.10 Appendix J - pongSprites.h

```
#ifndef PONGSPRITES_H
#define PONGSPRITES_H

#include "../pongDisplay/pongDisplay.h"
#include "pongEngine.h"
#include "../BasicFont/BasicFont.h"
#include <string.h>

#define SMALL 0
#define LARGE 1

void pongSprites_initBall( void );
void pongSprites_renderBall(int x, int y, short colour);
void pongSprites_renderPaddle(int x, int y, short colour);
void pongSprites_writeText(int x, int y, int size, char *text, short colour);
void pongSprites_renderRectangle(int x1,int x2, int y1, int y2, int colour);
void pongSprites_changePaddleSize(unsigned int size);
unsigned int pongSprites_getPaddleSizeY( void );

#endif
```

9.11 Appendix K - pongPhysics.c

```
/*
 * pongPhysics.c
 *
 *  Created on: 24 Apr 2019
 *      Author: John
 */

#include "pongPhysics.h"

int angle;
int speed;
int rounds; //Number of rounds elapsed
int nrand = 0;
int psrand[] = {293, 112, 46, 329, 228, 35, 100, 197, 335,
    340, 57, 339, 335, 175, 288, 51, 152, 330, 285, 335,
    236, 13, 306, 336, 244, 273, 208, 141, 236, 62, 254,
    11, 100, 17, 35, 296, 250, 114, 332, 12, 158, 137, 276,
    286, 67, 176, 160, 233, 255, 282, 245, 236, 59, 43,
    179, 346, 123, 211, 260, 182, 252, 321, 345, 197,
    50, 54, 303, 293, 335, 126, 71, 222, 170, 127, 299,
    211, 198, 330, 103, 273, 271, 137, 204, 27, 19, 191,
    281, 336, 47, 205, 169, 14, 121, 58, 286, 326, 190,
    60, 217, 235, 248};

//
// Code to regulate serving rules
//

int pongPhysics_serve (void) {
  int angle = psrand[nrand % 99] - 180;
  nrand++;
  if (angle < 20){
    angle += 20;
  }
  if (angle > 340){
    angle -= 20;
  }
  return angle;
}

//
// Code to regulate horizontal border collisions
//

int pongPhysics_borderCollision (int speed, int angle) {

  int outangle;

  outangle  = -angle;

  pongEngine_moveBall(outangle, speed);   //Used to avoid ball getting stuck at the border
  pongEngine_moveBall(outangle, speed);
  pongEngine_moveBall(outangle, speed);

  ResetWDT();
  return outangle;
}

// Code to regulate paddle collisions

int* pongPhysics_paddleCollision (int speed, int angle, int player) {

int outangle = 0;
int outspeed = 0;
int deltaAngle;
float sector1 = (0.167 * pongSprites_getPaddleSizeY());
float sector2 = (0.417 * pongSprites_getPaddleSizeY());
float sector3 = (0.083 * pongSprites_getPaddleSizeY());

int output[2];

outangle  =  180 - angle;
ResetWDT();
// Paddle sector 1 collision / no change in angle or speed
if ((pongEngine_getBallLocation_y() < pongEngine_getPaddleY(player) + (int)(sector1)) && (pongEngine_getBallLocation_y() >
     pongEngine_getPaddleY(player) - (int)(sector1))) {

  deltaAngle = 0;
  outspeed   = 4;

//Paddle sector 2 collision
} else if ((pongEngine_getBallLocation_y() >= pongEngine_getPaddleY (player) + (int)(sector1)) && (pongEngine_getBallLocation_y() <=
     pongEngine_getPaddleY (player) + (int)(sector2))) {

  deltaAngle = 10;
  outspeed   = 6;

} else if ((pongEngine_getBallLocation_y() < pongEngine_getPaddleY (player) - (int)(sector1)) && (pongEngine_getBallLocation_y() >=
     pongEngine_getPaddleY (player) - (int)(sector2))) {

  deltaAngle = -10;
  outspeed  = 6;

//Paddle sector 3 collision
```

```
} else if (pongEngine_getBallLocation_y() > pongEngine_getPaddleY(player) + (int)(sector2) && pongEngine_getBallLocation_y() <=
    pongEngine_getPaddleY(player) + (int)(sector3)) {

  deltaAngle = 20;
  outspeed   = 8;

} else if (pongEngine_getBallLocation_y() < pongEngine_getPaddleY(player) - (int)(sector2) && pongEngine_getBallLocation_y() >=
    pongEngine_getPaddleY(player) - (int)(sector3)) {

  deltaAngle = -20;
  outspeed   = 8;

} else {

  outspeed = speed;

  if ((pongEngine_getBallLocation_x() <= 50 + 15) && (pongEngine_getBallLocation_x() >= 50 + 10)) {

  outangle = 135;

  } else if ((pongEngine_getBallLocation_x() >= 270 - 15) && (pongEngine_getBallLocation_x() <= 270 - 10)) {

  outangle = 45;

  }
}

outangle = outangle + deltaAngle;


if (outangle > 180) { //outangle < 360
  outangle = -(360 - outangle);
} else if (outangle < -180) {
  outangle = 360 + outangle;
} else if (outangle == 360) { // == 0 = 180 / ==180 = 0
  outangle = 0;
}

output[0] = outangle;
output[1] = outspeed;



pongEngine_moveBall(output[0], output[1]);
pongEngine_moveBall(output[0], output[1]);
pongEngine_moveBall(output[0], output[1]);
ResetWDT();

return output;


}

void resetRand(){
  nrand = 0;
}
```

9.12 Appendix L - pongPhysics.h

```
/*
 * pongPhysics.h
 *
 *  Created on: 24 Apr 2019
 *      Author: John
 */

#ifndef PONGENGINE_PONGPHYSICS_H_
#define PONGENGINE_PONGPHYSICS_H_

#include <cmath>
#include <stdlib.h>
#include "pongSprites.h"
#include "pongEngine.h"
#include "../HPS_Watchdog/HPS_Watchdog.h"
#include "../pongDisplay/pongDisplay.h"


int pongPhysics_serve (void);
int pongPhysics_borderCollision (int speed, int angle);
int* pongPhysics_paddleCollision (int speed, int angle, int player);
void resetRand( void );




#endif /* PONGENGINE_PONGPHYSICS_H_ */
```

9.13 Appendix M - pongInputs.c

```c
#include "pongInputs.h"
//#include "../HPS_IRQ/HPS_IRQ.h"
//#include "../pongEngine/pongEngine.h"
//#include "../pongDisplay/pongDisplay.h"


/* Notes: Add audio output
 *      */

/* To set function running, simply include PS2_Keyboard.h and then call inputsInitialise */
/* Requires scatter file change to FPGARomRamIRQ.scat */


volatile unsigned int *PS2_DATA = (unsigned int *) 0xFF200100;     // PS2 base address
volatile unsigned int *PS2_CONTROL = (unsigned int *) 0xFF200104; // PS2 control address
volatile unsigned int *KEY_ptr       = (unsigned int *) 0xFF200050; // Keys

volatile unsigned int keyBuffer[3] = {0,0,0};

bool inputsIsInit = false;

volatile unsigned int mode;

// https://www.avrfreaks.net/sites/default/files/PS2%20Keyboard.pdf


/* IRQ handlers */
void keyboardISR(HPSIRQSource interruptID, bool isInit, void* initParams){
  if (!isInit){
    *PS2_CONTROL |= 0x0;
    inputKeyboard();
    //*PS2_DATA |= 0x00;
    *PS2_CONTROL |= 0x1;
    emptyFIFO();
  }
  HPS_ResetWatchdog();
}

void pushbuttonISR(HPSIRQSource interruptID, bool isInit, void* initParams) {
    if (!isInit) {
        unsigned int press;
        //Read the Push-button interrupt register
        press = KEY_ptr[3];
        //Then clear the interrupt flag by writing the value back
        KEY_ptr[3] = press;

        if (press == 1){
          Input(_UP, keySpeed);
        }
        else if (press == 2){
          Input(_DOWN, keySpeed);
        }
        else if (press == 4){
          Input(_W, keySpeed);
        }
        else if (press == 8){
          Input(_S, keySpeed);
        }
        else if (press == 10){
          Input(99, keySpeed);
        }
    }
    //Reset watchdog.
    HPS_ResetWatchdog();
}


/* Actual functions */

void inputsInitialise(void) {
  if (!inputsIsInit){
    // Register IRQs
    HPS_IRQ_registerHandler(IRQ_LSC_PS2_PRIMARY, keyboardISR);
    HPS_ResetWatchdog();
    HPS_IRQ_registerHandler(IRQ_LSC_KEYS, pushbuttonISR);
    HPS_ResetWatchdog();

    // Flag keyboard as initialised
    inputsIsInit = true;
    HPS_ResetWatchdog();
  }
}

char PS2Scan( void ) {
  int keyboard_data, RVALID;
  char inputKey;

  keyboard_data = *(PS2_DATA);     // Read Data register
  RVALID = keyboard_data & 0x8000;  // Extract RVALID
  if (RVALID){
    inputKey = keyboard_data & 0xFF; HPS_ResetWatchdog();
  } else { inputKey = 0; }//Handle any odd errors
  HPS_ResetWatchdog();
  //printf("%X \n", inputKey); //Debug for checking scancodes
```

```
  return inputKey;
}

void inputKeyboard( void ) {
  char key = PS2Scan();
  keyBuffer[0] = keyBuffer[1];
  keyBuffer[1] = keyBuffer[2];
  keyBuffer[2] = PS2Scan();

  if (key == _W)        {
    keyBuffer[2] = 1; Input(_W, keyBSpeed);
  } else if (key == _S)    {
    keyBuffer[2] = 2; Input(_S, keyBSpeed);
  } else if (key == _UP)    {
    keyBuffer[2] = 3; Input(_UP, keyBSpeed);
  } else if (key == _DOWN)  {
    keyBuffer[2] = 4; Input(_DOWN, keyBSpeed);
  } else if (key == _ESC)   {
    keyBuffer[2] = 5; Input(_ESC, keyBSpeed);
  } else if (key == _RETURN)  {
    keyBuffer[2] = 6; Input(_RETURN, keyBSpeed);
  } else if ((key == _BKSPACE) && (keyBuffer[0] != 7)) {
    keyBuffer[2] = 7; toggleSound();//Input(7, keyBSpeed);
  } else if (key == _LEFT)    {
    keyBuffer[2] = 8; Input(_LEFT, keyBSpeed);
  } else if (key == _RIGHT)   {
    keyBuffer[2] = 9; Input(_RIGHT, keyBSpeed);
  }

  // Ignore make/break signals
  if ((key == 0xF0) || (key == 0xE0)) keyBuffer[2] = 0;

  // Avoid doubling motions on single press due to make/break
  if ((keyBuffer[2] == keyBuffer[0]) && (keyBuffer[1] == 0)) keyBuffer[2] = 0;

  HPS_ResetWatchdog();
}

void emptyFIFO( void ){
  int RAVAIL = *(PS2_DATA) & 0xFFFF0000;
  while (RAVAIL > 0x1000){
    char temp;
    temp = *(PS2_DATA) & 0xFF; // Read necessary to clear FIFO
    RAVAIL = *(PS2_DATA) & 0xFFFF0000;
    temp &= temp; // Get rid of annoying warning
  }
}

void Input(unsigned int key, unsigned int speed){
  if ((mode == GAME) || (mode == GAME_AI)){ // Common to games
    if (key == _W){
      pongEngine_paddleMove(1, DOWN,  speed);
      //pongEngine_destroyBall(); // Partially repairs artifacts - constrains them to path of ball
    }
    else if (key == _S){
      pongEngine_paddleMove(1, UP,  speed);
      //pongEngine_destroyBall();
    } else if (key == _ESC){ // Escape
      setInputMode(MENUS);
    }
  }

  if (mode == GAME){ // 2P only
    if (key == _UP){
      pongEngine_paddleMove(2, DOWN,  speed);
      //pongEngine_destroyBall();
    }
    else if (key == _DOWN){
      pongEngine_paddleMove(2, UP,  speed);
      //pongEngine_destroyBall();
    }
  }

//menuMove(unsigned int direction)
  if (mode == MENUS){
    if (speed == keyBSpeed){
      if (key == _UP){
        menuMove(_UP);
      } else if (key == _DOWN){
        menuMove(_DOWN);
      } else if (key == _LEFT){
        menuMove(_LEFT);
      } else if (key == _RIGHT){
        menuMove(_RIGHT);
      } else if (key == _RETURN){
        setMenu(4, 1); menuMove(0);
      }
    } else {
      if (key == _UP){
        menuMove(_RIGHT);
      } else if (key == _DOWN){
        menuMove(_LEFT);
      } else if (key == _W){
        menuMove(_UP);
      } else if (key == _S){
        menuMove(_DOWN);
      }
```

```
    }
  }


  if (key == _BKSPACE){ // Never reached
    toggleSound();
  }
  ResetWDT();
}

void enableInputs(int enable){
  if (enable == 1){
    // Enable interrupts
    KEY_ptr[2] = 0xF;      // Set key interrupts
    //*PS2_CONTROL |= 0x1;  // Set interrupt on PS2
    *PS2_CONTROL = (1<<0);
  } else {
    // Disable interrupts
    KEY_ptr[2] = 0x00;     // Set key interrupts
    //*PS2_CONTROL &= 0x0;  // Set interrupt on PS2
    *PS2_CONTROL = (0<<0);
  }
}

void setInputMode(unsigned int _mode){
  //if (_mode == MENUS) { mode = MENUS; } else if (_mode == GAME){ mode = GAME; } else if (_mode == GAME_AI){ mode = GAME; }
  mode = _mode;
  if (mode == MENUS){ Displays_mode(NOFRAMEBUFFER); } else { Displays_mode(SOFTWAREOCTOFB); }
}

int getInputMode( void ){
  return mode;
}
```

9.14 Appendix N - pongInputs.h

```c
/* To set function running, simply include PS2_Keyboard.h and then call inputsInitialise */
/* Requires scatter file change to an IRQ ready file */
// http://www-ug.eecg.utoronto.ca/msl/nios_devices_SoC/datasheets/PS2%20Keyboard%20Protocol.htm

#ifndef PONGINPUTS_PONGINPUTS_H_
#define PONGINPUTS_PONGINPUTS_H_

#include <stdbool.h>
#include "../pongEngine/pongEngine.h"
#include "../pongDisplay/pongDisplay.h"
#include "../pongSound/pongSound.h"
#include "../pongScreens/pongScreens.h"
#include <string.h>
#include "../HPS_IRQ/HPS_IRQ.h"
#include "../HPS_Watchdog/HPS_Watchdog.h"
#include <stdio.h>
#include <stdlib.h>


// Keyboard scancodes for scan set 2
// From https://techdocs.altium.com/display/FPGA/PS2+Keyboard+Scan+Codes
#define _ESC     0x76
#define _RETURN   0x5A
#define _BKSPACE  0x66
#define _W       0x1D
#define _A       0x1C
#define _S       0x1B
#define _D       0x23
#define _LEFT    0x6B
#define _RIGHT    0x74
#define _UP      0x75
#define _DOWN    0x72

// Definitions for modes
#define MENUS     0
#define GAME      1
#define GAME_AI   2

// Set paddle speeds
#define keySpeed  20
#define keyBSpeed 10


/* These are all pretty much used exclusively by this library. Interrupts handle everything and pass to InputControl. */

// Initialise the inputs
void inputsInitialise(void);

// Get raw keyboard scan codes, including make/break codes
char PS2Scan(void);

// Convert keyboard input to usable inputs and pass to Input()
void inputKeyboard( void );

// Clears FIFO on PS/2
void emptyFIFO( void );

// Takes input from keyboard/keys and acts on it depending on screen mode
void Input(unsigned int key, unsigned int speed);

// Set screen mode. Add switching between menus?
void setInputMode(unsigned int _mode);

// Get screen mode
int getInputMode( void );

// Enable/Disable inputs
void enableInputs(int enable);

#endif /* PONGINPUTS_PONGINPUTS_H_ */
```

9.15 Appendix O - pongSounds.c

```c
/*
 * pongSound.c
 *
 *  Created on: 20 Apr 2019
 *      Author: Sam
 */

#include "pongSound.h"
#include "notes.h"


volatile unsigned int SoundOn = 1;

// Variables
volatile int sound = 1;
int freq = 1000;
float duration = 1;
volatile unsigned VOLUME = 4;

//Pointers
volatile unsigned char* fifospace_ptr;
volatile unsigned int*  audio_left_ptr;
volatile unsigned int*  audio_right_ptr;
volatile unsigned int* HPS_timer_ptr = (unsigned int *) 0xFFC08000;


//Variables
//Phase Accumulator
double phase = 0.0;  // Phase accumulator
double inc   = 0.0;  // Phase increment
double baseampl  = 0.0;  // Tone amplitude (i.e. volume)
signed int audio_sample = 0;


void timerISR(HPSIRQSource interruptID, bool isInit, void* initParams) {
    if (!isInit) {
        volatile unsigned int * HPS_timer_ptr = (unsigned int *) 0xFFC08000;
        //Clear the Timer Interrupt Flag
        //By reading timer end of interrupt register
        HPS_timer_ptr[3];
        // Turn sound off
        sound = 0;
    }
    //Reset watchdog.
    HPS_ResetWatchdog();
}



void pongSound_Init(){
  //Initialise the Audio Codec.
  WM8731_initialise(0xFF203040);
}

void Sound(int _freq, float _duration){
  //Duration in milliseconds
  if (SoundOn){
    freq = _freq; duration = _duration;
    sound = 1;


    HPS_timer_ptr[2] = 0; // write to control register to stop timer
    // Set the timer period
    HPS_timer_ptr[0] = duration * 100000; // period = 1/(100 MHz) x (100 x 10^6) = 1 sec
    // Write to control register to start timer, with interrupts
    HPS_timer_ptr[2] = 0x03; // mode = 1, enable = 1
    // Register interrupt handler for timer
    HPS_IRQ_registerHandler(IRQ_TIMER_L4SP_0, timerISR);
    HPS_ResetWatchdog();

    //Clear both FIFOs
    WM8731_clearFIFO(true,true);
    //Grab the FIFO Space and Audio Channel Pointers
    fifospace_ptr = WM8731_getFIFOSpacePtr();
    audio_left_ptr = WM8731_getLeftFIFOPtr();
    audio_right_ptr = WM8731_getRightFIFOPtr();
    //Initialise Phase Accumulator
    inc   = ((float) freq) * 360.0 / F_SAMPLE;  // Calculate the phase increment based on desired frequency - e.g. 440Hz
    baseampl  = 8388608.0;              // Pick desired amplitude (e.g. 2^23). WARNING: If too high = deafening!
    phase = 0.0;
    // Primary function while loop
    while(sound == 1){
      //Always check the FIFO Space before writing or reading left/right channel pointers
      if ((fifospace_ptr[2] > 0) && (fifospace_ptr[3] > 0)) {
        //If there is space in the write FIFO for both channels:
        //Increment the phase
        phase = phase + inc;

        //Ensure phase is wrapped to range 0 to 360* (range of lookupSin function)
        while (phase >= 360.0) {
          phase = phase - 360.0;
        }
        // Calculate next sample of the output tone.
        audio_sample = (signed int)( (1<<VOLUME)*baseampl * lookupSin((int) phase) );
```

```
95
96          // Output tone to left and right channels.
97          *audio_left_ptr = audio_sample;
98          *audio_right_ptr = audio_sample;
99        }
100       ResetWDT();
101     }
102     // Disable timer ISR
103     HPS_timer_ptr[2] = 0x02; // mode = 1, enable = 0
104     //Finally reset the watchdog.
105     HPS_ResetWatchdog();
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
141 float lookupSin(unsigned int degree){
142   // Angles found with matlab, x10^5 and rounded
143   static unsigned int lookup[91] = { 0, 1745, 3490, 5234, 6976, 8716, 10453, 12187, 13917,
144             15643, 17365, 19081, 20791, 22495, 24192, 25882, 27564,
145             29237, 30902, 32557, 34202, 35837, 37461, 39073, 40674,
146             42262, 43837, 45399, 46947, 48481, 50000, 51504, 52992,
147             54464, 55919, 57358, 58779, 60182, 61566, 62932, 64279,
148             65606, 66913, 68200, 69466, 70711, 71934, 73135, 74314,
149             75471, 76604, 77715, 78801, 79864, 80902, 81915, 82904,
150             83867, 84805, 85717, 86603, 87462, 88295, 89101, 89879,
151             90631, 91355, 92050, 92718, 93358, 93969, 94552, 95106,
152             95630, 96126, 96593, 97030, 97437, 97815, 98163, 98481,
153             98769, 99027, 99255, 99452, 99619, 99756, 99863, 99939,
154             99985, 100000 };
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   } else return 0;
167 }
```

9.16 Appendix P - pongSounds.h

```
/*
 * pongSound.h
 *
 *  Created on: 20 Apr 2019
 *      Author: Sam
 */

/*      Warning - sounds are BLOCKING     */

#include "../DE1SoC_WM8731/DE1SoC_WM8731.h"
#include "../HPS_Watchdog/HPS_Watchdog.h"
#include "../HPS_IRQ/HPS_IRQ.h"

#ifndef _PONGSOUND_PONGSOUND_H
#define _PONGSOUND_PONGSOUND_H

//Define some useful constants
#define F_SAMPLE 48000.0        // Sampling rate of WM8731 Codec (Do not change)

// Initialise driver
void pongSound_Init( void );

// Play a sound of frequency _freq Hz for _duration milliseconds
void Sound(int _freq, float _duration);

// Enable (1) or disable (0) sounds
void enableSound(unsigned int _onoff);

// Set volume, 1-9
void setVolume(unsigned int _volume);

// Toggle sound
void toggleSound( void );

// Fast sin implementation using lookup table(0-90*), characteristics of sin and fixed point(ish) storage
float lookupSin(unsigned int degree);

// Should probably add volume control too


// Custom sounds here
void startSound( void );

void paddleBeep( void );

void ballOutBeep( void );


#endif /* PONGSOUND_PONGSOUND_H */
```

9.17 Appendix Q - pongScreens.c

```
/*
 * pongScreens.c
 *
 *  Created on: 20 Apr 2019
 *      Author: Sam
 */

#include "pongScreens.h"
//#include <string.h>

volatile unsigned int menuSelector = 0;
volatile unsigned int settings[] = {0,5,4,0,0};
volatile unsigned int menuSelectorOld = 0;
volatile unsigned int settingsOld[] = {0,5,4,0,0};
short menuColours[] = {_BLUE, _BLACK, _BLACK, _BLACK, _BLACK};

unsigned int settingsMax[] = {1, 9, 9, 1, 1};

unsigned int numMenuItems = sizeof(settings)/sizeof(int);
volatile unsigned int gameMode    = GAME_AI;
volatile unsigned int gameModeOld   = GAME_AI;
int n = 0;
volatile unsigned int *slider_ptr = (unsigned int *)0xFF200040;
unsigned int last_slider;

void setMenu(unsigned int _menuSelector, unsigned int _setting){
  menuSelectorOld = menuSelector;
  settingsOld[_menuSelector] = settings[_menuSelector];

  menuSelector = _menuSelector;
  settings[_menuSelector] = _setting;
}

void menuMove(unsigned int direction){
  unsigned int i;
  char str_txt[30];
  enableInputs(0);

  // Save old settings and reset menu colours
  for (i = 0; i<numMenuItems; i++){
    settingsOld[i] = settings[i];
    menuColours[i] = _BLACK;
  }

  menuSelectorOld = menuSelector;

  if (direction == _DOWN){
    if (menuSelector == 4){ menuSelector = 0; } else menuSelector++;
    //menuSelector++;
  } else if (direction == _UP){
    if (menuSelector == 0){ menuSelector = 4; } else menuSelector--;
    //menuSelector--;
  } else if (direction == _LEFT){
    if (settings[menuSelector] == 0) { settings[menuSelector] = settingsMax[menuSelector]; } else settings[menuSelector]--;
    //settings[menuSelector]--;
  } else if (direction == _RIGHT){
    if (settings[menuSelector] == settingsMax[menuSelector]) { settings[menuSelector] = 0; } else settings[menuSelector]++;
    //settings[menuSelector]++;
  }
  //printf("%d %d\n", menuSelector, settings[menuSelector]);// For debug only


  /* Set menu item colours */
  if (menuSelector != menuSelectorOld){
    menuColours[menuSelector] = _BLUE;
    pongSprites_renderBall(50, (71+(int)25*menuSelectorOld), _WHITE); ResetWDT(); // Get rid of old ball
    pongSprites_renderBall(50, (71+(int)25*menuSelector), _BLACK); ResetWDT();    // Replace
    // Menu items
    pongSprites_writeText(100, 20,  LARGE, "Main menu", _RED); ResetWDT();
    pongSprites_writeText(75, 65,   LARGE, "Mode: ", menuColours[0]); ResetWDT();
    pongSprites_writeText(75, 90,   LARGE, "Paddle size: ", menuColours[1]); ResetWDT();
    pongSprites_writeText(75, 115,  LARGE, "Volume: ", menuColours[2]); ResetWDT();
    pongSprites_writeText(75, 140,  LARGE, "Reset score > ", menuColours[3]); ResetWDT();
    pongSprites_writeText(75, 165,  LARGE, "START >", menuColours[4]); ResetWDT();
  }

  /* Menu effects */
  if (menuSelector == 0){ // Game mode
    if (settings[0] == 1){       // If press right on Reset
      pongSprites_writeText(240, 65, 1, "<AI>", _WHITE); ResetWDT();
      pongSprites_writeText(240, 65, 1, "<2P>", _MAGENTA); ResetWDT();
      gameMode = GAME;
    } else {
      pongSprites_writeText(240, 65, 1, "<2P>", _WHITE); ResetWDT();
      pongSprites_writeText(240, 65, 1, "<AI>", _MAGENTA); ResetWDT();
      gameMode = GAME_AI;
    }
  }

  if ((menuSelector == 1) && (settings[menuSelector] != settingsOld[menuSelector])){ // Paddle size
    sprintf(str_txt, "<%d>", settingsOld[1]);
    pongSprites_writeText(242, 90, LARGE, str_txt, _WHITE); ResetWDT();
    sprintf(str_txt, "<%d>", settings[1]);
    pongSprites_writeText(242, 90, LARGE, str_txt, _MAGENTA); ResetWDT();
```

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

```
    pongSprites_writeText(240, 65, LARGE, "<AI>", _MAGENTA); ResetWDT();
  } else {
    pongSprites_writeText(240, 65, LARGE, "<2P>", _MAGENTA); ResetWDT();
  }

  sprintf(str_txt, "<%d>", settings[1]); // Paddlesize options
  pongSprites_writeText(242, 90, LARGE, str_txt, _MAGENTA); ResetWDT();

  sprintf(str_txt, "<%d>", settings[2]); // Volume options
  pongSprites_writeText(242, 115, LARGE, str_txt, _MAGENTA); ResetWDT();

  ResetWDT();
  //Displays_forceRefresh(); // Hmm that frame buffer's being odd again
  Displays_Refresh();
  enableInputs(1);
  while(getInputMode() == MENUS){
    /* Most of this can probably be pushed to menuMove */

    if ((menuSelector != menuSelectorOld) || (settings[menuSelector] != settingsOld[menuSelector])){
      // As sounds are blocking, and rely on a timer ISR, they need to be called outside menuMove,
      // as it is essentially an extension of the input ISRs
      if (settings[2] != settingsOld[2]){
        Sound(G4, 50);
      }

      //Displays_forceRefresh();

      menuSelectorOld = menuSelector;
      for (i = 0; i<numMenuItems; i++){
        settingsOld[i] = settings[i];
        menuColours[i] = _BLACK;
      }
    }
  ResetWDT();
  }
  last_slider = *slider_ptr;
  Displays_clearScreen();
}

void testScreen_AI( void ){
  int dir;
  int vel;
  int *arr;


  // Clear screen and set input mode
  Displays_clearScreen();
  setInputMode(GAME_AI);

  // Initialise engine
  pongEngine_init();
  ResetWDT();

  pongEngine_createBall();
  //pongSprites_writeText(96, 60, 1, "AI MODE", 0xFFFF);

  Displays_forceRefresh(); pongEngine_refreshScore();
  n = 0;
  while (getInputMode() == GAME_AI){
    ResetWDT();
    Displays_forceRefresh();

    if (n < 1) {

         n++;
         dir = pongPhysics_serve ();
         vel = 2;
      }

    if ((pongEngine_getBallLocation_x() <= 50 + 15) && (pongEngine_getBallLocation_x() >= 50 + 10))   {

      if ((pongEngine_getBallLocation_y() <= pongEngine_getPaddleY (1) + 30 + 15) && (pongEngine_getBallLocation_y() >=
     pongEngine_getPaddleY (1) - 30 - 15)) {

      paddleBeep();
        arr = pongPhysics_paddleCollision(vel,dir,1);
        dir = arr[0];
        vel = arr[1];


      }

    } else if ((pongEngine_getBallLocation_x() >= 270 - 15) && (pongEngine_getBallLocation_x() <= 270 - 10))  { // -10

      if ((pongEngine_getBallLocation_y() <= pongEngine_getPaddleY (2) + 30 + 15) && (pongEngine_getBallLocation_y() >=
     pongEngine_getPaddleY (2) - 30 - 15))  {

      paddleBeep();
      arr = pongPhysics_paddleCollision(vel,dir,2);
      dir = arr[0];
      vel = arr[1];


      }
    }

    //collisions
    if ( pongEngine_getBallLocation_y() <= 23 + 20  ) {
```

```
      enableInputs(0);
      paddleBeep();
      enableInputs(1);

      dir = pongPhysics_borderCollision(vel, dir);


    } else if ( pongEngine_getBallLocation_y() >= 230 - 10){
      enableInputs(0);
      ballOutBeep();
      enableInputs(1);

      dir = pongPhysics_borderCollision(vel, dir);
    }



    // AI bit
    if ( pongEngine_getBallLocation_y() > pongEngine_getPaddleY(2)+10){

      pongEngine_paddleMove(2, UP, 2);

    } else if ( pongEngine_getBallLocation_y() < pongEngine_getPaddleY(2)-10){

      pongEngine_paddleMove(2, DOWN,  2);
    }

    pongEngine_moveBall(dir, vel);

    if (*slider_ptr != last_slider) {

      setInputMode(MENUS);
      last_slider = *slider_ptr;
    }
    //This function adds points
    if ( pongEngine_getBallLocation_x() <= 10  ) { //+20
        pongEngine_destroyBall(); //destroy ball
        Displays_forceRefresh(); //force refresh for buffers prior to points
        enableInputs(0); //disable inputs
        paddleBeep(); //play sound
        enableInputs(1); //enable inputs
        pongEngine_resetPaddles(); //reset paddles
        pongEngine_resetBallLoc(); //reset ball location
        pongEngine_createBall(); //create ball
        pongEngine_paddleCreate(1); //create paddle
        pongEngine_paddleCreate(2); //create paddle
        pongEngine_moveBall(1, 0); //reset angle instructions
        pongEngine_moveBall(0, 0);
        Displays_forceRefresh(); pongEngine_refreshScore(); //force refresh
        n = 0; //reset var
        pongEngine_addPoint(2); //add point
    } else if ( pongEngine_getBallLocation_x() >= 320 - 10){
      pongEngine_destroyBall(); //destroy ball
      enableInputs(0); //disable inputs
      paddleBeep(); //play sound
      enableInputs(1); //enable inputs
      pongEngine_resetPaddles(); //reset paddles
      pongEngine_resetBallLoc(); //reset ball location
      pongEngine_createBall(); //create ball
      pongEngine_paddleCreate(1); //create paddle
      pongEngine_paddleCreate(2); //create paddle
      pongEngine_moveBall(1, 0); //reset angle instructions
      pongEngine_moveBall(0, 0);
      Displays_forceRefresh(); pongEngine_refreshScore(); //force refresh
      n = 0; //reset var
      pongEngine_addPoint(1); //add point


    }
  }

  //gameEngine_paddleDestroy(1);
  //gameEngine_paddleDestroy(2);
  Displays_clearScreen();
}

void testScreen( void ){
  int dir = 0;
  int vel;
  int* arr;

  // Clear screen and set input mode
  Displays_clearScreen();
  setInputMode(GAME);

  // Initialise engine
  pongEngine_init();
  ResetWDT();

  pongEngine_createBall();
  //pongSprites_writeText(96, 60, 1, "2P MODE", 0xFFFF);
  Displays_forceRefresh(); pongEngine_refreshScore();
  n = 0;
  while (getInputMode() == GAME){
    ResetWDT();
    Displays_forceRefresh();

```

```
    if (n < 1) {
          n++;
          dir = pongPhysics_serve ();
          vel = 3;
      }

    if ((pongEngine_getBallLocation_x() <= 50 + 15) && (pongEngine_getBallLocation_x() >= 50 + 10))   {

      if ((pongEngine_getBallLocation_y() <= pongEngine_getPaddleY (1) + 30) && (pongEngine_getBallLocation_y() >=
     pongEngine_getPaddleY (1) - 30)) {

      paddleBeep();
        arr = pongPhysics_paddleCollision(vel,dir,1);
        dir = arr[0];
        vel = arr[1];

      }
    } else if ((pongEngine_getBallLocation_x() >= 270 - 15) && (pongEngine_getBallLocation_x() <= 270 - 10))  { // -10

      if ((pongEngine_getBallLocation_y() <= pongEngine_getPaddleY (2) + 30) && (pongEngine_getBallLocation_y() >=
     pongEngine_getPaddleY (2) - 30))  {

      paddleBeep();
      arr = pongPhysics_paddleCollision(vel,dir,2);
      dir = arr[0];
      vel = arr[1];

      }
    }

    //collisions
    if ( pongEngine_getBallLocation_y() <= 23 + 20  ) {
      enableInputs(0);
      paddleBeep();
      enableInputs(1);

      dir = pongPhysics_borderCollision(vel, dir);


    } else if ( pongEngine_getBallLocation_y() >= 230 - 10){
      enableInputs(0);
      ballOutBeep();
      enableInputs(1);

      dir = pongPhysics_borderCollision(vel, dir);
    }

    //This function adds points
    if ( pongEngine_getBallLocation_x() <= 10  ) { //+20
      pongEngine_destroyBall(); //destroy ball
      Displays_forceRefresh(); //force refresh for buffers prior to points
      enableInputs(0); //disable inputs
      paddleBeep(); //play sound
      enableInputs(1); //enable inputs
      pongEngine_resetPaddles(); //reset paddles
      pongEngine_resetBallLoc(); //reset ball location
      pongEngine_createBall(); //create ball
      pongEngine_paddleCreate(1); //create paddle
      pongEngine_paddleCreate(2); //create paddle
      pongEngine_moveBall(1, 0); //reset angle instructions
      pongEngine_moveBall(0, 0);
      Displays_forceRefresh(); pongEngine_refreshScore(); //force refresh
      n = 0; //reset var
      pongEngine_addPoint(2); //add point
    } else if ( pongEngine_getBallLocation_x() >= 320 - 10){
      pongEngine_destroyBall(); //destroy ball
      enableInputs(0); //disable inputs
      paddleBeep(); //play sound
      enableInputs(1); //enable inputs
      pongEngine_resetPaddles(); //reset paddles
      pongEngine_resetBallLoc(); //reset ball location
      pongEngine_createBall(); //create ball
      pongEngine_paddleCreate(1); //create paddle
      pongEngine_paddleCreate(2); //create paddle
      pongEngine_moveBall(1, 0); //reset angle instructions
      pongEngine_moveBall(0, 0);
      Displays_forceRefresh(); pongEngine_refreshScore(); //force refresh
      n = 0; //reset var
      pongEngine_addPoint(1); //add point
    }

    pongEngine_moveBall(dir, vel);

    if (*slider_ptr != last_slider){
      setInputMode(MENUS);
      last_slider = *slider_ptr;
    }
  }
  //gameEngine_paddleDestroy(1);
  //gameEngine_paddleDestroy(2);
  Displays_clearScreen();
}
```

9.18 Appendix R - pongScreens.h

```
/*
 * pongScreens.h
 *
 *  Created on: 20 Apr 2019
 *      Author: Sam
 */

#ifndef PONGSCREENS_H_
#define PONGSCREENS_H_

#include "../pongInputs/pongInputs.h"
#include "../pongSound/pongSound.h"
#include "../pongSound/notes.h"
#include "../pongEngine/pongSprites.h"
#include "../pongDisplay/pongDisplay.h"
#include "../DE1SoC_SevenSeg/sevenSeg.h"
#include "../HPS_Watchdog/HPS_Watchdog.h"
#include "../HPS_usleep/HPS_usleep.h"
#include "../pongEngine/pongPhysics.h"
#include <stdio.h>
#include <stdbool.h>

#define _BLACK   (0x0000)
#define _WHITE   (0xFFFF)
#define _RED     (0xF800)
#define _GREEN   (0x07C0)
#define _BLUE    (0x001F)
#define _YELLOW  (_RED | _GREEN)
#define _CYAN    (_GREEN | _BLUE)
#define _MAGENTA (_BLUE | _RED)

void menuMove(unsigned int direction);

void setMenu(unsigned int _menuSelector, unsigned int _setting);

void startScreen( void );

void testScreen( void );

void testScreen_AI( void );

void gameMenu( void );

#endif /* PONGSCREENS_H_ */
```

9.19 Appendix S - main.c

```
#include "DE1Soc_VGA/DE1SoC_VGA.h"
#include "pongDisplay/pongDisplay.h"
#include "pongEngine/pongSprites.h"
#include "pongEngine/pongEngine.h"
#include "pongScreens/pongScreens.h"
#include "pongInputs/pongInputs.h"
#include "pongSound/pongSound.h"
#include "HPS_Watchdog/HPS_Watchdog.h"


int main(void) {
  int FS = 3;

  // Initialise displays
  Displays_init(0xC8000000,0xC9000000,0xFF200060,0xFF200080);
  Displays_frameSkip(FS);
  ResetWDT();

  // Initialise interrupts
  HPS_IRQ_initialise(NULL);
  HPS_ResetWatchdog();

    // Initialise keyboard/pushbuttons
  inputsInitialise();

  // Initialise sounds
  pongSound_Init();

  // Init ball for menu
  pongSprites_initBall();

  // Run loading screen
  startScreen();

  // Turn on inputs
  enableInputs(1);
  while(1){
    if (getInputMode() == MENUS){
      // Run start menu
      gameMenu();
    } else if (getInputMode() == GAME){
      // Run 2P test screen
      testScreen();
    } else if (getInputMode() == GAME_AI){
      // Run test screen
      testScreen_AI();
    }
  }
}
```

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