UNIVERSITY OF LEEDS

ELEC5620M
Mini Project

DE1-SoC Pong

Alexander Bolton - 200938078
Sam Wilcock - 201285260
John Jakobsen -

May 2019

Submitted in accordance with the requirements for the degree of
Master of Science in Embedded Systems Engineering

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 member's 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...

Section 4 will discuss...

Section 5 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.

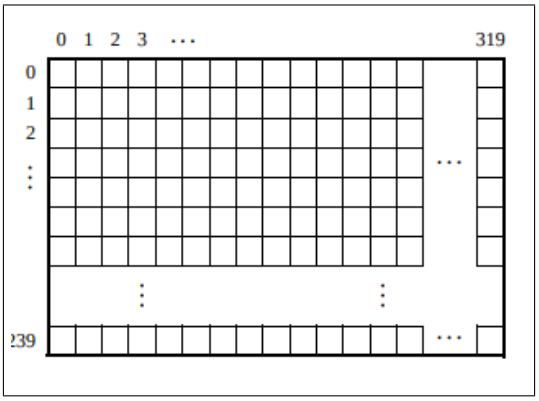

Figure 1: Pixel layout for pixel buffer of VGA controller [1]

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.

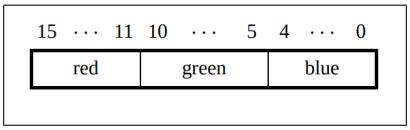

Figure 2: Pixel Colour Layout [1]

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

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

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

2.2 Display Driver

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

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

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

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

```
1  void DisplaysLocal_quadRefresh(){
2    int x;
3    int y;
4    int q1change = memcmp(frontFrameBuffer1,rearFrameBuffer1, sizeof(frontFrameBuffer1));
5    int q2change = memcmp(frontFrameBuffer2,rearFrameBuffer2, sizeof(frontFrameBuffer2));
6    int q3change = memcmp(frontFrameBuffer3,rearFrameBuffer3, sizeof(frontFrameBuffer3));
7    int q4change = memcmp(frontFrameBuffer4,rearFrameBuffer4, sizeof(frontFrameBuffer4));
8    for(y = 0; y < PixelHeight/2; y++){
9        for(x = 0; x < PixelWidth/2; x++){
10         if(q1change != 0){
11           if(frontFrameBuffer1[x][y] != rearFrameBuffer1[x][y]){
12             Displays_drawPixel(x,y,frontFrameBuffer1[x][y]);
13             rearFrameBuffer1[x][y] = frontFrameBuffer1[x][y];
```

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.

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

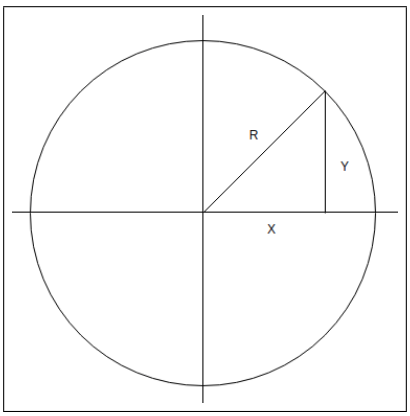

Figure 5: Pythagoras Theorem demonstrated inside circle

The paddle was set by iterating through a square of defined values and setting pixels to the colour desired by iterating through the height and width of the rectangle and setting values to the desired colour.

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.

```
int pongEngine_calcAngle(int x1, int y1, int x2, int y2){
  double diff_y = y1 - y2;
  double diff_x = x1 - x2;
  double angle = atan2(diff_y, diff_x);
  float ang_d = angle * 180/PI;
  int ang_dint = ang_d;
  return ang_dint;
}
```

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 an char array exit character was missing.

3 Controls and Menus

4 Game Physics

5 Problems Encountered

6 Conclusion

7 Appendix

References

[1] Altera, *DE1-SoC Computer System with Nios II*. Altera, 2014.