ECE 4760

Lab 3: TFT-LCD video game

Author:	*NetID:*
Jonah Okike-Hephzibah	jo356
Sebastian Roubert	sr949
Rahul Desai	rd542

Section: Thursday 4:30pm
Performed on October 12th, 2017

Contents

List of Figures

1 Introduction

The purpose of this lab was to create a working TFT-LCD video game that was easy to use. The game consisted of multiple balls being inserted onto screen, two barriers opposite where the balls entered, and a paddle whose horizontal position could be controller by the player via a potentiometer. To get a point, the user must bounce a ball off the paddle to the back-side of a barrier. To lose a point, the ball must pass the paddle and hit the wall behind it. Losing a point while having zero points results in the user losing the game.

The game operates at 15 frames per second, although this could be easily alter in the code to run faster or slower. There is drag on the balls to give a more realistic view of balls on a table. Three different sounds are played from direct memory access (DMA) whenever you a get a point, lose a point, or lose the game, respectively.

The objective of the lab was to have at least 30 balls on the screen. We far surpassed that. During the demo we achieved 165 balls and during testing, we achieved 200+ balls.

We are confident we made a game that is easy to learn, semi-fun to play, and absolutely impossible to not lose.

2 Design and Testing

2.1 Software

2.1.1 Balls Instantiation

Each ball in our system had certain properties associated with playing. To address this, we used a struct to define each ball with the following

```
// The structure that defines the properties of a ball.
struct balls {
    fix16 x; // The x position
    fix16 y; // The y position
    fix16 vx; // The velocity in the x direction
    fix16 vy; // The velocity in the y direction
    int valid; // Flag to determine whether the ball is on screem being animated
    int hitCounter; // The balls hit counter to determine if it can participate in a collision
};
```

Each of these properties were used throughout our code. From there, we predefined an array of the balls struct to call so that we could have up to 300 balls as seen below

```
int numBalls=300; // The number of balls that are being created
static struct balls bArray[300]; // The array of balls both valid and invalid
```

2.1.2 Drawing

The predefined tft draw function (tft_drawCircle) took a considerable amount of time to draw one ball due to the optimization it runs to fill in the circle, we decided to create our own to save time in the animation thread. To do this, we used the balls center position ((x_0,y_0) in the ball structure) and then drew pixels using tft_drawPixel at the following locations:

(x_0, y_0 + r)
(x_0, y_0 - r)
(x_0 + r, y_0)
(x_0 - r, y_0)

Here, r is the radius of the ball which is equal to 2. In the code, this function is named "drawCircle" and takes in as parameters a pointer to the ball struct as well as the color of the ball.

2.1.3 Animation

The animation protothread consisted of the collision code, barrier drawing, paddle drawing and ball drawing. Due to the number of operations that were ran in this thread, this ended up being the heaviest section of code in the overall system. This section will focus primarily on the logic for adding balls onto the screen.

In the beginning of the animation protothread, a for loop would fill in the previously defined ball structure with the necessary parameters for each ball. The loop was made such that only the first two balls were valid in the beginning of the game while the rest were made invalid. We set the number of balls (numBalls) equal to 300 although we did not simulate this many balls at once. This will be further discussed in the results section.

```
for (i = 0; i < numBalls; i++) {

// Start with 2 valid balls.
if (i <= 1) {
bArray[i].valid = 1 ;
bArray[i].x = int2fix16(180 - 5*i) ;
bArray[i].y = int2fix16(30  + 8*i) ;
bArray[i].vx = int2fix16(4) ;
bArray[i].vy = int2fix16(3) ;
```

```
bArray[i].hitCounter = 0 ;
} else {
// The rest of the balls will be invalid with a position outside of
// the valid playing area
bArray[i].valid = 0 ;
bArray[i].x = int2fix16(0) ;
        bArray[i].y = int2fix16(0) ;
        bArray[i].vx = int2fix16( -1*(i%3 + 1) ) ;
        bArray[i].vy = float2fix16( -1*1.5 ) ;
        bArray[i].hitCounter = 0 ;
}
}
```

The position of the two valid balls were also staggered such that the first ball had a position of (180,30) and the second (175,38). The remaining valid balls were positioned off of the screen at (0,0) with initial velocities in the range of -1 to -3 in the x and -1.5 in the y. These velocities are not used as the actual velocity of each ball is set when they become valid.

Since we did not want any of the animation code to run unless the game was concurrently running, we placed that would not allow the main body of the code to run unless the game begin counter had passed 3 seconds and the game end counter had not passed 30 seconds. The overall use of these two counters will be discussed in the print thread section. Once these two conditions were satisfied, the two barriers of length 60 pixels would be drawn onto the screen at locations (0,220) and (175,220) (Note: these coordinates signify the leftmost point of the barrier). Two do this, we made use of the tft_drawLine function:

```
tft_drawLine(0, 220, 60,220 ,ILI9340_GREEN);
tft_drawLine(235, 220, 175, 220,ILI9340_GREEN);
```

We would then zero the counter for the number of balls on the screen so that the value could be updated in that frame which transitioned us to the for loop that would iterate through every ball in the balls structure array.

```
for (i = 0; i<numBalls; i++){
.
    .
    .
}
```

The beginning of the for loop would check to see if the current ball met the specification to become invalid. For the ball to not be valid, the velocity in both the x and y directions needed to be less than or equal

to 0.5. If the ball did not meet this specification, then we would then check the i^{th} balls validity, skipping collision calculations for this ball if it was not valid and proceeding with the collision calculations if it was. By skipping the calculations, the code would automatically check to see if it was legal to animate that ball. The conditions for doing this were as follows:

1. The last time a ball was added onto the display was two frames prior to the current frame.
2. The number of balls that have been added to the display in the current frame had not exceeded the pre-defined amount (3).

If these conditions held true, then the balls valid parameter would be set to true, the position would be set to (150,7) (the place we decided to have the balls shoot into the system), and the velocity set to a value between -3 and 3 in the x direction and 3 and 8 for the y direction. The exact value of the balls velocity was entirely dependent on the balls position in the array as the exact calculation was as follows:

```
bArray[i].vx = float2fix16( -3 + i%13*0.5 ) ;
bArray[i].vy = float2fix16( i%5+3 ) ;
```

The counter that tracks the number of balls that have been added into the system in the current frame would then be incremented ensuring that the conditions will no longer be valid once the number of newly animated balls equaled 3. If the ball was valid, a second for loop would then be initiated starting with the next ball from the i^{th} ball to check for ball-to-ball collisions:

```
for (j = i+1; j < numBalls; j++){
.
.
.
}
```

This was done because all balls below i had already checked for collisions with ball i, reducing the amount of work needed for ball i. Once again, the code would check to see whether the j^{th} ball was valid and only proceed with the calculations if the ball was valid. If ball j made it past this step, the distance between the two balls centers (dx and dy) would be calculated to check the proximity of ball j to ball i. If the balls were less than 4 pixels apart in both the x and y directions, a second check was made to see if the squared magnitude of the distance was less than 16 and if ball i had a hitCounter value equal to zero:

```
(multfix16(dx,dx)+multfix16(dy,dy))<int2fix16(16) && bArray[i].hitCounter == 0
```

If these conditions were true, then the collision code would be called. If these conditions did not hold, then the balls may not have actually been close enough for a collision to happen or ball i may have already participated in a collision and still needed time to escape. In this case, the collision code would not be called and the loop would continue on. After this nested loop completed checking for collisions, the code would check the hitCounter of ball i and decrement it if the value was greater than zero. This whole process would be repeated for each ball in the range of 0 to numBalls (300), with the last ball concluding the ball-to-ball collision checks.

A second for loop would be initiated right after the completion of the ball-to-ball collision loop. This loop was used for redrawing the valid balls. To begin with, the loop would paint the current ball to black, ensuring that if it had just been made invalid, it would not stay animated on the tft display. The code would then check the validity of the ball, proceeding if the ball was valid and moving to the next ball if it was not. If the ball was valid, the code would start by adding drag to the balls velocity:

```
bArray[i].vx = bArray[i].vx - multfix16(bArray[i].vx,drag);
bArray[i].vy = bArray[i].vy - multfix16(bArray[i].vy,drag);
```

In the above code snippet, "drag" is a pre-defined constant equal to 0.01. The location of the ball would then be updated by adding this velocity to its current x and y position. This point marked the beginning of the 3 remaining collision types (ball-to-wall, ball-to-barrier and ball-to-paddle). The specifics of each of these will be discussed in the collision section of this report. The code would first check the ball-to-wall collisions, then ball-to-barrier. If the ball hit the barrier from the bottom side, the ball would be marked invalid and the scoreCounter would be incremented by one as well as the soundEffect flag set to 1. If the ball passed the paddle, it would also be marked invalid and the scoreCounter would be decremented by one and the soundEffect flag would be set to 2. The soundEffect flags use will be discussed later in the report. At the end of these checks, the validity of the ball would be checked once more to ensure that it was still legal to animate this ball. If the ball was valid, we called the drawCircle function to draw the ball at the new position. This process continues for each ball in the array.

Once this loop ends, the code moves on to animating the paddle. We decided to track the center of the paddle rather than an edge. Thus, we needed to find out what the max values of the ADC input were and relate them to the maximum and minimum values of the tft display. To do this, we had the tft display the value that it read in from the ADC and moved the trim. pot. to its minimum (all the way to the left) and maximum (all the way to the right) positions, enabling us to read the range. We found that the ADC values ranged from 0 to 519 and wanted to force the paddles center to range from 30 to 205 pixels. Thus, our conversion from the ADC value to a pixel position was as follows:

$$paddleCenter = 0.3372 * ADCValue + 30 \tag{1}$$

where 0.3372 is equal to $\frac{maxPixelValue-minPixelValue}{maxADCValue-minADCValue}$. Before finding the new paddle center, we would erase the current paddle by drawing over it in black. The above equation would then be calculated then the paddle would be redrawn with the calculations result as its center.

```
tft_drawLine((short)(paddleCenter-30), 295, (short)(paddleCenter+30),295 ,ILI9340_BLACK);
adcRead = (int) (0.3372 * (float) ReadADC10(0)) ;
paddleCenter = adcRead + 30 ;
tft_drawLine((short)(paddleCenter-30), 295, (short)(paddleCenter+30),295 ,ILI9340_GREEN);
AcquireADC10() ;
```

As seen in the above code snippet, the function "ReadADC10()" was called with zero as a parameter. This tells the ADC to read out the first value in the ADC. At the end of the code snippet, the function "AcquireADC10()" is called to allow the ADC to read the next value.

After all of these various calculations had occurred, the code would determine how much time it had taken to go through all of the various calculations then yield for 67 milliseconds - this time. The calculation of the elapsed time was done by reading a timer value at the beginning of the thread (begin_time) and subtracting this value from the time at which the process outlined above had been complete. This was done to have our display run at 15 frames per second (fps) since 1 frame runs in 0.0667 seconds ($\frac{1}{15} = 0.0667$).

```
PT_YIELD_TIME_msec(67 - (PT_GET_TIME() - begin_time)) ;
```

This concluded the code that rand if there was game play. If the game was over or had not started yet, the code would set soundEffect equal to 3 and delay for a millisecond. This was done primarily to allow other processes have access to the processor.

2.1.4 Collisions

There are three possible collisions to consider in our video: 1) ball to wall, 2) ball with barrier, 3) ball to paddle, and 4) ball to ball.

The first three cases of collisions are essentially the same. Whenever the distance of the ball's center to the wall/barrier/paddle was less than the ball's radius, we call that a collision. As soon as a collision was found, we updated the ball's position such that it was right on the edge of collision. Because the walls/barriers/paddle doesn't move from a collision with the ball, and we model this as a bounce with conservation of momentum and no loss of energy, we simply change the sign of the ball's velocity in the direction in which hit the wall/barrier/paddle. For example, if the ball was hitting the right side of the wall, then x-velocity would be switched from right to left as shown below:

```
// Check to see if the updated position puts the ball outside of
// the walls in the x direction
if (bArray[i].x<int2fix16(5) || bArray[i].x>int2fix16(235)){

    // If it is, set the position of the ball to be right
    // at the wall
    if (bArray[i].x<int2fix16(5) ) bArray[i].x=int2fix16(5);
    if (bArray[i].x>int2fix16(235) ) bArray[i].x=int2fix16(235);

    // Then flip the velocity of the ball
    bArray[i].vx = -bArray[i].vx ;
}
```

Similar code was written for all the other possibilities. We tested wall, paddle and barrier collisions with a single ball and see if the behavior followed the model.

The case of the ball-to-ball collision was a bit more complex.

We looped through each pair of balls on the screen, we dub these "valid balls." We then check if the distance between the center of the balls is less than twice the ball's radii. If it is, we then model this as a collision. If there is a collision, we calculate what the change in velocity is for each ball. The change in velocity in a collision with conservation of momentum is given by the following

$$\Delta\vec{v}_i = -\vec{r}_{ij}\left(\frac{\vec{r}_{ij}\cdot\vec{v}_{ij}}{\|\vec{r}_{ij}\|^2}\right) \tag{2}$$

where $\Delta\vec{v}_i$ is the change in velocity of the ball in question, $\vec{r}_{ij}$ is relative position of the ball in question to the ball it is colliding with, and $\vec{v}_{ij}$ is the relative velocity of the ball in question to the ball it is colliding with. We did all these calculations with fixed point arithmetic in a function called collisionCalculate that returned the change in velocity with the following

```
struct dv deltaV ; // This is the velocity update that the function will return

r01[0] = firstBall->x  - secondBall->x  ;
r01[1] = firstBall->y  - secondBall->y  ;

v01[0] = firstBall->vx  - secondBall->vx  ;
v01[1] = firstBall->vy  - secondBall->vy  ;

rDotV = multfix16(r01[0],v01[0]) + multfix16(r01[1],v01[1]);
```

```
deltaV.dvx = divfix16(multfix16(-r01[0],rDotV),int2fix16(16)) ;
deltaV.dvy = divfix16(multfix16(-r01[1],rDotV),int2fix16(16)) ;
```

From here, $\Delta\vec{v}_i$ was added to the velocity of the ball in question and subtracted from the ball with which the ball in question was colliding.

We tested this function first with two balls facing each other then expanded to multiple balls.

To account for times when the value of $\vec{r}_{ij}$ was too small such that $\vec{r}_{ij} \cdot \vec{v}_{ij}$ would be too small an approximation, i.e. modeling when the balls had a head-on collision, we just switched the velocities using the following

```
// If the value of rDotV was too small for the division step,
// we simply just swap the velocities of each ball
if ( rDotV < 4){
    deltaV.dvx = -v01[0];

    deltaV.dvy = -v01[1];

}
```

Notice how we picked the threshold value being that $\vec{r}_{ij} \cdot \vec{v}_{ij}$ was less than 4.
To make sure that balls did not get stuck in a collision, each ball had a parameter called hitCounter. Whenever a ball went through a collision, its hitCounter value would be set to 5. Whenever the hitCounter was greater than 0, it could not enter another collision. Upon each frame, the hitCounter would be decremented if it was less than 0. Essentially, this made sure that no ball would get stuck in a collision and that it would be able to collide again in a reasonable of frames.

2.1.5 Sound Playback

The sound playback consisted of setting up the DMA to transfer a table of sound values to the DAC. There were two tables in use for the sounds: 1. A sine table (table1) of size table_size=256 and 2. A sound file from the video game *Luigis Mansion* (AllDigits). In the game, the sound plays when the gold mouse appears. The sine table was constructed in the main body of the code (as previously done in lab 2) with the one exception being the period. Rather than creating a sine table with a period of 2π, we opted to use a period of 16π mainly because we preferred the tone that was generated using the DMA.

```
for (i = 0; i < table_size; i++){
        table1[i] = (short)(2047*sin((float)i*50.2655/(float)table_size));
```

```
        table1[i] = (short)( DAC_config_chan_A | (table1[i] + 2048) ) ;
}
```

The Luigis Mansion sound was processed from a wav file to a header file using MATLAB code that was provided by Bruce. However, we needed to edit part of the code because it was previously written for use with a 4-bit DAC while we made use of a 12-bit DAC. The changes we made were as follows:

1. Resampled the wav file at one-tenth the original sampling rate.
2. Changed the max value from being 16 (2^4) to 4096 (2^{12}).
3. Removed the byte packing since this is not needed for a 12-bit DAC.

Once these changes were made, we loaded the wav file into the MATLAB code which outputted a header which held an array of type *const unsigned char*. This proved to be troublesome since the values that were in the array were too big to fit into a char variable. Thus, we had to change the definition of the array to be of type short:

From: static const unsigned char AllDigits[]
To: static short AllDigits[]

From here, all that was left to do was mask the first 4 bits of the AllDigits array to hold the address of the DAC. This was done in the main function:

```
for( i = 0; i < sizeof(AllDigits); i++) {
        AllDigits[i] = (short) (DAC_config_chan_A | AllDigits[i]) ;
}
```

Note, this was also done for the sine table as can be seen in the second line of the sine table creation code snippet. In both cases, DAC_config_chan_A holds the address to the channel A DAC (0b0011000000000000) of which is defined at the beginning of the code.

The DMA was setup to run in auto mode so that once a transfer was initiated, it could only be stopped once a terminate command was sent. Since we had two separate sounds that we were using, we initially experimented with using one DMA to transmit both of these sounds to the DAC, but this did not end up working. For this reason, we employed the use of DMA channels 0 and 1 with channel zero have a 0 priority and channel 1 having a priority of 1.

```
DmaChnOpen(dmaChn, 0, DMA_OPEN_AUTO);
DmaChnOpen(1, 1, DMA_OPEN_AUTO);
```

(Note: dmaChn = 0). Once these DMA channels were opened, channel 0 was configured to send sine table data to the DAC (Note: the sine table entries had its first 4 bits masked to contain the DACA configuration bits). Channel 0 of the DMA was configured to send the audio file of which was also pre-configured in the same manner as the sine table.

```
DmaChnOpen(dmaChn, 0, DMA_OPEN_AUTO);
DmaChnOpen(1, 1, DMA_OPEN_AUTO);
```

As seen in the above code snippet, the DMA transferred the sound to SPI2BUF, the address of the SPI which had already been configured to operate in framed SPI mode. The last configuration that was made was to set the frequency at which the DMA would send data to the DAC. We employed the use of timer2 which we configured at various time scales for each transfer (44 KHz for decrementing sound from the sine table, 110 KHz for incrementing sound and 4.4 KHz for the end game sound). This was done by writing the following lines of code:

```
DmaChnSetEventControl(dmaChn, DMA_EV_START_IRQ(_TIMER_2_IRQ));
DmaChnSetEventControl(1, DMA_EV_START_IRQ(_TIMER_2_IRQ));
```

To verify that the sounds did indeed playback at the designated frequencies, we played each sound indefinitely and connected the output to an oscilloscope. The results of these tests will be discussed later in the report.

Once the DMA was configured, a protothread would constantly check to see if the soundEffect flag was set to any of the sound values:

soundEffect = 1: Play the sound corresponding to a sound increment.
soundEffect = 2: Play the sound corresponding to a sound decrement.
soundEffect = 3: Play the sound corresponding to the game countdown/ ending.

This flag was used so that a transfer would only occur if it is required to by the change in score.

To first start the sound transfer for the incrementing and decrementing sounds, the timer2 period would need to be rewritten. This was to ensure that if the the previously played and the current sound had conflicting timer periods, this would not affect the current sound playback. The sound transfer from the DAC would be initiated using the DmaChnEnable function with an input of the DMA channel. At this point, we set the timer back to the zero position ensuring that the transfer would start at the right time. The transfer would then be allowed to continue for 10 milliseconds before being terminated by the DmaChnAbortTxfer function, also with an input of the DMA channel. We would then set the soundEffect variable to zero to prevent the protothread from initiating another transfer.

```
WritePeriod2(400) ;
DmaChnEnable(dmaChn) ;
WriteTimer2(0) ;
PT_YIELD_TIME_msec(10) ;
DmaChnAbortTxfer(dmaChn);
soundEffect = 0 ;
```

Note: the above code snippet was used to play the score incrementing sound.

To test that the soundEffect logic actually worked, we hard set the value of soundEffect to be 1 and 2 so that no matter what event occurred in the game, only the sound corresponding to either 1 or 2 would play. For each case, the correct sound was played. To test if the logic for soundEffect = 3 worked, we set the logic back to the default case and allowed for the game to run till the end. In doing this, we expected that the end game sound should play and nothing else. This was the result that we saw.

For all of the above to work, we needed to set the SPI into framed SPI mode. To do this, we followed the configuration that was used by Tahmid and Bruce [1].

2.1.6 Parameter Display

Part of the requirements for this lab was to display the following values:

1. The score
2. Frame rate
3. Number of balls currently being animated
4. Elapsed game play time

To accomplish all of these tasks, we ran a protothread that would update these values every second. This thread also handled counting the elapsed time so updating these values every second essentially had a double purpose. On top of this, the thread also used a count down timer to give the user 3 seconds to prepare before the game started (figure 1).

Figure 1: Game Begin Countdown

This thread also cleared the display of the game field and displayed "Game Over" once the time had passed 30 seconds (figure 4).

2.2 Hardware

2.2.1 DAC Configuration

The DAC was setup in a similar fashion that that of lab 2. The big difference was that we no longer had the option of which chip select pin to use. To actually use the DAC, we needed to physically jump from the DAC chip select pin to pin RB10 as we had defined in our code:

```
PPSOutput(4, RPB10, SS2);
```

2.2.2 Game Controller

The game controller made use of a trim potentiometer (trim pot.) to move the paddle from the left end of the screen to the right end. The trim pot. is essentially a variable resistor of which we can read various resistance values by moving a wiper along the resistor (as shown in figure 2). Note: In our circuit, the input voltage was not 5 VDC but 3 VDC and the analog input that was used in our circuit was Analog Input 11 (A11).

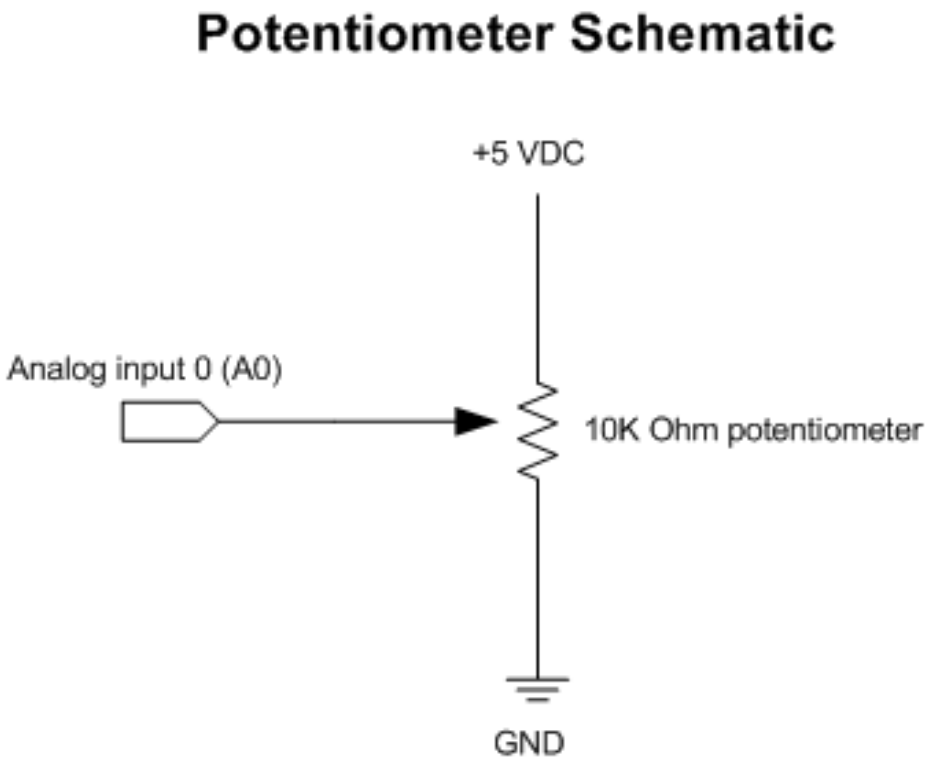

Figure 2: Circuit Schematic of a Trimmer Potentiometer

As the wiper moves along the resistor, it allows the current to flow along the portion of the resistor that is between the input voltage and the wiper. The current then flows out of the wiper into the analog input which is read by the software. What is not shown in this diagram is that we placed a resistor in series with the input voltage. This was to ensure that the full current coming from the 3V source would not be sent to the analog input in the case that the wiper was not measuring a resistance.

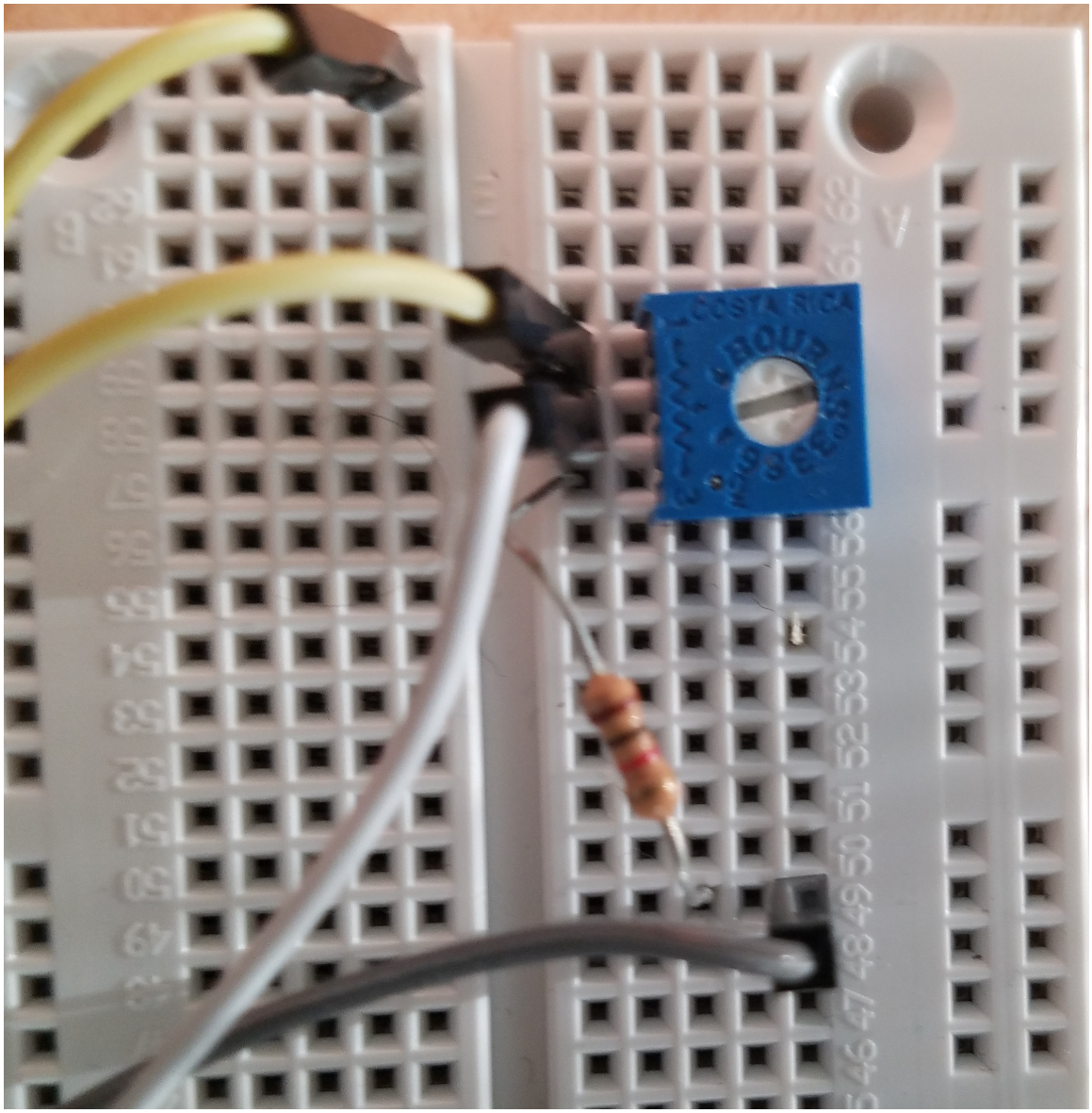

Figure 3: Actual Circuit for the Trimmer Potentiometer. In this figure, the yellow cable corresponds to the ground connection, the white connection is the analog input and the grey wire in series with the resistor is the voltage into the circuit.

3 Documentation

static PT_THREAD (protothread_timer(struct pt *pt))

The purpose of this thread was to count down from 3 seconds before the game starts to provide a more immersive user experience. During game play, the thread would update the fps, elapsed time, score and number of animated balls. At the end of game play, the thread would continuously print out "Game Over".

static PT_THREAD (protothread_anim(struct pt *pt))

This protothreads deals with animation of the balls and paddles. This protothread loops through each ball and checks whether it is entering a collision, if so it calls collisionCalculate, updates the ball position and velocity by taking into account drag. For the paddle, it reads the user input from the potentiometer and calls tft_drawLine. To draw the circle, it calls the function drawCircle.

static PT_THREAD (protothread_DACWrite(struct pt *pt))

This protothread would initiate the 3 different sound transfers that were required for this project. The thread would constantly check to see whether the soundEffect flag had been set to any of the pre-defined sound values, rewrite the timer to the period required for the sound playback and then initiate the sound transfer. The sound transfer would be allowed to run for 10 ms for the increment and decrement sounds and 2 seconds for the game over/countdown screen.

struct dv collisionCalculate(struct balls *firstBall, struct balls *secondBall)
This function is called in protothread_anim whenever the change in velocity due to a collision was necessary. It returns a change in velocity, dv. It calculates this change in velocity via dynamics explained in section 2.1.4.

void drawCircle(struct balls *ball, unsigned short color)
This function utilized the function tft_drawPixel to draw the balls. Each ball would be defined by 4 pixels, at the top, left, right, and bottom, respectively, as describes in 2.1.2.

4 Results and Discussion

4.1 Gameplay

Figure 4: End screen

Figure 4 shows the end screen which is displayed after some amount of time into the game when we decide to stop (in this case 30 seconds). The end tune was played alongside this screen and these events were linked to be executed together after completion of that time.

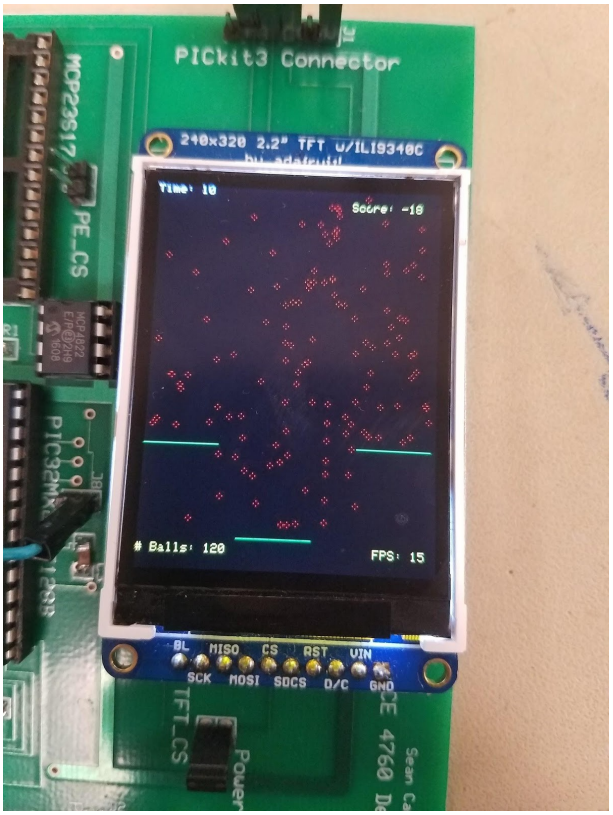

Figure 5: Gameplay with frame rate.

Figure 5 displays the frame rate that was set (in this case 15fps) and the number of balls animated 10 seconds into the game and total of 165 balls were seen on the screen at one point. 200+ balls were seen during points of testing. The screen shows the setup of the game as well which has two barriers at y position 220 and a moving paddle below it. The score variable gets updated and displays the score,though it is really difficult to keep a positive score.

4.2 DMA to DAC Output

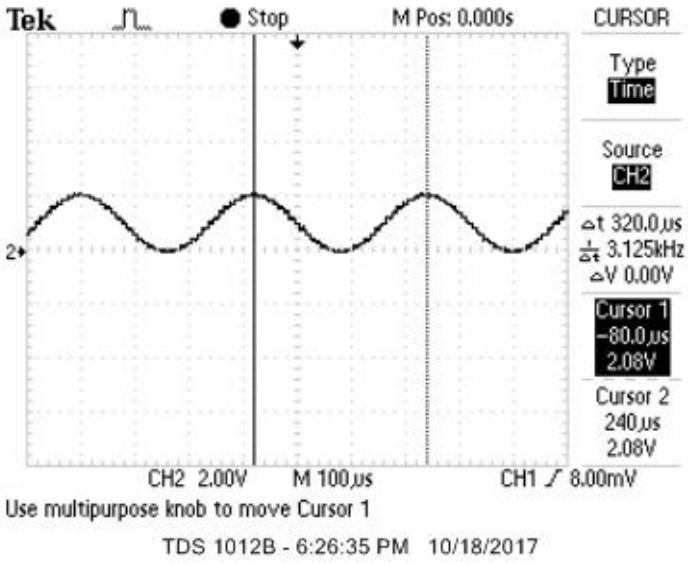

Figure 6: Oscilloscope Graph of the DAC Output of the Incrementing Score Sound Sampled at 110 KHz

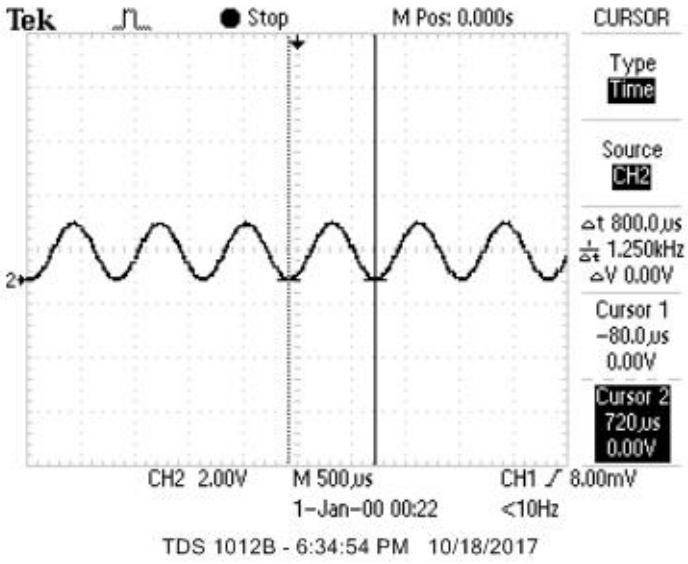

Figure 7: Oscilloscope Graph of the DAC Output of the Decrementing Score Sound Sampled at 44 KHz

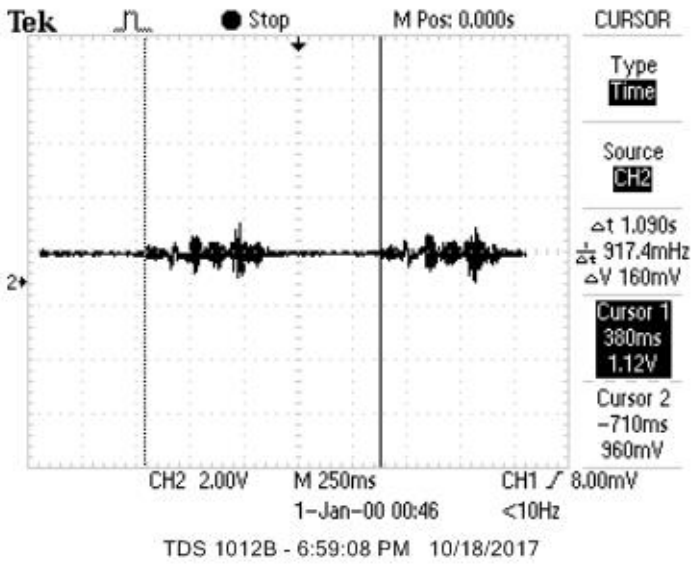

Figure 8: Oscilloscope Graph of the DAC Output of the Game End Sound Sampled at 4.4 KHz

As seen in these figures, the frequencies do not match up with the specifications that we had outlined. This is because our sine tables frequency is no longer at 2π but at 16π. This means that instead of sampling the table of size 256 every 1 period, we were sampling this table every 8 periods. If we take this into account, the theoretical value of the frequency should be as follows:

$$\text{Output Frequency} = \text{Sampling Frequency} \cdot \frac{8}{256}$$

Applying this formula to both the increment and decrement frequencies, we see that the theoretical values now become:

$$\text{Output Frequency (44KHz)} = 44\text{e}3 \cdot \frac{8}{256} = 1{,}375.0 \text{ Hz}$$
$$\text{Output Frequency (110KHz)} = 110\text{e}3 \cdot \frac{8}{256} = 3{,}437.5 \text{ HZ}$$

The correction for the sound file is a little different. Since the sound files sampling rate was scaled down by 10 (from 43,584 Hz to 4358 Hz), we had an array of size 4358 of which corresponded to 1 period of sound. The formulation to get the outputted frequency now becomes:

$$\text{Output Frequency (4.4KHz)} = 4.4\text{e}3 \cdot \frac{1}{4358} = 1.0096 \text{ Hz}$$

These values observed on the oscilloscope differ by a small amount of error. //For the 1375Hz frequency calculated above it shows 1250Hz on the oscilloscope and has a 9.26 percent decrease.
For the 3437.5Hz, we observe that the decrease of 9.09 percent which results to 3125Hz is shown on the oscilloscope and the 0.9174Hz instead of 1.0096Hz has a decrease of 9.4 percent. This could be attributed some noise entering into the system.

5 Conclusion

In this lab, we created a ball game with an optimized system to animate maximum number of balls on the screen. For the collisions, the balls were checked if they were colliding and then their respective velocities were updating by modeling a collision with no change in overall momentum and no less of energy.

We used a potentiometer to move the paddle. This was carried out by getting the analog values from the potentiometer and getting the corresponding digital values from the ADC to be used to update the paddle position on the display screen. The center of the paddle was tracked for its movement.

We manipulated the DrawPixel function instead of the DrawCircle function to make the process more faster and to use up less space for doing so. This reduced the CPU load of drawing animating circles resulting in displaying more balls on the screen.The sound outputs were generated for score decrement, score up and a tune at the end. The sound frequencies were resampled at 8khz except the score up tune which was done at 44khz and these tunes were generated from a header file produced by a Matlab script provided by Bruce Land. The DMA channel was used to drive the SPI channel for the sound outputs and the channel was tested using the sine wave generator used in LAB 2.

There are couple ways to expand on this lab to make a more robust and immersive experience. For example, we could interface with an SD card to provide more sounds without taking up more memory on the MCU. We could have scenarios to play more games depending on the user's choice. Each game could have it's own set of sounds on the SD card. We could use multiple potentiometers to create more possibilities of games, such as tennis or pong or snake.
Some problems were faced during the ball collisions, where the balls started acting abnormally after a while which were later fixed by adding multiple condition checks to only go forward with the collision detection. These problems were also troubleshooted by varying the hit counter and adjusting it to see how the balls acted in that scenario.

We implemented the whole lab by setting flags when certain events like score up or score down occurred and then added other subsections like the corresponding sound outputs to occur only if the particular flags are set.

Work Distribution

Jonah Collisions, ball animations, ball instantiation, DMA, Timer, GUI, Potentiometer.

Rahul Collision, ball animations, game set up

Sebastian Collisions, ball animations, ball instantiation, DMA.

6 References

[1] Mahbub, S.T. and Land, B.R. *DMA driven video, ADC and DAC on PIC32: Real time instrumentation on microcontroller.* Online. URL: `http://people.ece.cornell.edu/land/courses/ece4760/PIC32/pic32_DMA.pdf`.

7 Appendix

7.1 C Code

```
/*
 * File: lab3main.c
 * Author: Jonah Okike-Hephzibah, Sebastian Roubert, and Rahul Desai
 * Adapted from:
 * TFT_test_BRL4.c by
 * Author: Bruce Land
 * Target PIC: PIC32MX250F128B
 */

////////////////////////////////////
// clock AND protoThreads configure!
// You MUST check this file!
#include "config.h"
// threading library
#include "pt_cornell_1_2_1.h"
#include "lm_gold_mouse.h"

////////////////////////////////////
// graphics libraries
#include "tft_master.h"
#include "tft_gfx.h"
// need for rand function
#include <stdlib.h>
#include <math.h>
////////////////////////////////////

// A-channel, 1x, active
#define DAC_config_chan_A 0b0011000000000000

// Systems operating frequency:
```

```
#define SYS_FREQ 40000000
#define uint16_t unsigned short
#define uint8_t unsigned char
#define dmaChn 0 // The dma channel that the score increment and decrement plays on

static unsigned int DAC_data1 ;
static SpiChannel spiChn = SPI_CHANNEL2 ;
static int spiClkDiv = 2 ; // 20 MHz DAC clock
static int soundEffect = 0 ; // Flag that determines which sound effect to play

uint16_t table1[256]; // Used to create the sine table


/************************************************
The following code was developed by us to fulfill the requirements
set forth by Lab 3 of ECE 4760: Digital Systems Design using
Microcontrollers.

The goal was to create a TFT-LCD video game.

This game consisted of balls on the screen, two barriers opposite
where the balls entered, and a paddle whose horizontal position
could be controlled by the user via a potentiometer.

The balls on the screen can bounce off each other, bounce off the
walls, and bounce off the barriers. The balls also felt drag from the
table.

To get a point, you must bounce a ball off the paddle to the back-side
of a barrier. To lose a point, the ball must pass the paddle and hit the
wall behind it. To lose the game, you must reach negative points,
i.e. lose a point when you have zero points.

A sound is played from DMA whenever a you get a point, lose a point,
or lose the game.
 ************************************************/

// string buffer
char buffer[60];

// === thread structures ============================================
// thread control structs
// note that UART input and output are threads
static struct pt pt_timer, pt_color, pt_anim, pt_DACWrite ;

// system 1 second interval tick
```

```
int sys_time_seconds ; // counter for the elapsed time
static int ballRadius = 2 ;
static unsigned int adcRead = 0 ; // value to read the data from the potentiometer.
static int animBalls = 0 ; // counter for the number of balls currently being animated.
static int gameStart = 0 ; // Flag that states whether the game has started or not

// === the fixed point macros ========================================
typedef signed int fix16 ;
#define multfix16(a,b) ((fix16)(((( signed long long)(a))*(( signed long long)(b)))>>16)) //multiply
    two fixed 16:16
#define float2fix16(a) ((fix16)((a)*65536.0)) // 2^16
#define fix2float16(a) ((float)(a)/65536.0)
#define fix2int16(a) ((int)((a)>>16))
#define int2fix16(a) ((fix16)((a)<<16))
#define divfix16(a,b) ((fix16)((((signed long long)(a)<<16)/(b))))
#define sqrtfix16(a) (float2fix16(sqrt(fix2float16(a))))
#define absfix16(a) abs(a)

// The structure that defines the properties of a ball.
struct balls {
    fix16 x; // The x position
    fix16 y; // The y position
    fix16 vx; // The velocity in the x direction
    fix16 vy; // The velocity in the y direction
    int valid; // Flag to determine whether the ball is being animated
    int hitCounter; // The balls hit counter to determine if it can participate in a collision
};

// Struct that defines the velocity update for a collision
struct dv {
    fix16 dvx;
    fix16 dvy;
};

// Definition of the function that calculates the velocity update of a collision
struct dv collisionCalculate( struct balls *firstBall, struct balls *secondBall ) ;

//struct balls ball1;
//struct balls ball2;
int scoreCounter = 0 ; // counter that keeps track of the current game score
static int timeLeft = 0 ; // stores the remaining time in the animation thread after
// all the calculations have been carried out. This ensures that the frame rate is
// adequate to run the calculations for the designated number of balls
static int logis = 0 ;
static short r = 0 ; // Variable for the radius of the balls
static short x0, y0; // Variables to hold the center position of a ball
```

```
#define fps 67 ; // Current frame rate is defined to be 15 frames/sec
static int ballCounter = 2 ; // This value defines the rate at which balls are
// added to the screen. Balls are added every 2 frames.
static int paddleCenter = 115 ; // Variable to define the paddles center

// Definition of the drawCircle function.
void drawCircle( struct balls *ball, unsigned short color );

// This function takes in a pointer to a ball struct and color and draws a
// circle by drawing 4 pixels around the balls center (x0,y0))
void drawCircle( struct balls *ball,unsigned short color ) {

    r = 2 ;
    //tft_drawPixel()
    x0 = (short) fix2int16(ball->x) ;
    y0 = (short) fix2int16(ball->y) ;
    tft_drawPixel(x0 , y0+r, color);
    tft_drawPixel(x0 , y0-r, color);
    tft_drawPixel(x0+r, y0 , color);
    tft_drawPixel(x0-r, y0 , color);

}

// === Timer Thread =================================================
// update a 1 second tick counter as well as updates the tft display with the
// frame rate, score, elapsed time, and number of animated balls
static PT_THREAD (protothread_timer(struct pt *pt))
{
    PT_BEGIN(pt);

      while(1) {
        // yield time 1 second
        PT_YIELD_TIME_msec(1000) ;

        // Before the game starts, this displays a countdown till the game begins.
        // The current count down is 3 seconds long.
        if (gameStart <= 3) {

            // When 3 seconds have elapsed, the screen is painted black so that
            // the balls can be animated without the countdown text appearing.
            if (gameStart == 3) { tft_fillRect(0,0,236,320,ILI9340_BLACK ) ;
            } else {
            tft_fillRect(0,0,236,320,ILI9340_BLACK ) ;
            tft_setCursor(50,50) ;
            tft_setTextColor(ILI9340_WHITE) ; tft_setTextSize(6);
            tft_writeString("Game") ;
```

```
    tft_setCursor(20,100) ;
    tft_setTextColor(ILI9340_WHITE) ; tft_setTextSize(6);
    tft_writeString("Begins") ;
    tft_setCursor(110,150) ;
    tft_setTextColor(ILI9340_WHITE) ; tft_setTextSize(3);
    tft_writeString("In") ;

    tft_setCursor(110,200) ;
    tft_setTextColor(ILI9340_RED) ; tft_setTextSize(6);
    sprintf(buffer,"%d",3-gameStart) ;
    tft_writeString(buffer) ;
    }

    gameStart++ ;
} else {

// The game has begun and the elapsed time counter begins to increment.
sys_time_seconds++ ;

// If the elapsed time is greater than 30 seconds, the game is now over
if (sys_time_seconds > 30) {

    //tft_fillRect(x+r, y, w-2*r, h, color);
    tft_fillRect(0,0,236,320,ILI9340_WHITE ) ;

    tft_setCursor(50,100) ;
    tft_setTextColor(ILI9340_RED) ; tft_setTextSize(6);
    tft_writeString("Game") ;
    tft_setCursor(50,150) ;
    tft_setTextColor(ILI9340_RED) ; tft_setTextSize(6);
    tft_writeString("Over") ;
    sys_time_seconds++ ;
    soundEffect = 3 ;

} else {

    // If the game is not over, update the elapsed time, score, number
    // of animated balls and the frame rate.
tft_fillRoundRect(0,0, 100, 14, 1, ILI9340_BLACK);
tft_setCursor(0, 0);
tft_setTextColor(ILI9340_WHITE); tft_setTextSize(1);
sprintf(buffer,"Time: %d",sys_time_seconds) ;
tft_writeString(buffer);

// draw sys_time
tft_fillRoundRect(165,10, 100, 14, 1, ILI9340_BLACK);// x,y,w,h,radius,color
```

```
        tft_setCursor(165, 10);
        tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
        //if (ball1.hitCounter > 0){
        sprintf(buffer,"Score: %d", scoreCounter);
        tft_writeString(buffer);

        tft_fillRoundRect(0,300, 100, 14,1, ILI9340_BLACK) ;
        tft_setCursor(0,300);
        tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1) ;
        sprintf(buffer,"# Balls: %d",animBalls) ;
        tft_writeString(buffer) ;

        tft_fillRoundRect(190,300, 100, 14,1, ILI9340_BLACK) ;
        tft_setCursor(190,300);
        tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1) ;
        sprintf(buffer,"FPS: %d",15) ;
        tft_writeString(buffer) ;
        }
        // NEVER exit while
      } // END WHILE(1)
      }
  PT_END(pt);
} // timer thread

// This function calculates the velocity update for two balls (inputted as pointers
// to the ball objects).
struct dv collisionCalculate( struct balls *firstBall, struct balls *secondBall ) {

    fix16 r01[2] ; // Vector for the difference in position for the two balls
    fix16 v01[2] ; // Vector for the difference in velocity for the two balls
    fix16 rDotV ; // Variable that stores the dot product between r01 and v01.

    struct dv deltaV ; // This is the velocity update that the function will return

    r01[0] = firstBall->x - secondBall->x ;
    r01[1] = firstBall->y - secondBall->y ;

    v01[0] = firstBall->vx - secondBall->vx ;
    v01[1] = firstBall->vy - secondBall->vy ;

    rDotV = multfix16(r01[0],v01[0]) + multfix16(r01[1],v01[1]);

    deltaV.dvx = divfix16(multfix16(-r01[0],rDotV),int2fix16(16)) ;
    deltaV.dvy = divfix16(multfix16(-r01[1],rDotV),int2fix16(16)) ;

    // If the value of rDotV was too small for the division step,
```

```
    // we simply just swap the velocities of each ball
    if ( rDotV < 4){
        deltaV.dvx = -v01[0];

        deltaV.dvy = -v01[1];

    }

    return deltaV;
}

// === Animation Thread =============================================
// update a 1 second tick counter

// Defines the gravitational force and drag coefficient for the system
static fix16 g = float2fix16(0.1), drag = float2fix16(.01);
static struct dv dvCurr; // This struct holds the velocity update of a collision
int counting = 0 ; // This variable is used to ensure that balls are being added
// into the display every 2 frames.

static PT_THREAD (protothread_anim(struct pt *pt))
{
    PT_BEGIN(pt);

    // These variables are used to hold the difference in position of two balls
    static fix16 dx, dy ;

    int numBalls=300; // The number of balls that are being created
    static struct balls bArray[300]; // The array of balls both valid and invalid

    int i =0;
    int j =0;
    int maxTurn = 0 ; // The counter for the number of balls that are being added
    // in the current frame

    // Define each ball in the ball array.
    for (i = 0; i < numBalls; i++) {

        // Start with 2 valid balls.
        if (i <= 1) {
            bArray[i].valid = 1 ;
            bArray[i].x = int2fix16(180 - 5*i) ;
            bArray[i].y = int2fix16(30 + 8*i) ;
            bArray[i].vx = int2fix16(4) ;
            bArray[i].vy = int2fix16(3) ;
            bArray[i].hitCounter = 0 ;
```

```
    } else {
        // The rest of the balls will be invalid with a position outside of
        // the valid playing area
        bArray[i].valid = 0 ;
        bArray[i].x = int2fix16(0) ;
        bArray[i].y = int2fix16(0) ;
        bArray[i].vx = int2fix16( -1*(i%3 + 1) ) ;
        bArray[i].vy = float2fix16( -1*1.5 ) ;
        bArray[i].hitCounter = 0 ;
    }
}

// Variable to hold the time at which the thread begins the calculations
static int begin_time ;

  while(1) {

    begin_time = PT_GET_TIME();

    // Only calculate the ball positions if the game has officially begun
    // meaning, the tft_screen has been cleared of the game count down
    // text. This will also prevent further calculations from occurring
    // once the elapsed game time has been exceeded.
    if (sys_time_seconds< 30 && gameStart > 3) {

        // Draws the two barriers
        tft_drawLine(0, 220, 60,220 ,ILI9340_GREEN);
        tft_drawLine(235, 220, 175, 220,ILI9340_GREEN);

        // clears the counter variable for the number of balls present on
        // the screen
        animBalls = 0 ;

        // Loops through all of the balls.
        for (i = 0; i<numBalls; i++){

            // If the velocity is very low, this means that it is one of the oldest balls
            // and should no longer be valid. This sets the balls valid flag to
            // zero so that it will no longer be animated until shot back onto
            // the screen.
            if( absfix16(bArray[i].vx) <= float2fix16(0.5) && absfix16(bArray[i].vy) <= float2fix16
                (0.5)) {
                bArray[i].valid = 0 ;
            } else {
                // Only proceed with looping through the subsequent balls
                // from ball i's perspective if it is valid.
```

```
if (bArray[i].valid == 1){

    // Increment the ball counter because ball i is valid.
    animBalls++ ;

    // Loop through the subsequent balls
    for (j = i+1; j < numBalls; j++){

        // If ball j is not valid, do not consider a collision for it
        if (bArray[j].valid == 1) {

            // Calculate the distance between ball i and j's
            // centers.
            dx = absfix16(bArray[i].x - bArray[j].x);
            dy = absfix16(bArray[i].y - bArray[j].y);

            // If the distance between both x and y are less than 4,
            // then the balls are likely to collide
            if (dx < int2fix16(4) && dy < int2fix16(4)) {
                // This is a double check to ensure that the radial distance
                // is less than 16 and ball i is allowed to participate in a
                // collision.
                if ( ((multfix16(dx,dx)+multfix16(dy,dy))<int2fix16(16)) && (bArray[i
                    ].hitCounter == 0)) {

                    // Calculates the velocity update in both the
                    // x and y directions.
                    dvCurr = collisionCalculate( &bArray[i], &bArray[j] ) ;

                    // Applies the update for ball 1
                    bArray[i].vx = bArray[i].vx + dvCurr.dvx ;
                    bArray[i].vy = bArray[i].vy + dvCurr.dvy ;

                    // Applies the update for ball 2
                    bArray[j].vx = bArray[j].vx - dvCurr.dvx ;
                    bArray[j].vy = bArray[j].vy - dvCurr.dvy ;

                    // Set the hitCounter of both balls so that
                    // they both can no longer participate in a collision
                    // for the next 5 frames
                    bArray[i].hitCounter = 5;
                    bArray[j].hitCounter = 5;
                }
            }
        }
    }
```

```
            // Decrements the hit counter if it is greater than zero
            if (bArray[i].hitCounter > 0) bArray[i].hitCounter -= 1;

        // If ball i was not valid, we check to see if we can make it valid
        // The conditions for being able to be valid are that this
        // frame is allowed to add balls (using the counting counter)
        // and that the max number of balls has not been added for this
        // frame (maxTurn not greater than 3).
        } else if (counting == ballCounter && maxTurn < 3) {
            // If it legal to make ball i valid, set its starting
            // position to be at (150,7) with a velocity that is varied
            // between -3 and +3 in the x and 3 to 8 in the y.
            bArray[i].x = int2fix16(150) ;
            bArray[i].y = int2fix16(7) ;
            bArray[i].vx = float2fix16( -3 + i%13*0.5 ) ;
            bArray[i].vy = float2fix16( i%5+3 ) ;
            bArray[i].valid = 1 ;
            maxTurn++ ;
        }
    }
}

// If more balls have been shot unto the screen in this frame,
// set the counting value to -1 and reset the maxTurn variable
if (counting == ballCounter) {counting = -1 ; maxTurn = 0; }

// Loop through each ball and determine whether they need a position
// update
for (i=0; i < numBalls; i++) {

    // always clear the color of the ball in case it was just set to
    // be invalid
    drawCircle( &bArray[i], ILI9340_BLACK ) ;

    // Only do the calculations if the ball is valid
    if (bArray[i].valid == 1) {

        // Add drag to the ball
        bArray[i].vx = bArray[i].vx - multfix16(bArray[i].vx,drag);
        bArray[i].vy = bArray[i].vy - multfix16(bArray[i].vy,drag);

        // Update the position of the ball
        bArray[i].x = bArray[i].x + bArray[i].vx ;
        bArray[i].y = bArray[i].y + bArray[i].vy ;
```

```
// Check to see if the updated position puts the ball outside of
// the walls in the x direction
if (bArray[i].x<int2fix16(5) || bArray[i].x>int2fix16(235)){

    // If it does, set the position of the ball to be right
    // at the wall
    if (bArray[i].x<int2fix16(5) ) bArray[i].x=int2fix16(5);
    if (bArray[i].x>int2fix16(235) ) bArray[i].x=int2fix16(235);

    // Then flip the velocity of the ball
    bArray[i].vx = -bArray[i].vx ;

}

// Check to see if the updated position puts the ball outside
// of the walls in the y direction
if (bArray[i].y<int2fix16(5) || bArray[i].y>int2fix16(315)){

    // If it does, set the position of the ball to be right
    // at the wall
    if (bArray[i].y<int2fix16(5) ) bArray[i].y=int2fix16(5);
    if (bArray[i].y>int2fix16(315) ) bArray[i].y=int2fix16(315);

    // Flip the velocity of the ball so it does not continue
    // moving towards the wall
    bArray[i].vy = -bArray[i].vy ;
}

// Check to see whether the ball is within the bounds of the
// barrier.
if ( bArray[i].y<=int2fix16(221) && bArray[i].y>=int2fix16(216)){
    // The ball is on the right side of barrier so now we need
    // check to see if it is actually about to hit a barrier.
    // If it is going to hit either of the barriers, we set the
    // position to be right outside the barrier and flip the
    // velocity of the ball perpendicular to the barrier.
    if ( bArray[i].x>int2fix16(0) && bArray[i].x<int2fix16(62) ) {bArray[i].vy = -
        bArray[i].vy ; bArray[i].y = int2fix16(215) ;
    }else if ( bArray[i].x>int2fix16(175) && bArray[i].x<=int2fix16(235) ) {bArray[i
        ].vy = -bArray[i].vy ; bArray[i].y = int2fix16(215) ;}

}else if ( bArray[i].y>=int2fix16(219) && bArray[i].y<=int2fix16(225)){

    // The ball is on the left side of the barrier so we need
    // to check whether the ball is going through the barrier.
    // If it is, we increment the score of the player, and set the
```

```
            // ball to be invalid. The increment score sound is also triggered
            // by setting the soundEffect flag equal to two.
            if ( bArray[i].x>int2fix16(0) && bArray[i].x<int2fix16(61) ) { bArray[i].valid =
                0; scoreCounter++ ; soundEffect = 2 ; }
            else if ( bArray[i].x>int2fix16(175) && bArray[i].x<=int2fix16(235) ) {bArray[i].
                valid = 0; scoreCounter++ ; soundEffect = 2 ;}

        }

        // If the ball is at the proximity of the paddle, we need to
        // check if has made contact with the paddle
        if (bArray[i].y>=int2fix16(295)){
            if (bArray[i].x <= int2fix16(paddleCenter+30) && bArray[i].x >= int2fix16(
                paddleCenter-30)) {

                // If the ball has made contact with the paddle, we
                // set the balls position to be right on top of the
                // paddle and reverse the velocity.
                bArray[i].vy = -bArray[i].vy ;
                bArray[i].y = int2fix16(294) ;
            } else {
                // If the ball did not make contact with the paddle,
                // then the ball has gotten past the paddle and is
                // no longer valid. We decrement the score and set the
                // sound flag to play the decrement sound.
                bArray[i].valid = 0 ;
                scoreCounter-- ;
                soundEffect = 1 ;
            }
        }

        if (bArray[i].valid == 1){

            // If the ball is still valid after all of these checks,
            // we redraw the ball onto the tft.
            drawCircle( &bArray[i], ILI9340_RED ) ;
        }
    }
}

// The counting variable is incremented so that more balls can be
// added to the screen once counting == ballCounter
counting = counting+1 ;

// Clear the current drawing of the paddle. The paddle is defined from
// its center.
```

```
            tft_drawLine((short)(paddleCenter-30), 295, (short)(paddleCenter+30),295 ,ILI9340_BLACK);

            // The 0.3372 comes from the ratio of the maximum value of the center
            // (235-60) divided by the maximum value of the trim pot (519)) Note,
            // This is for a length of 60.

            // Read the value from the adc and convert it to a position of the
            // center of the paddle
            adcRead = (int) (0.3372 * (float) ReadADC10(0)) ;
            paddleCenter = adcRead + 30 ; // Account for the offset.
            // Redraw the paddle at the newly calculated position.
            tft_drawLine((short)(paddleCenter-30), 295, (short)(paddleCenter+30),295 ,ILI9340_GREEN);

            AcquireADC10() ; // This sets the ADC up so that the next value is
            // able to be read.
            // Have the thread yield for the remaining time to achieve the desired
            // frame rate
            PT_YIELD_TIME_msec(67 - (PT_GET_TIME() - begin_time)) ;

        // If the game is over or has not begun, play the sound file designated
        // by soundEffect = 3.
        } else { soundEffect = 3 ; PT_YIELD_TIME_msec(1000); }
    }
    PT_END(pt);
} // animation thread

static PT_THREAD (protothread_DACWrite(struct pt *pt)) {

        PT_BEGIN(pt);

    // This corresponds to the incrementing score sound
    if (soundEffect == 1) {

        // This sound is played at a higher sampling frequency (110 ksamples/sec)
        WritePeriod2(400) ;
        DmaChnEnable(dmaChn) ;
        WriteTimer2(0) ;
        PT_YIELD_TIME_msec(10) ;
        DmaChnAbortTxfer(dmaChn);
        soundEffect = 0 ;

    } else if (soundEffect == 2) {

        // This sound corresponds to a decrementing score and is sampled at the
        // required sampling rate of 44 ksamples/sec
        WritePeriod2(1000) ;
```

```
        DmaChnSetTxfer(dmaChn, &table1 , (void*)&SPI2BUF, 512, 2, 2);
        DmaChnEnable(dmaChn) ;
        WriteTimer2(0) ;
        PT_YIELD_TIME_msec(10) ;
        DmaChnAbortTxfer(dmaChn);
        soundEffect = 0 ;

    } else if (soundEffect == 3) {

        // This third sound is played whenever the game is about to begin or has
        // ended. It makes use of dma channel 1 and lasts longer than the
        // increment and decrement sounds.
        WritePeriod2(10000) ;
        DmaChnEnable(1) ;
        WriteTimer2(0) ;
        PT_YIELD_TIME_msec(2000) ;
        DmaChnAbortTxfer(1);
        soundEffect = 0 ;
    }

    // The soundEffect variable is cleared right after every sound is completed
    // to ensure no overlapping sounds.

        PT_END(pt);

}

// === Main ======================================================
void main(void) {
 //SYSTEMConfigPerformance(PBCLK);

        int table_size = 256 ;
    int i;

    // This for loop creates the sine table that is used for the increment and
    // decrement sounds
    for (i = 0; i < table_size; i++){
        table1[i] = (short)(2047*sin((float)i*50.2655/(float)table_size));
        table1[i] = (short)( DAC_config_chan_A | (table1[i] + 2048) ) ;
    }

    // This for loop configures the first 4 bits of the audio file to contain the
    // DAC config bits for DACA
    for( i = 0; i < sizeof(AllDigits); i++) {
        AllDigits[i] = (short) (DAC_config_chan_A | AllDigits[i]) ;
    }
```

```
  // Set up the ADC:
  CloseADC10() ;

  #define PARAM1 ADC_FORMAT_INTG16 | ADC_CLK_AUTO | ADC_AUTO_SAMPLING_OFF
  #define PARAM2 ADC_VREF_AVDD_AVSS | ADC_OFFSET_CAL_DISABLE | ADC_SCAN_OFF | ADC_SAMPLES_PER_INT_1 |
       ADC_ALT_BUF_OFF | ADC_ALT_INPUT_OFF
  #define PARAM3 ADC_CONV_CLK_PB | ADC_SAMPLE_TIME_5 | ADC_CONV_CLK_Tcy2
  #define PARAM4 ENABLE_AN11_ANA
  #define PARAM5 SKIP_SCAN_ALL

  SetChanADC10( ADC_CH0_NEG_SAMPLEA_NVREF | ADC_CH0_POS_SAMPLEA_AN11 );
  OpenADC10( PARAM1, PARAM2, PARAM3, PARAM4, PARAM5 );

  EnableADC10() ;

      // The timer is currently configured to run at 4.8 kHz for the audio file.
  // NOT THE SINE TABLE
      OpenTimer2(T2_ON | T2_SOURCE_INT | T2_PS_1_1, 9179);

  // set up DAC pins
      ANSELB =0; // turn off analog on B

      PPSOutput(2, RPB5, SDO2);
  PPSOutput(4, RPB10, SS2);
  // control CS for DAC

  // Configure the SPI to operate in framed mode.
  SpiChnOpen(SPI_CHANNEL2, SPI_OPEN_ON | SPI_OPEN_MODE16 | SPI_OPEN_MSTEN | SPI_OPEN_CKE_REV |
      SPICON_FRMEN | SPICON_FRMPOL, 2);

  // Open the desired DMA channel.
  DmaChnOpen(dmaChn,0, DMA_OPEN_AUTO);

  // transfer 2 bytes per interrupt call from the sine table
  DmaChnSetTxfer(dmaChn, &table1 , (void*)&SPI2BUF, 512, 2, 2);
  DmaChnSetEventControl(dmaChn, DMA_EV_START_IRQ(_TIMER_2_IRQ));

  // Sets up DMA channel 1 to transfer the audio file.
  DmaChnOpen(1, 1, DMA_OPEN_AUTO);
  DmaChnSetTxfer(1, &AllDigits, (void*)&SPI2BUF, sizeof(AllDigits), 2, 2);
  DmaChnSetEventControl(1, DMA_EV_START_IRQ(_TIMER_2_IRQ));

ANSELA = 0; ANSELB = 0;

// === config threads ==========
```

```
  // turns OFF UART support and debugger pin, unless defines are set
  PT_setup();

  // === setup system wide interrupts ========
  INTEnableSystemMultiVectoredInt();

  // init the threads
  PT_INIT(&pt_timer);
  PT_INIT(&pt_color);
  PT_INIT(&pt_anim);
  PT_INIT(&pt_DACWrite);

  // init the display
  tft_init_hw();
  tft_begin();
  tft_fillScreen(ILI9340_BLACK);
  //240x320 vertical display
  tft_setRotation(0); // Use tft_setRotation(1) for 320x240

  // seed random color
  srand(1);

  // round-robin scheduler for threads
  while (1){
      PT_SCHEDULE(protothread_timer(&pt_timer));
      PT_SCHEDULE(protothread_anim(&pt_anim));
      PT_SCHEDULE(protothread_DACWrite(&pt_DACWrite));
      }
  } // main

// === end ==================================================
```