# PADDLE BALL

## INTRODUCTION

Paddle Ball is a video game in which ball-like particles enter from one side of the screen and a paddle with its position being controlled by an analog input has been used to bounce the balls into the collector walls to score points. If the balls get past the paddle to the left side of the screen the score gets decremented.

The goal was to optimize the code to generate as many balls as possible. This has been partly achieved by using DMA channel to produce sound effects for the game. This DMA channel was used to drive the SPI DAC in order to produce sound effects during points scoring and at the end of the game. The balls in this video game have been modelled as frictionless balls of equal mass. But a small drag component has been added to slow the balls after a certain period of time.

## DESIGN

### Ball dynamics/Integration Algorithm -

The balls generated in the video game are assumed to frictionless and of equal mass. The coordinate system for the TFT display has x increasing to the right and y increasing downwards. When a collision happens between two balls the impact force acts in a direction parallel to the line connecting the centers of the two colliding balls. The change in velocity will also be parallel with respect to the connecting line between their centers. Consequently, the component of velocity parallel to the line will have it sign reversed and the component perpendicular to the line will remain unchanged.

The components of the integration algorithm are described below:

- Initially a structure has been created for the balls with different objects such as the x_position,y_position,x_velocity,y_velocity and hit_counter in fix16 format.

- Two ‘for’ loops have been used to make sure that every ball created in the game would have a collision with all the others. To implement the ball dynamics easily in the code all the vectors have been resolved separately into their X and Y counterparts. Since this process is time consuming, the collision has been performed between any two balls only if they were less than 2 radii apart.

- If the distance between the centers of the balls is greater than 2 then the velocity change is calculated according to the formula given below:

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

where

$$\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$$

$$\vec{v}_{ij} = \vec{v}_i - \vec{v}_j$$

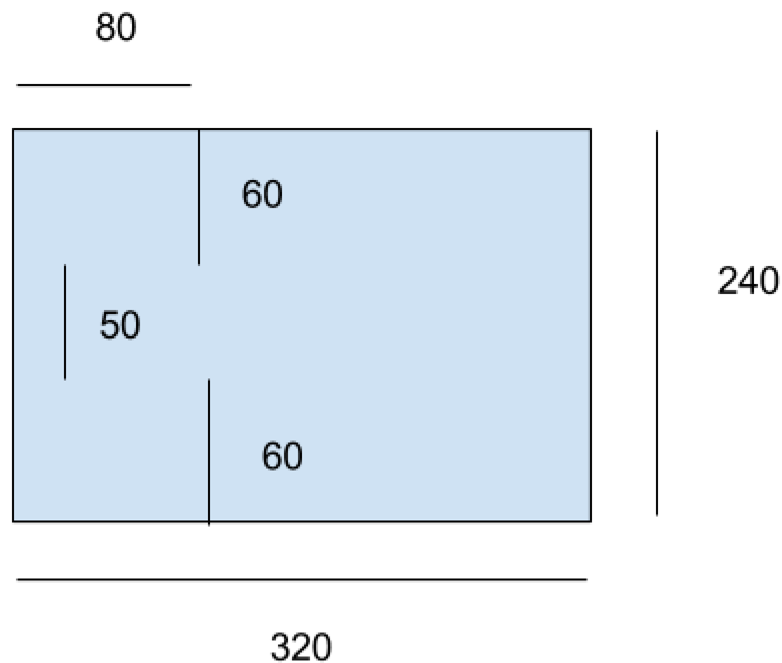
- In the code the velocity_change has also been calculated separately for x and y components. Subsequently this velocity_change had been added to the initial velocities of the two balls considered(i.e i and j) to generate new velocities for the balls after collision. The snippet of the code below shows the velocity update of the balls i and j.

```
dot_product = multfix16(rij_x, vij_x) + multfix16(rij_y, vij_y);
deltavi_x = -((multfix16(rij_x, dot_product)) >> 4);
deltavi_y = -((multfix16(rij_y, dot_product)) >> 4);

//new vi components
ball[i].x_vel = ball[i].x_vel + deltavi_x;
ball[i].y_vel = ball[i].y_vel + deltavi_y;

//new vj components
ball[j].x_vel = ball[j].x_vel - deltavi_x;
ball[j].y_vel = ball[j].y_vel - deltavi_y;
```

- It is not possible for the ball to ball collisions to be exact because of finite time steps. One consequence of this is that balls tend to capture each other when they collide and they will start orbiting one another. To avoid this, capture a variable ‘hitcounter’ has been initialised to 4 and decremented to 0 giving a time elapse of few frames before the ball collides with another ball or the paddle.

**Setting up of the gaming Environment-**

- A playing field consisting of a rectangle 320 wide x 240 high is setup on the LCD screen with two internal barriers. The barriers have been placed at about 1/3 of the x-width of the screen and about 1/4 of the screen length. Essentially each barrier is of length 60.




Playing Field

- The balls are launched from the coordinates (309,120) which is the bottom-center of the display. The x-component of the velocity of each ball is set to -2 pixels/frame and the y-component of the velocity has been randomized to take values between -1.5 to 1.5 pixels/frame (*multfix16((((rand() & 0xffff)<<1) - int2fix16(1)), float2fix16(1.5))*).

- **Paddle draw**: The paddle location is controlled by the player using the screwdriver. The voltage varied (through a potentiometer) is converted into a digital signal using the Analog to Digital Converter. The five setup parameters (param_1, param_2, param_3, param_4 and param_5) required for the configuration of the OpenADC10 have been initialized in the 'main'. The position at which the paddle has to be drawn is known by reading the result of channel 9 conversion from the idle buffer (adc_9 = ReadADC10(0)). The paddle is drawn every time according to the position specified by the user at the given frame rate.

- Thus at each frame time, the velocity and the position of all the balls on the screen are updated and the paddle is redrawn according to its new position.

- **Setup of wall and paddle collisions**: If the ball collides towards the right side of the two barriers the x-component of the velocity of the ball is negated (ball[i].x_vel = -ball[i].x_vel) which essentially means a head on collision. Similarly, when the ball collides with the paddle the x-component of the velocity is negated reversing the direction of the ball.

- **Score setup in the game**: If the ball which are deflected into the left side of the bottom and top walls the score gets incremented by 1 and the ball is removed from the screen. If the ball gets past the paddle towards the left side of the screen the score is decremented by 1 and the ball is removed from the screen.

- **Launching of the balls**: Memory allocation for about 350 balls was set up for this game. For every 250 msec the ball status is made valid as long the the ball is within the maximum ball limit of 350 and the frame rate is greater than 17. Using this logic the maximum number of balls possible are made to appear on the screen within the given frame rate condition.

- **Clearing the balls:** If the status of the ball is valid and the ball count is below the maximum number of balls a counter (valid_time) starts counting from 1 to 100. Once it finishes counting the ball status becomes invalid and it is cleared off the screen by printing the ball black. This way the oldest balls on the screen will get cleared at regular intervals of time.

- **Parameters displayed on the screen**: A score, a frame rate, number of current balls, and time have been displayed on the screen in that order. The frame rate has been calculated by using frame_rate = (1000/(PT_GET_TIME() - begin_time)) . The time to be displayed on the screen is calculated in seconds using a timer thread. Inside this thread a variable sys_time_seconds is incremented for every 1000 msec using PT_YIELD_TIME_msec(1000) . The time limit for the game is 60 seconds after which the screen gets cleared and ‘GAME OVER’ is printed on the screen.

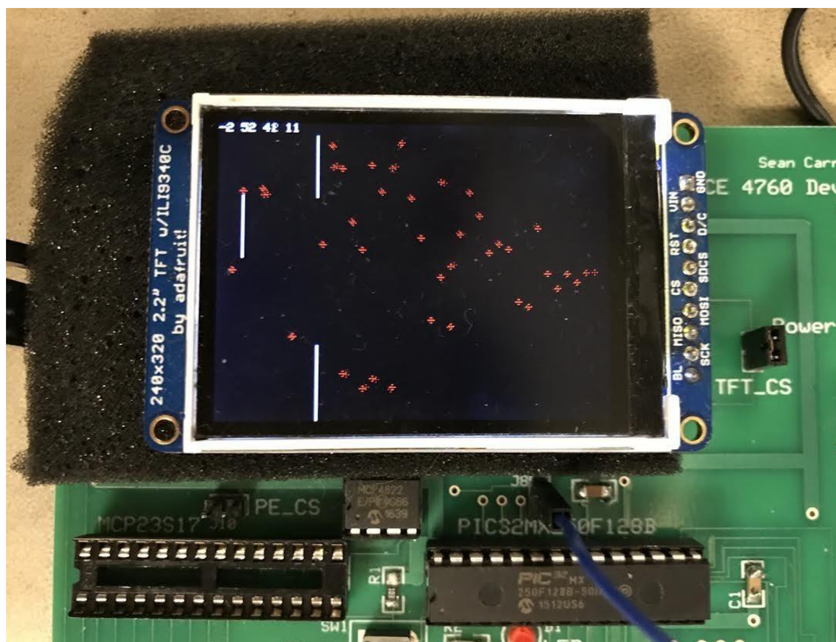
**DMA setup and creation of sound effects-**

In this project a DMA channel has been utilized to create the sound effects for the video game to generate the maximum number of balls on the screen. DMA uses memory controllers separate from the CPU to accelerate data movement between memory locations, or between peripherals and memory. The PIC32 has 4 DMA controllers which can stream an aggregate of about 3.2 megabytes/sec without affecting CPU performance, in many cases.

**DMA setup in the main-**

Three channels of DMA have been initialized to produce three different sound effects (one during increment of score, one during decrement of score and one during game over). All the three channels (0,1,2) have been initialized in default mode and the destination address is set as the SPI2BUF which is also set up in the main. Additionally, a separate DMA thread was created to yield a time of 500 msec.

<u>**RESULTS**</u>




<u>Display of the game setup</u>

1. Number of maximum balls generated: 168
2. Game time: 60 seconds

3. Synthesis rate was 44ksamples/second for one sound
4. Generated all the sounds using DMA and not ISR

**CONCLUSION**

TFT-LCD video game was successfully implemented and were able to generate around 168 balls and had 3 separate sounds for +1 score, -1 score and end game. The balls generation was random and the generated balls were bouncing of each other and the walls properly. TFT frame time was maintained at 17 frames/second allowing us as much time as possible for computation and the TFT screen displayed 4 parameters - score, frame rate, animated balls and time in seconds. All the sound effects were generated using DMA and the synthesis rate was 44 ksamples/second for one sound.

**APPENDIX CODE**

```
/*
 * File:        TFT_test_BRL4.c
 * Author:      Dev Sanghvi
 * Adapted from:
 *              main.c by
 * Author:      Syed Tahmid Mahbub
 * Target PIC:  PIC32MX250F128B
 */

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

////////////////////////////////////
// graphics libraries
#include "tft_master.h"
#include "tft_gfx.h"
// need for rand function
#include <stdlib.h>
////////////////////////////////////
#include <math.h>
/* Demo code for interfacing TFT (ILI9340 controller) to PIC32
 * The library has been modified from a similar Adafruit library
 */
// Adafruit data:
```

```
/***************************************************
  This is an example sketch for the Adafruit 2.2" SPI display.
  This library works with the Adafruit 2.2" TFT Breakout w/SD card
  ----> http://www.adafruit.com/products/1480

  Check out the links above for our tutorials and wiring diagrams
  These displays use SPI to communicate, 4 or 5 pins are required to
  interface (RST is optional)
  Adafruit invests time and resources providing this open source code,
  please support Adafruit and open-source hardware by purchasing
  products from Adafruit!

  Written by Limor Fried/Ladyada for Adafruit Industries.
  MIT license, all text above must be included in any redistribution
 ****************************************************/
```

```
#include "Mike_digits_8khz_packed.h"
#include "game_end.h"
// string buffer
char buffer[60];

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

// system 1 second interval tick
static int sys_time_seconds;

// === 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)

// === Timer Thread =================================================
// update a 1 second tick counter
static PT_THREAD (protothread_timer(struct pt *pt))
{
    PT_BEGIN(pt);
      while(1) {
        // yield time 1 second
        PT_YIELD_TIME_msec(1000) ;
        sys_time_seconds++ ;

        // NEVER exit while
      } // END WHILE(1)
  PT_END(pt);
} // timer thread


struct BALL {
    fix16 x_pos;
    fix16 y_pos;
    fix16 x_vel;
```

```
    fix16 y_vel;
    fix16 hit_counter;
    int ball_status;
    int valid_time;
};


// change both these variables for change in the number of balls
//================================================
//number of balls being animated
static int ball_number = 350;

//memory allocation for balls
static struct BALL ball[350];

//=================================================

//ball radius
static fix16 ball_radius = int2fix16(2);

//ball indices
static fix16 i = 0;
static fix16 j = 0;


// x coordinate of balls
static fix16 xi = 0;
static fix16 xj = 0;

// y coordinate of balls
static fix16 yi = 0;
static fix16 yj = 0;

//x and y coordinates of the distance vector between any 2 balls
static fix16 rij_x = 0;
static fix16 rij_y = 0;

//x and y components of the relative velocity of ball i and ball j
static fix16 vij_x = 0;
static fix16 vij_y = 0;


//the change in the velocity of ball i
static fix16 deltavi_x = 0;
static fix16 deltavi_y = 0;
```

```
// intermediate computation variable for dot product
static fix16 dot_product = 0;

//15 fps
static int begin_time = 0;

//velocity swap variables
static fix16 temp_x = 0;
static fix16 temp_y = 0;


//score tracker
static int score = 0;


//paddle location
static short paddle_xpos = 0;
static short paddle_ypos = 0;

//testing
static int print_flag = 0;
static int k = 0;
static int a =0;
static int frame_rate = 0;
static int disappear_index = 0;

//initial shooting position
static int launch_timer_flag = 0;
static int temp_timer = 0;

//live ball count tracker
static int animated_ball = 0;

//game end condition
static int game_end = 0;
static int clearing_flag = 0;


//music elements
#define sine_table_size 256
volatile unsigned short sine_table[sine_table_size];
static int s = 0;
#define DAC_config_chan_A 0b0011000000000000
```

```
#define DAC_config_chan_B 0b1011000000000000
static int plus_song = 0;
static int minus_song = 0;
static int game_over_song = 0;
#define dmaChn_0 0
#define dmaChn_1 1
#define dmaChn_2 2

// === Animation Thread =============================================
// update a 1 second tick counter
//static  fix16  xc=int2fix16(10),  yc=int2fix16(150),  vxc=int2fix16(2),
vyc=0;
static fix16 g = float2fix16(0.1), drag = float2fix16(0.00001);

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


    //adc test code
    //===========================
    static int adc_9;

    //wall draw
    tft_drawLine(80, 0, 80, 60, ILI9340_WHITE);
    //
    tft_drawLine(80, 180, 80, 240, ILI9340_WHITE);
    while (1) {
        // yield time 1 second
        begin_time = PT_GET_TIME();

        if(launch_timer_flag == 0)
        {
            temp_timer = PT_GET_TIME();
            launch_timer_flag =  1;
        }

        //GAME OVER CONDITION

        if(sys_time_seconds == 60)
        {
           game_end = 1;
        }

        if(game_end == 0)
```

```
            {
            //adc test code
            //================================
            // read the ADC AN11
            // read the first buffer position
                  adc_9 = ReadADC10(0); // read the result of channel 9
conversion from the idle buffer
            AcquireADC10(); // not needed if ADC_AUTO_SAMPLING_ON below

            //paddle location and printing
            //======================================================
            paddle_ypos = ((adc_9 * 190) >> 10);
            //start paddle from left side of screen
            paddle_ypos = 189 - paddle_ypos;

            //ball launcher.
            if ( (a < ball_number) && (frame_rate > 17) )
            {
                if((PT_GET_TIME() - temp_timer) > 250 )
                {
                    ball[a].ball_status = 1;
                    animated_ball = animated_ball + 1;
                    a = a + 1;
                    temp_timer = PT_GET_TIME();
                }
            }


            // check for each ball with every other ball
            for (i = 0; i < ball_number; i++)
            {
                for (j = i + 1; j < ball_number; j++)
                {
                    if ((ball[i].ball_status == 1) && (ball[j].ball_status
== 1))
                    {
                             // the sign of rij i.e. its direction may  be
flipped
                        rij_x = ball[i].x_pos - ball[j].x_pos;
                        rij_y = ball[i].y_pos - ball[j].y_pos;


                        // are the balls close enough for collision ?
                                 if ((absfix16(rij_x) <= int2fix16(4)) &&
(absfix16(rij_y) <= int2fix16(4)))
```

```
                        {
                                if (absfix16(rij_x) <= int2fix16(2)  &&
(absfix16(rij_y) <= int2fix16(2)))
                            {
                                //prevent the overlap of balls. i.e. rij
components cannot be 0
                                temp_x = ball[i].x_vel;
                                temp_y = ball[i].y_vel;

                                ball[i].x_vel = ball[j].x_vel;
                                ball[i].y_vel = ball[j].y_vel;

                                ball[j].x_vel = temp_x;
                                ball[j].y_vel = temp_y;

                                ball[i].hit_counter = int2fix16(4);

                            }
                                else if ((((multfix16(rij_x, rij_x))  +
(multfix16(rij_y,  rij_y)))  <=  int2fix16(16)) && (ball[i].hit_counter ==
int2fix16(0)))
                            {
                                    // x component of relative velocity
between ball i and j
                                vij_x = ball[i].x_vel - ball[j].x_vel;

                                    // y component of relative velocity
between ball i and j
                                vij_y = ball[i].y_vel - ball[j].y_vel;


                                     // delta vi computation. The below
implementation relies on ball radius being 2
                                dot_product = multfix16(rij_x, vij_x) +
multfix16(rij_y, vij_y);


                                //delta vi
                                        deltavi_x = -((multfix16(rij_x,
dot_product)) >> 4);
                                        deltavi_y =  -((multfix16(rij_y,
dot_product)) >> 4);

                                //new vi components
                                ball[i].x_vel = ball[i].x_vel + deltavi_x;
```

```
                                ball[i].y_vel = ball[i].y_vel + deltavi_y;

                                //new vj components
                                ball[j].x_vel = ball[j].x_vel - deltavi_x;
                                ball[j].y_vel = ball[j].y_vel - deltavi_y;

                                // avoid balls from orbiting one another
                                ball[i].hit_counter = int2fix16(4);
                            }
                        }
                        if (ball[i].hit_counter > int2fix16(0))
                        {
                               ball[i].hit_counter = ball[i].hit_counter -
int2fix16(1);
                        }

                    }
                }
            }
            for(i = 0; i < ball_number; i++)
            {
                // i updates

                if (ball[i].ball_status == 1)
                {
                    //clear the old ball location
                                  tft_drawPixel(fix2int16(ball[i].x_pos),
fix2int16(ball[i].y_pos), ILI9340_BLACK);
                              tft_drawPixel(fix2int16(ball[i].x_pos) + 2,
fix2int16(ball[i].y_pos), ILI9340_BLACK);
                              tft_drawPixel(fix2int16(ball[i].x_pos) - 2,
fix2int16(ball[i].y_pos), ILI9340_BLACK);
                                  tft_drawPixel(fix2int16(ball[i].x_pos),
fix2int16(ball[i].y_pos) + 2, ILI9340_BLACK);
                                  tft_drawPixel(fix2int16(ball[i].x_pos),
fix2int16(ball[i].y_pos) - 2, ILI9340_BLACK);



                    //update ball i velocity
                                      ball[i].x_vel  =  ball[i].x_vel  -
multfix16(ball[i].x_vel, drag);
                                      ball[i].y_vel  =  ball[i].y_vel  -
multfix16(ball[i].y_vel, drag);
```

```
                    //update ball i location
                    ball[i].x_pos = ball[i].x_pos + ball[i].x_vel;
                    ball[i].y_pos = ball[i].y_pos + ball[i].y_vel;

                    //perimeter limiter
                      if (ball[i].x_pos > int2fix16(315)) ball[i].x_vel  =
-ball[i].x_vel;
                      if (ball[i].y_pos < int2fix16(10) || ball[i].y_pos >
int2fix16(230)) ball[i].y_vel = -ball[i].y_vel;

                    // top and bottom wall bouncing on the right side
                      if ((ball[i].x_vel < int2fix16(0)) && (ball[i].x_pos
<= int2fix16(85)) &&(ball[i].x_pos >= int2fix16(80)) && (ball[i].y_pos <=
int2fix16(60))) ball[i].x_vel = -ball[i].x_vel;
                      if ((ball[i].x_vel < int2fix16(0)) && (ball[i].x_pos
<= int2fix16(85)) &&(ball[i].x_pos >= int2fix16(80)) && (ball[i].y_pos >=
int2fix16(180))) ball[i].x_vel = -ball[i].x_vel;


                    //ball i is lost to the left perimeter
                    if (ball[i].x_pos < int2fix16(5))
                    {
                        score = score - 1;
                        animated_ball = animated_ball - 1;
                        ball[i].ball_status = 0;
                        //TRIGGER DMA SOUND
                        DmaChnEnable(dmaChn_0);
                    }

                    //ball i top wall
                      if ((ball[i].x_vel > int2fix16(0)) && (ball[i].x_pos
>= int2fix16(76)) && (ball[i].x_pos <= int2fix16(79)) && (ball[i].y_pos <=
int2fix16(60)))
                    {
                        score = score + 1;
                        animated_ball = animated_ball - 1;
                        ball[i].ball_status = 0;
                        //TRIGGER DMA SOUND
                        DmaChnEnable(dmaChn_1);

                    }

                    //ball i bottom wall
                      if ((ball[i].x_vel > int2fix16(0)) && (ball[i].x_pos
>= int2fix16(76)) && (ball[i].x_pos <= int2fix16(80)) && (ball[i].y_pos >=
```

```
int2fix16(180)))
                    {
                        score = score + 1;
                        animated_ball = animated_ball - 1;
                        ball[i].ball_status = 0;
                        //TRIGGER DMA SOUND
                        DmaChnEnable(dmaChn_1);

                    }

                    //ball i paddle bounce
                     if ((ball[i].x_pos < int2fix16(22)) && (ball[i].x_pos
>   int2fix16(18))   &&   (ball[i].y_pos   >=   int2fix16(paddle_ypos))   &&
(ball[i].y_pos <= int2fix16(paddle_ypos + 50)))
                    {
                        ball[i].x_vel = -ball[i].x_vel;
                    }



                    //remove ball i if it becomes invalid
                    if (ball[i].ball_status == 1)
                    {
                        //  draw ball i
                                   tft_drawPixel(fix2int16(ball[i].x_pos),
fix2int16(ball[i].y_pos), ILI9340_RED);
                               tft_drawPixel(fix2int16(ball[i].x_pos) + 2,
fix2int16(ball[i].y_pos), ILI9340_RED);
                               tft_drawPixel(fix2int16(ball[i].x_pos) - 2,
fix2int16(ball[i].y_pos), ILI9340_RED);
                                   tft_drawPixel(fix2int16(ball[i].x_pos),
fix2int16(ball[i].y_pos) + 2, ILI9340_RED);
                                   tft_drawPixel(fix2int16(ball[i].x_pos),
fix2int16(ball[i].y_pos) - 2, ILI9340_RED);
                    }
                }
            }

            //oldest ball disappear
                                if((disappear_index   <   ball_number)   &&
(ball[disappear_index].ball_status == 1))
            {
                ball[disappear_index].valid_time +=1;
                if(ball[disappear_index].valid_time > 100)
                {
```

```
                    ball[disappear_index].ball_status = 0;

                     tft_drawPixel(fix2int16(ball[disappear_index].x_pos),
fix2int16(ball[disappear_index].y_pos), ILI9340_BLACK);
                    tft_drawPixel(fix2int16(ball[disappear_index].x_pos) +
2, fix2int16(ball[disappear_index].y_pos), ILI9340_BLACK);
                    tft_drawPixel(fix2int16(ball[disappear_index].x_pos) -
2, fix2int16(ball[disappear_index].y_pos), ILI9340_BLACK);
                     tft_drawPixel(fix2int16(ball[disappear_index].x_pos),
fix2int16(ball[disappear_index].y_pos) + 2, ILI9340_BLACK);
                     tft_drawPixel(fix2int16(ball[disappear_index].x_pos),
fix2int16(ball[disappear_index].y_pos) - 2, ILI9340_BLACK);
                    disappear_index +=1;

                }

            }
            //PADDLE DRAW
            tft_drawLine(20, 0, 20, 240, ILI9340_BLACK);
                    tft_drawLine(20,  paddle_ypos,  20,  paddle_ypos  +  50,
ILI9340_WHITE);


            frame_rate = (1000/(PT_GET_TIME() - begin_time));
            //display text on screen
            tft_fillRect(0, 0, 110, 10, ILI9340_BLACK);
            tft_setCursor(0, 0);
            tft_setTextColor(ILI9340_WHITE);
            tft_setTextSize(1);
                    sprintf(buffer,  "%d  %d  %d  %d",  score,  frame_rate,
animated_ball, sys_time_seconds);
                                    //sprintf(buffer,   "%f   %f   %f
%f",fix2float16(ball[i].x_vel),fix2float16(              ball[i].y_vel),
fix2float16(ball[j].x_vel), fix2float16(ball[j].y_vel));
            tft_writeString(buffer);



            PT_YIELD_TIME_msec(67 - (PT_GET_TIME() - begin_time));
        }else{
            if(clearing_flag == 0)
            {
                //display text on screen
                tft_fillRect(0, 0, 320, 240, ILI9340_BLACK);
                clearing_flag = 1;
```

```
            }
            tft_setCursor(0, 0);
            tft_setTextColor(ILI9340_WHITE);
            tft_setTextSize(3);
            sprintf(buffer, "GAME OVER !!");
            tft_writeString(buffer);
            //TRIGGER DMA SOUND
            DmaChnEnable(dmaChn_2);
            PT_YIELD_TIME_msec(1000);
        }

        } // END WHILE(1)
  PT_END(pt);
} // animation thread



//dma thread

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

    PT_YIELD_TIME_msec(500);

    PT_END(pt);
}



// === Main  ======================================================
void main(void) {
 //SYSTEMConfigPerformance(PBCLK);
  int i = 0;
  ANSELA = 0; ANSELB = 0;

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

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

    //adc setup
    //=================================================
```

```
    // the ADC ////////////////////////////////////////
    // configure and enable the ADC
        CloseADC10();  //  ensure  the  ADC  is  off  before  setting  the
configuration

    // define setup parameters for OpenADC10
      // Turn module on | ouput in integer | trigger mode auto | enable
autosample
        //  ADC_CLK_AUTO  --  Internal  counter  ends  sampling  and  starts
conversion (Auto convert)
      // ADC_AUTO_SAMPLING_ON -- Sampling begins immediately after last
conversion completes; SAMP bit is automatically set
    // ADC_AUTO_SAMPLING_OFF -- Sampling begins with AcquireADC10();
            #define   PARAM1      ADC_FORMAT_INTG16   |   ADC_CLK_AUTO   |
ADC_AUTO_SAMPLING_OFF //

    // define setup parameters for OpenADC10
     // ADC ref external  | disable offset test | disable scan mode | do 1
sample | use single buf | alternate mode 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 setup parameters for OpenADC10
    // use peripherial bus clock | set sample time | set ADC clock divider
    // ADC_CONV_CLK_Tcy2 means divide CLK_PB by 2 (max speed)
    // ADC_SAMPLE_TIME_5 seems to work with a source resistance < 1kohm
    #define PARAM3 ADC_CONV_CLK_PB | ADC_SAMPLE_TIME_5 | ADC_CONV_CLK_Tcy2
//ADC_SAMPLE_TIME_15| ADC_CONV_CLK_Tcy2

    // define setup parameters for OpenADC10
    // set AN11 and  as analog inputs
    #define PARAM4     ENABLE_AN11_ANA // pin 24

    // define setup parameters for OpenADC10
    // do not assign channels to scan
    #define PARAM5     SKIP_SCAN_ALL

    // use ground as neg ref for A | use AN11 for input A
    // configure to sample AN11
    SetChanADC10(ADC_CH0_NEG_SAMPLEA_NVREF | ADC_CH0_POS_SAMPLEA_AN11); //
configure to sample AN11
     OpenADC10(PARAM1, PARAM2, PARAM3, PARAM4, PARAM5); // configure  ADC
using the parameters defined above

    EnableADC10(); // Enable the ADC
```

```
//=================================================
//SPI INITIALIZATION
//=================================================

SpiChnOpen(SPI_CHANNEL2, SPI_OPEN_ON | SPI_OPEN_MODE16 | SPI_OPEN_MSTEN |
SPI_OPEN_CKE_REV | SPICON_FRMEN | SPICON_FRMPOL, 2);
PPSOutput(2, RPB5, SDO2);
PPSOutput(4, RPB10, SS2);



// DMA SETUP
//===================================================
        int table_counter;

              for  (table_counter  =  0;  table_counter  <  sine_table_size;
table_counter++)
        {
               sine_table[table_counter] = DAC_config_chan_A | ((unsigned
short) ((2047.0 * sin(table_counter*6.283/sine_table_size) )+ 2048.0));
        }


    //timer for -1 score
    OpenTimer2(T2_ON | T2_SOURCE_INT | T2_PS_1_1, 2500);

    //timer for +1 score
    OpenTimer3(T3_ON | T3_SOURCE_INT | T3_PS_1_1, 909);

    //timer for end game
    OpenTimer4(T4_ON | T4_SOURCE_INT | T4_PS_1_1, 3990);

    // Open the desired DMA channel.

    // We enable the default mode for all 3 channels
    DmaChnOpen(dmaChn_0, 0, DMA_OPEN_DEFAULT);
    DmaChnOpen(dmaChn_1, 1, DMA_OPEN_DEFAULT);
    DmaChnOpen(dmaChn_2, 0, DMA_OPEN_DEFAULT);

    // set the transfer parameters: source & destination address, source &
destination size, number of bytes per event
      //  Setting  the  last  parameter  to  one  makes  the  DMA  output  one
byte/interrupt
```

```
    //chn 0. Set the music
        DmaChnSetTxfer(dmaChn_0,  (void*)  sine_table,  (void*)  &SPI2BUF,
sine_table_size * 2, 2, 2);

    //chn1. Set the music
        DmaChnSetTxfer(dmaChn_1,  (void*)  plus_sound,  (void*)  &SPI2BUF,
sine_table_size * 2, 2, 2);

    //chn2 . Set the music
        DmaChnSetTxfer(dmaChn_2,  (void*)  game_end1,  (void*)  &SPI2BUF,
sine_table_size * 2, 2, 2);



     // set the transfer event control: what event is to start the DMA
transfer
    // In this case, timer2

    // USING THE APPROPRIATE TIMER

    //chn 0
    DmaChnSetEventControl(dmaChn_0, DMA_EV_START_IRQ(_TIMER_2_IRQ));

    //chn 1
    DmaChnSetEventControl(dmaChn_1, DMA_EV_START_IRQ(_TIMER_3_IRQ));

    //chn 2
    DmaChnSetEventControl(dmaChn_2, DMA_EV_START_IRQ(_TIMER_4_IRQ));


    // once we configured the DMA channel we can enable it
    // now it's ready and waiting for an event to occur...




  // init the threads
  PT_INIT(&pt_timer);
  PT_INIT(&pt_anim);
  PT_INIT(&pt_dma);

  // init the display
  tft_init_hw();
  tft_begin();
  tft_fillScreen(ILI9340_BLACK);
```

```
  //240x320 vertical display
  tft_setRotation(1); // Use tft_setRotation(1) for 320x240

  // seed random color
  srand(1);

    for(k=0;k<ball_number;k++)
    {
        ball[k].x_pos = int2fix16(309);
        ball[k].y_pos = int2fix16(120);
        ball[k].x_vel = float2fix16(-2);
        ball[k].y_vel = multfix16((((rand() & 0xffff)<<1) - int2fix16(1)),
float2fix16(1.5));
        ball[k].hit_counter = int2fix16(0);
        ball[k].ball_status = 0;
        ball[k].valid_time = 0;
    }


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

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