ECE 4760 Lab 3: TFT Game

Kevin Ying - kzy2
Kristina Nemeth - kan57
Anthony Viego - amv64

Wednesday 4:30PM Lab

Introduction

The purpose of this lab was to implement a video game in which a player controls a paddle which moves back and forth across the bottom of the screen, attempting to catch balls with the paddle in order to prevent balls from reaching the bottom of the screen. The goal was to be able to animate as many balls simultaneously as possible, while modeling all of the collisions between balls and walls, playing sounds for when balls are caught or escape the screen, and maintaining at least 15 frames per second. In order to increase the number of balls we were able to animate, we minimized the amount of information updated on the TFT and utilized a DMA channel in order to play sound with as little load on our main processor as possible.

Design and Testing

Hardware

The primary hardware for this lab was the big board as previously used in the other labs. There are two additional hardware components, a potentiometer used for the control of the paddle in the video game and a switch for the start of the game. The potentiometer would send a signal of a varying resistance to the ADC. This pin would additionally get pulled up as shown in the schematic below. This schematic is shown below in figure 1. The switch used for initiating the start of the game would be connected respectively to ground on one side and to 3.3V through a pull 330 Ohm pull up resistor as shown below in figure 2. The testing of functionality of the switch was done by probing the outputs of each side. The testing of the functionality of the potentiometer was done with the testing of the values of the ADC. For this process, the output was displayed on the screen.

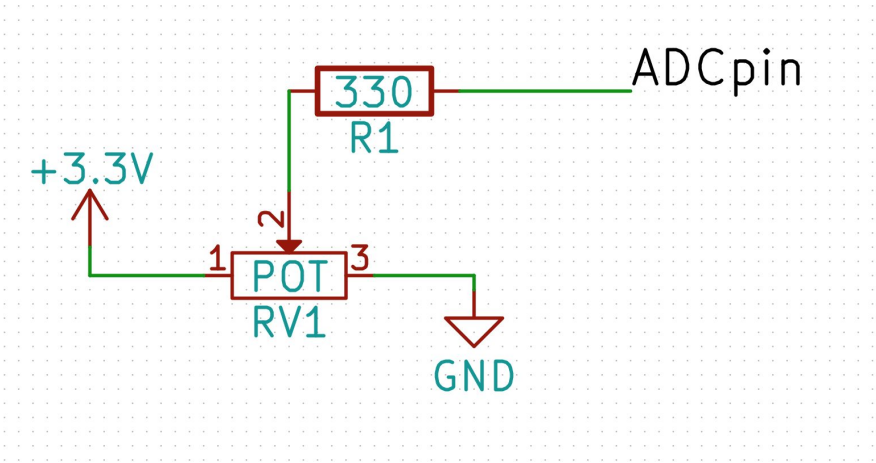

Figure 1: Potentiometer Schematic

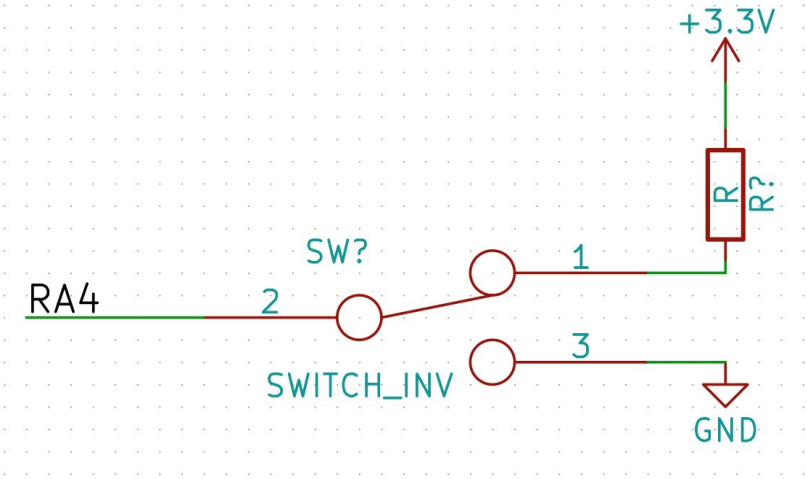

Figure 2: Turn on Switch Schematic

Software

For this lab we decided that we wanted to focus on speed so that we could simulate as many balls as possible while staying above 15fps. To accomplish this we decided that we wanted everything to be handled in a single thread rather than spread out across multiple threads. This was done to save time having to perform context switches when yielding one thread to another. Additionally, we decided to use a singly linked list to implement our balls so that they would be quick to delete and easy to add. Additionally, using a singly linked list eliminated the overhead of having to resize a dynamically sized array.

Ball velocity and position arithmetic was performed using fixed point numbers in order to increase the speed of operations over floating point numbers while maintaining high positional and velocity accuracy.

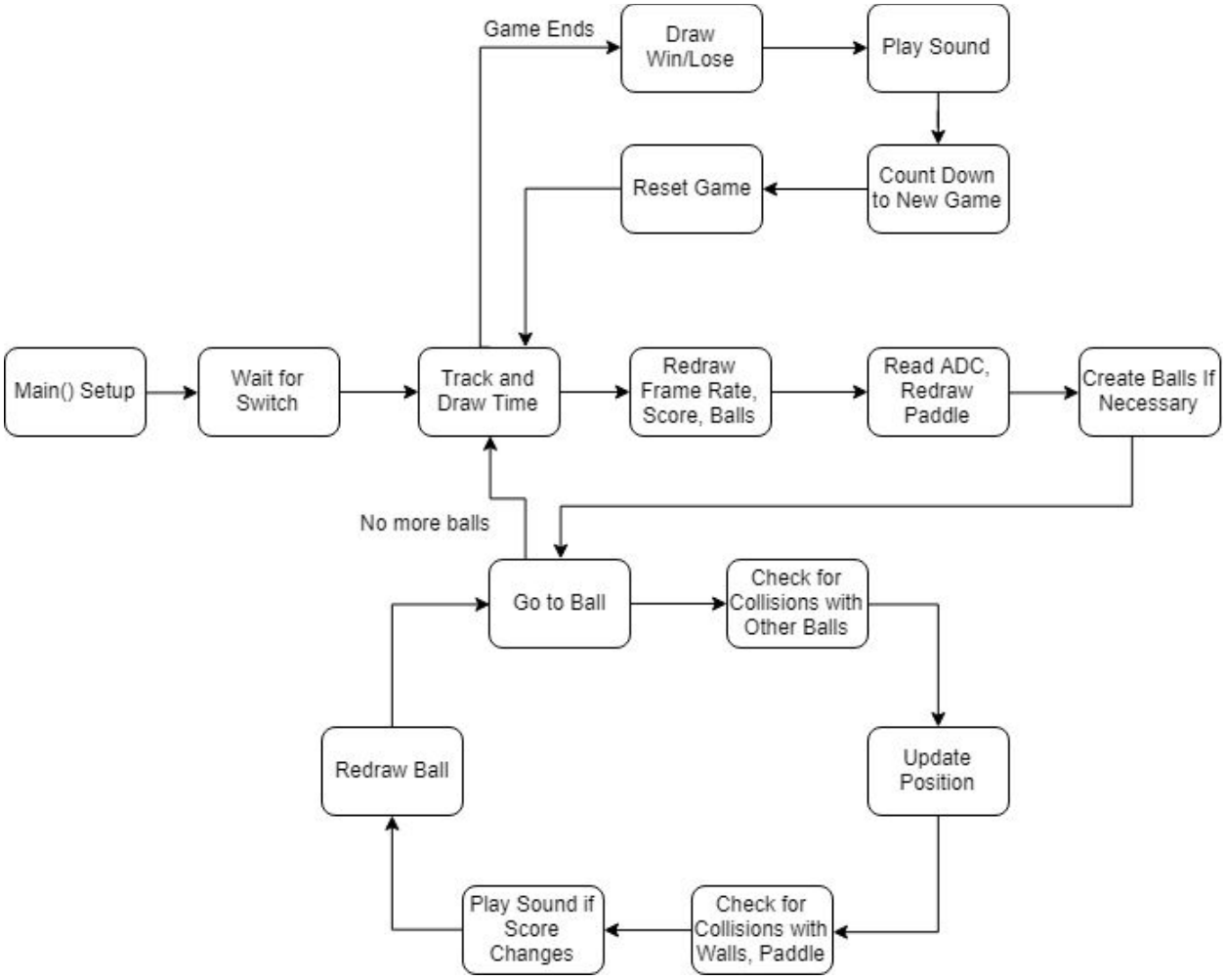

Figure 3: Software Block Diagram

Functions

Main()

Main handles the setup of most of the hardware used in this lab including the ADC, DMA, Timer, and SPI. In addition to this, main also sets up the sine tables for the three sine waves we used to represent the different sounds in our game. The DMA is configured to use channel 0 in auto mode and write the data from our global array DAC_DATA(TYPE) to the spi buffer. DAC_DATA(TYPE) is the bitwise ORed result of the sine tables with the configuration bits for DAC B. We have three DAC DATAS(MINUS, PLUS, OVER) each of which represents the sounds played when we lose a point, gain a point, and finish the game respectively. Main also configures timer23 to have a period of 909 since at a 40,000,000Hz clock this results in a sampling rate of about 44,000. Finally, the main function also initialize our TFT and draws the initial playing field. The main function then displays “toggle switch to start” on the TFT and waits in a while loop until the switch is toggled. By reading a timer based on the time that the switch is

toggled, we can generate a random seed with a high degree of randomness for the random ball velocities generated later. At that point the code enters into the single protothread we have. However, as mentioned we are using a singly linked list as our method of creating, maintaining, and removing the balls, and therefore we have a few helper functions to do that.

struct Balls* createDefault()

This function handles the memory allocation for all of our balls and throws an error if the memory allocation fails. It then calls the resetBall() function which initializes the fields of the Balls struct and finally calls the prepend() function to add the ball to the linked list. In order to make the game more unpredictable, we give each ball a random initial horizontal velocity between -1.5 and 1.5 pixels per frame, while the vertical velocity is a constant 1 pixel per frame downward. Before returning the pointer to the newly created ball it increments our global ball counter variable by 1, which is used to keep track of how many balls we currently have being simulated.

struct Balls* resetBall()

This function handles the initialization of new balls as well as resetting balls that have either left the field of play or have stopped moving. The x position, y position, and x velocity are fixed at 120 pixels , 20 pixels , and 1.5 pixels per frame respectively. The velocity in the y direction is generated randomly by using the formula (rand()) / RAND_MAX) * 3) - 1.5) where the seed for the random function is determined by the system time at which the game started. The function also sets the hit counter of the ball to 0 meaning that it can immediately collide with another ball.

void prepend(struct Balls* ball)

This function simply adds a new ball to the end of the linked list and sets the tail pointer to now point to the newly added ball.

void deleteAll(void)

When the game ends, we need to be able to remove all of the balls to reset the game state for the next round. This function traverses the linked list and frees each element.

Protothread Anim()

Protothread anim is the main chunk of our program and handles everything related to the game. As there are no ISRs and no other threads protothread anim loops infinitely once the game is started. The first portion of what the thread handles is the TFT. During the actual game, system time, FPS, score, and the number of balls have to be displayed on the screen in addition to the game boundaries and barriers, the balls, and the paddle. The system time was calculated by keeping an msecs integer variable, a prevtime variable, and a diffTime variable. At the end of every loop we called PT_GET_TIME() which would give us the system time in msecs and

assigned the value to diffTime - prevTime. This would then give us the total time for that loop in milliseconds. Then at the beginning of each loop we added that time to msecs and also assigned prevtime to equal PT_GET_TIME(). When msecs > 1000 we then increment systime by 1 and subtract 1000 from msecs, so that msecs was always an accurate measure of the current number of milliseconds since the last time increment, and systime the number of seconds since game start.

To calculate the FPS we did 1000/difftime, so for example if our diffTime was 62 ms then our fps was roughly 16. The number of balls on the screen was determined by the create function which incremented the ball count every time a new ball was created. Also, it should be noted that the boundaries and barriers were redrawn every loop to prevent balls from overwriting them. The paddle was also redrawn ever loop based on the adc values read from the potentiometer. To calculate the x position of the paddle we used the formula x = (adc_9/1023.0)*(190) where adc_9 is the result of reading values from the adc. We divided the adc value by 1023 to account for the resolution of the adc and then multiplied by 190 to factor in the width of the rectangle so that when at the maximum x value, the rectangle would not go off the screen. Something to notice is that all of this has to be drawn before we even handle the re-drawing of each ball and so we needed to figure out some way to handle this efficiently.

To accomplish this while being efficient as possible we employed a number of different tactics. For starters, we realized that having to draw a black rectangle to cover up the previous text before writing the next text was inefficient due to the large amount of extra pixels that needed to be updated. It also tended to increase the visible amount of flickering on the TFT. To get around this we used a prevscore, prevtime, and prevfps variable. We then wrote the previous versions in black to the TFT before writing the current versions in yellow which allowed us to cut down on the number of pixels being updated and saving us time. Additionally we used fast horizontal lines for the paddle rather than a rectangle in hopes of further improving performance. While these may have been minor improvements, but implementing them we managed to increase our ball count by roughly 20 balls while managing to stay above 15 fps.

The other major part of protothread anim was the handling of ball creation, ball collisions, resetting balls, and redrawing balls. To create new balls we used a simple hard coded maximum of 145 (resulting from testing of an upper limit) and stated that if ballcount is less than 145 then we create a new ball and prepend it to the singly linked list. Combined with our random y velocity being set when we create a new ball, this resulted in a spraying of balls upon the start of the game. For ball collisions we used a for loop with a while loop nested inside of it. For every ball, we looped through the following balls in the linked list until the next pointer became null. For each ball we checked to see if a collision occurred by checking to see if the x distance and y distance between the balls were both less than 4 pixels.

If a collision occurs we then checked to see if the hit counter was 0, if not we ignored the collision and continued(meaning we stopped that iteration of the loop and proceeded to check the next ball if there was one). If hitcounter was 0 then we checked to see if the square distance between the balls was less than one. If it was then the balls were practically inside of each and

we simply assigned the velocities of each ball to the other. If the balls were not inside of each other then we first calculated the multiplication of the difference in x position and the difference in x velocities + the multiplication of the difference in y positions plus the difference in y velocities. We then multiplied that result by the result from our inverse lookup table and set that equal to a variable called dotProd. Finally we set deltaVx equal to the product of dotProd and the difference in x positions and set deltaVy equal to the product of dotProd and the difference in y positions. We then updated the velocities of each ball by subtracting deltaVx and Vy from the ball found in the inner loop, and adding deltaVx and deltaVy to the ball from the outer loop. We then finished out velocity calculation for the outer loop ball by multiplying the x and y velocities by the drag coefficient and then subtracting the result from the velocities.

We then erased the ball by drawing a black circle based on the x and y positions and then updated the x and y positions based on the newly calculated velocities. We then checked to see if the ball would collide with any surface by seeing if the x or y positions was within 4 pixels of a wall of barrier. If it would then we flipped the x or y velocity depending on which surface it would collide with and set the x or y position to be 4 pixels from that surface. In theory, this would prevent us from having to redraw the boundaries, although we sometimes noticed that our boundaries would be overwritten in some early tests so we continued to redraw boundaries every frame to be safe. We then re-draw the ball using fillcircle at the new position.

If the ball collided with the paddle we increased the score by one and called resetBall. On the other hand if the ball passed through the bottom of the screen without hitting the paddle then we decremented the score by 1 and called resetBall.

The final thing protothread anim handled was the end of game screen and the playing of sound. For the sound we use the DMA to send information to the spi buffer which then sent that information to the DAC. To ensure that plus score and minus score sounds only lasted for a small amount of time, we used a variable called turnOffTime which we set equal to the current time, PT_GET_TIME(). We then added a number of milliseconds to this time that we wanted the sound to last for and compared it to prevTime at the beginning of each loop. If prevTime >= turnOffTime + (amount) then we aborted the DMA transfer until the next plus or minus in score.

For the winning and losing screen we wrapped the entire game handling code in a while loop with the condition that (systime < 30). Once systime exceeded 30 seconds the game ended and we enabled the DMA, set systime to 0, and set turnOffTime to be 1000 which resulted in sound being played for one second. If the final score was above 0 at 30 seconds then we printed YOU WIN! on the screen and if not we printed YOU LOSE!. In both cases we also reset the score, prevNumBalls, and prevScore so that we did not reprint those values once the game restarted. We also set the game to restart automatically after 5 seconds by printing Next game in (time) and then yielding for 1000s in between each number to give a countdown for the next game.

Testing

To test the code implemented, we added the code at incremental stages to make sure that each aspect of the code functioned separately. Initially we tested out the functionality of the paddle corresponding to the ADC. To test this, we drew the paddle on the screen and printed the ADC values onto the screen to ensure that it was acting as expected. To test ball dynamics, we initially implemented the graphical implementation of only one ball and how it bounced off of the walls. Following this we implemented two balls on the screen, and their collisions with each other then four balls on the screen. At this stage we gradually continued to increase the number of balls on the screen to detect potential failure states. At this stage we additionally tested the maximum number of balls we could simulate on the screen while consistently having at least 15 FPS, by having text appear on the screen when the FPS dropped below 15. Following this we separately tested the functionality of DMA by initially just playing one sound. We then implemented the remaining sounds and following that integrated that into the remainder of the game.

Results

We were able to simulate 145 balls with a frame rate that was consistently at least 15 FPS, and sometimes were able to play games with up to 155 balls simultaneously, depending upon the particular starting circumstances of the game. Most of the time, at 145 balls, the frame rate was 16 FPS, or one iteration of all of the game updates every 62ms. All of the ball collisions were accurate enough to be visually plausible and balls were caught by the player-controlled paddle in a predictable manner. There was also little visual tearing, except for on the time, which was only updated once per second, and almost no flickering of the displayed text. Our implementation also allowed for the entire game screen to be used, with the text over parts of the playing field.

Also, the audio outputs were crisp and distinct frequencies which were at approximately the same volume, providing a pleasant listening experience. We were able to have all of our audio outputs be synthesized at 44 Ksamples/second.

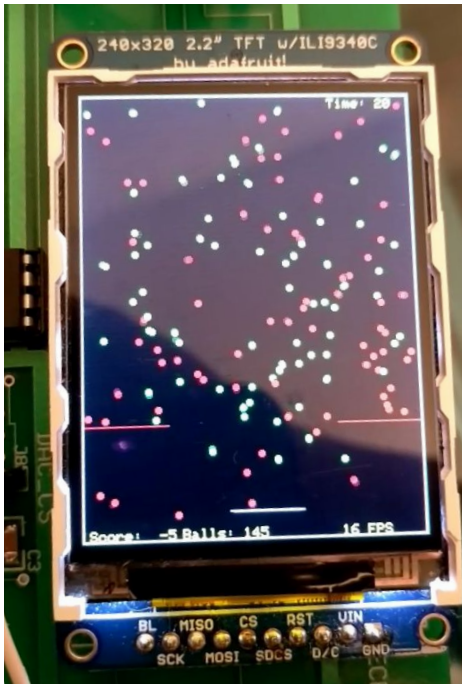

Figure 4: Photo of TFT Display at 145 Simultaneous Balls

One thing that could be improved is that occasionally, pairs of balls which spawn at the same time or collide at very specific velocities will become entangled with each other until a third ball collides with the pair and causes the balls to separate again. Visual inspection of the TFT screen during gameplay suggests that this happens to 1-2 pairs of balls at any time on average, or an approximate error rate of 3/145 ≅ 2% of the balls. However, most of the time this error rate would only result in one or two pairs of balls which were entangled in this way reaching the bottom of the screen, where a player's focus would be while playing the game, and therefore these errors were generally not noticed.

Conclusion

Throughout this lab, we focused more on implementation of graphics than in previous labs due to the creation of a game. This lead us to learn several things which we previously had not focused on in the other labs. An essential part of the lab was the dynamic movement of the balls on the screen to ensure that they accurately reflects the laws of physics on the screen in terms of general movement of the balls and the collisions of the balls with each other and with other . This also included the relation of a physical movement of the turning of the potentiometer to the moving of the paddle of the screen. Following this, an essential part of the lab was focusing on implementing an efficient solution so that as many balls could be generated on the

screen while keeping the FPS of at least 15. The final focus of this lab was the usage of DMA SPI to produce sound effects.

Overall we faced several small issues with the implementation of the location of the paddle properly corresponding to the ADC values which were fixed by fixing the calculation surrounding the location of the ADC value and proper usage of data types. We primarily debugged this by displaying the values causing the paddle to have irregular motion. Another error we faced throughout the lab is the implementation of the sound through DMA. One issue was that the sine table would not get run through completely creating a jump in our displayed sine wave and harsh noises. Therefore, we had to modify our frequency to fit the size of our precomputed tables.

In the future, we could further improve the ball collision detection mechanics in order to further decrease the number of artifacts on screen. A potential improvement or next level of this lab could be the addition of a second player with a paddle to allow for a competitive mode.

Appendix

Full Code

```
////////////////
////////////////
////
                    // clock AND protoThreads configure!
                    // You MUST check this file!
                    #include "config.h"
                    // threading library
                    #include "pt_cornell_1_2_3.h"
                    #include <math.h>
                    //#include <plib.h>


                    ////////////////////////////////////
                    // graphics libraries
                    // SPI channel 1 connections to TFT
                    #include "tft_master.h"
                    #include "tft_gfx.h"
                    // need for rand function
                    #include <stdlib.h>
                    #include <time.h>
```

```
////////////////////////////////////


////////////////////////////////////
// pullup/down macros for keypad
// PORT B
#define EnablePullDownB(bits) CNPUBCLR=bits; CNPDBSET=bits;
#define DisablePullDownB(bits) CNPDBCLR=bits;
#define EnablePullUpB(bits) CNPDBCLR=bits; CNPUBSET=bits;
#define DisablePullUpB(bits) CNPUBCLR=bits;
//PORT A
#define EnablePullDownA(bits) CNPUACLR=bits; CNPDASET=bits;
#define DisablePullDownA(bits) CNPDACLR=bits;
#define EnablePullUpA(bits) CNPDACLR=bits; CNPUASET=bits;
#define DisablePullUpA(bits) CNPUACLR=bits;
////////////////////////////////////


//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)


//#define RAND_MAX 1.5
////////////////////////////////////
// some precise, fixed, short delays
// to use for extending pulse durations on the keypad
// if behavior is erratic
#define NOP asm("nop");
// 1/2 microsec
#define wait20
NOP;NOP;NOP;NOP;NOP;NOP;NOP;NOP;NOP;NOP;NOP;NOP;NOP;NOP;NOP;NOP;NO
P;NOP;NOP;
// one microsec
#define wait40 wait20;wait20;
////////////////////////////////////
```

```
/* 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
****************************************************/


// quick inverse table for distances
fix16 inverseTable[16];


// string buffer
char buffer[60];
char keypadbuffer[128];
float testbuffer[7] = {697, 770, 852, 941, 1209, 1336, 1477};
// DDS sine table
#define sine_table_size 256
volatile int sin_table[sine_table_size];
unsigned int DAC_DATA_OVER[sine_table_size];
unsigned int DAC_DATA_PLUS[sine_table_size];
unsigned int DAC_DATA_MINUS[sine_table_size];
////////////////////////////////////
// DAC ISR
// A-channel, 1x, active
#define DAC_config_chan_A 0b0011000000000000
// B-channel, 1x, active
#define DAC_config_chan_B 0b1011000000000000
#define Fs 44100.0
```

```
#define two32 4294967296.0 // 2^32
//== Timer 2 interrupt handler
=========================================
//volatile unsigned int DAC_data; // output value
volatile SpiChannel spiChn = SPI_CHANNEL2; // the SPI channel to use
volatile int spiClkDiv = 2; // 20 MHz max speed for this DAC
// the DDS units:
static float Fout = 660;
static float Fout2 = 770;
volatile unsigned int phase_accum_main, phase_incr_main;//
// profiling of ISR
volatile int isr_time;


static int ballcount = 0;


static struct pt DAC, anim;


struct Balls{
    fix16 xc;
    fix16 yc;
    fix16 vxc;
    fix16 vyc;
    int hitcounter;
    struct Balls* next;
} typedef ball_struct;
static struct Balls *head = NULL;
static struct Balls *tail = NULL;


struct Balls* createDefault(void) {
   struct Balls *new_ball = (struct Balls*)malloc(sizeof(struct
Balls));
   if (new_ball == NULL){
       //write something to TFT
       // error!
   }

   resetBall(new_ball);
   new_ball->next = NULL;
   if (head == NULL) {
       head = new_ball;
       tail = new_ball;
```

```
    } else {
        prepend(new_ball);
    }

    ballcount = ballcount + 1;
    return new_ball;
}


struct Balls* create(int xc, int yc, float vxc, float vyc){
    struct Balls *new_ball = createDefault();

    new_ball->xc = int2fix16(xc);
    new_ball->yc = int2fix16(yc);
    new_ball->vxc = float2fix16(vxc);
    new_ball->vyc = float2fix16(vyc);

    return new_ball;
}


void prepend(struct Balls* ball){
    tail->next = ball;
    tail = ball;
}


void resetBall(struct Balls* ball){
    ball->xc = int2fix16(120);
    ball->yc = int2fix16(20);
    ball->vxc = float2fix16((((float)(rand()) / RAND_MAX) * 3) - 1.5);
    ball->vyc = float2fix16(1.5);
    ball->hitcounter = 0;
}


void deleteAll(void){
    if (!head)
        return;
    while(head->next) {
        struct Balls *temp;
        temp = head->next;
        free(head);
        head = temp;
```

```
    }

    free(head);
    ballcount = 0;

    head = NULL;
    tail = NULL;
}


// === Animation Thread =============================================
// update a 1 second tick counter
static fix16 xc=int2fix16(10), yc=int2fix16(150), vxc=int2fix16(2),
vyc=int2fix16(5);
static fix16 g = float2fix16(0.1), drag = float2fix16(.001);
static float speedx = 1.5, speedy = -1.5;
static PT_THREAD (protothread_anim(struct pt *pt))
{
    PT_BEGIN(pt);

    static int timeSinceBall = 0;
    static int score = 0;
    static int prevScore = 0;
    static int prevFrameRate = 15;
    static int frameRate = 15;
    static int prevNumBalls = 0;


    static int prevTime = 0;
    static int sys_time_seconds = 0;
    static int msec = 0;
    static int turnOffTime = 1000;

    // paddle
    static int adc_9;
    static fix16 ADC_scale ;
    static short x_value;

    ADC_scale = float2fix16(3.3/1023.0); //Vref/(full scale)
    while(1) {
      tft_fillScreen(ILI9340_BLACK);

      static int diffTime = 67;
```

```
    while(sys_time_seconds < 30) {

      // Draw current time
      msec += diffTime;
      prevTime = PT_GET_TIME();
      if (msec > 1000) {
          tft_setCursor(180, 3);
          tft_setTextSize(1);

          // erase previous score
          tft_setTextColor(ILI9340_BLACK);
          sprintf(buffer, "Time: %d", sys_time_seconds);
          tft_writeString(buffer);

          sys_time_seconds++;
          msec -= 1000;

          // write current score
          tft_setCursor(180, 3);
          tft_setTextColor(ILI9340_YELLOW);
          sprintf(buffer, "Time: %d", sys_time_seconds);
          tft_writeString(buffer);
      }

      if (prevTime >= turnOffTime + 100) {
          DmaChnAbortTxfer(0);
      }

      // Draw boundaries
      tft_drawLine(0, 240, 60, 240, ILI9340_RED);
      tft_drawLine(180, 240, 240, 240, ILI9340_RED);
      tft_drawRect(0, 0, 240, 320, ILI9340_WHITE);

      // ******** DRAWING PADDLE ************
      // 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

      // Un-draw then re-draw paddle
      tft_drawFastHLine(x_value, 300, 50,ILI9340_BLACK);
```

```
        x_value = (adc_9/1023.0)*(190); //updated to be 240-50 to
account for width of rectangle
        tft_drawFastHLine(x_value, 300, 50,ILI9340_WHITE);

        // Draw frame rate
        frameRate = (int)((float)1000/diffTime);
        if (frameRate != prevFrameRate) {
            tft_setCursor(180, 312);
            tft_setTextSize(1);

            // erase previous score
            tft_setTextColor(ILI9340_BLACK);
            sprintf(buffer, "%d FPS", prevFrameRate);
            tft_writeString(buffer);

            // write current score
            tft_setCursor(180, 312);
            tft_setTextColor(ILI9340_YELLOW);
            sprintf(buffer, "%d FPS", frameRate);
            tft_writeString(buffer);

            prevFrameRate = frameRate;
        }

        // Draw score and number of balls
        if (score != prevScore || ballcount != prevNumBalls) {
            tft_setCursor(5, 312);
            tft_setTextSize(1);

            // erase previous score
            tft_setTextColor(ILI9340_YELLOW);
            tft_writeString("Score: ");
            tft_setTextColor(ILI9340_BLACK);
            sprintf(buffer, "%3d", prevScore);
            tft_writeString(buffer);
            tft_setTextColor(ILI9340_YELLOW);
            tft_writeString("\tBalls: ");
            tft_setTextColor(ILI9340_BLACK);
            sprintf(buffer, "%3d", prevNumBalls);
            tft_writeString(buffer);

            // write current score
            tft_setCursor(5, 312);
```

```
            tft_setTextColor(ILI9340_YELLOW);
            sprintf(buffer, "Score: %3d\tBalls: %3d", score,
ballcount);
            tft_writeString(buffer);

            prevScore = score;
            prevNumBalls = ballcount;
        }

        if (ballcount < 145){
            if (timeSinceBall)
                timeSinceBall--;
            else {
                struct Balls *ball = createDefault();
                timeSinceBall = 2;
            }
        }

        struct Balls *outer = head;
        int i = 0;
        for (; i < ballcount; ++i){
            if (outer == NULL){
                break;
            }
            if (outer->hitcounter) {
                outer->hitcounter--;
            } else {
                struct Balls *inner = outer;
                while (inner->next) {
                    inner = inner->next;
                    fix16 deltax = inner->xc - outer->xc;
                    fix16 deltay = inner->yc - outer->yc;
                    if ((deltax < int2fix16(4) && deltax >
int2fix16(-4))
                            && (deltay < int2fix16(4) && deltay >
int2fix16(-4))){
                        if (inner->hitcounter) {
                            continue;
                        }
                        fix16 squareDist = multfix16(deltax, deltax) +
                                multfix16(deltay, deltay);
                        if (squareDist > int2fix16(16)) {
                            continue;
                        }
```

```
                        outer->hitcounter = 2;
                        if (squareDist < int2fix16(1)) {
                            fix16 tempVx = inner->vxc;
                            fix16 tempVy = inner->vyc;
                            inner->vxc = outer->vxc;
                            inner->vyc = outer->vyc;
                            outer->vxc = tempVx;
                            outer->vyc = tempVy;
                            continue;
                        }
                        fix16 dotProd = multfix16(deltax, inner->vxc -
outer->vxc)
                                + multfix16(deltay, inner->vyc -
outer->vyc);
                        dotProd = multfix16(dotProd,
inverseTable[fix2int16(squareDist)]);
                        fix16 deltaVx = multfix16(dotProd, deltax);
                        fix16 deltaVy = multfix16(dotProd, deltay);


                        // update velocities
                        inner->vxc -= deltaVx;
                        inner->vyc -= deltaVy;
                        outer->vxc += deltaVx;
                        outer->vyc += deltaVy;

                        break;
                    }
                }
            }

            // erase disk
            tft_fillCircle(fix2int16(outer->xc), fix2int16(outer->yc),
2, ILI9340_BLACK); //x, y, radius, color
            // compute new velocities
            vyc = vyc - multfix16(vyc, drag) ;
            vxc = vxc - multfix16(vxc, drag);


            outer->xc = outer->xc + outer->vxc;
            outer->yc = outer->yc + outer->vyc;


            if (outer->xc<int2fix16(4)){
                outer->vxc = -outer->vxc;
```

```
            outer->xc = int2fix16(4);
        } else if (outer->xc>int2fix16(236)){
            outer->vxc = -outer->vxc;
            outer->xc = int2fix16(236);
        }
        if (outer->yc>int2fix16(316)) { //ADD AND condition to
check if its also not in the x_value used for the rectangle capture
            resetBall(outer);
            timeSinceBall = 2; // reset timer for ball so that two
balls don't spawn at same time
            score--;
            // Play sound
            DmaChnAbortTxfer(0);
            DmaChnSetTxfer(0, DAC_DATA_MINUS, (void*)&SPI2BUF, 512,
2, 2);
            DmaChnEnable(0);
            turnOffTime = PT_GET_TIME();
        } else if (outer->yc<int2fix16(4)){
            outer->vyc = -outer->vyc;
            outer->yc = int2fix16(4);
        }
        if ((outer->yc<int2fix16(243) && outer->yc>int2fix16(237))
&&
        (outer->xc<=int2fix16(60) || outer->xc>=int2fix16(180))){
            outer->vyc = -outer->vyc;
            if (outer->vyc < 0)
                outer->yc = int2fix16(236);
            else
                outer->yc = int2fix16(244);
        }


        if ((outer->yc<int2fix16(303) && outer->yc>int2fix16(297))
&&
        (outer->xc<=int2fix16(x_value+50) &&
outer->xc>=int2fix16(x_value))){
            resetBall(outer);
            timeSinceBall = 2; // reset timer for ball so that two
balls don't spawn at same time
            score++;
            // Play sound
            DmaChnAbortTxfer(0);
            DmaChnSetTxfer(0, DAC_DATA_PLUS, (void*)&SPI2BUF, 512,
2, 2);
            DmaChnEnable(0);
            turnOffTime = PT_GET_TIME();
```

```
            }
            //  draw disk
            if (i % 2) {
                tft_fillCircle(fix2int16(outer->xc),
fix2int16(outer->yc), 2, ILI9340_GREEN); //x, y, radius, color
            } else {
                tft_fillCircle(fix2int16(outer->xc),
fix2int16(outer->yc), 2, ILI9340_RED); //x, y, radius, color
            }


            outer = outer->next;

            diffTime = PT_GET_TIME() - prevTime;
            if (diffTime > 67) {
                tft_setCursor(30, 80);
                //tft_fillScreen(ILI9340_BLACK);
                tft_setTextSize(4);
                tft_setTextColor(ILI9340_WHITE);
                tft_writeString("FAIL");
                PT_YIELD_TIME_msec(1000);
            }
          }
         // PT_YIELD_TIME_msec(33 + PT_GET_TIME() - startTime);
        } // END WHILE(time < GAME_END)

        //tft_fillCircle(100, 100, 50, ILI9340_WHITE);
        // GAME RESTART CODE
        DmaChnAbortTxfer(0);
        DmaChnSetTxfer(0, DAC_DATA_OVER, (void*)&SPI2BUF, 512, 2, 2);
        DmaChnEnable(0);
        tft_fillScreen(ILI9340_BLACK);
        tft_setTextSize(4);
        tft_setTextColor(ILI9340_WHITE);
        if (score >= 0) {
            tft_setCursor(30, 80);
            tft_writeString("YOU WIN!");
        } else {
            tft_setCursor(20, 80);
            tft_writeString("YOU LOSE!");
        }

        deleteAll();
```

```
      tft_setTextSize(2);
      tft_setTextColor(ILI9340_WHITE);
      tft_setCursor(30, 160);
      sprintf(buffer, "Final Score: %d", score);
      tft_writeString(buffer);

      tft_setCursor(30, 190);
      tft_writeString("Next game in ...");

      tft_setCursor(30, 220);
      tft_writeString("5  ");
      PT_YIELD_TIME_msec(1000);
      DmaChnAbortTxfer(0);
      tft_writeString("4  ");
      PT_YIELD_TIME_msec(1000);
      tft_writeString("3  ");
      PT_YIELD_TIME_msec(1000);
      tft_writeString("2  ");
      PT_YIELD_TIME_msec(1000);
      tft_writeString("1");
      PT_YIELD_TIME_msec(1000);

      timeSinceBall = 0;
      score = 0;
      prevScore = 0;
      prevNumBalls = 0;

      prevTime = PT_GET_TIME();
      sys_time_seconds = 0;
      msec = 0;
      turnOffTime = 1000;

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


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


   ANSELA = 0;
```

```
   ANSELB = 0;


   // set up DAC on big board

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




   // define setup parameters for OpenADC10
   // Turn module on | output 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 peripheral 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 as analog input
   #define PARAM4     ENABLE_AN11_ANA || ENABLE_AN10_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
   SetChanADC10( ADC_CH0_NEG_SAMPLEA_NVREF | ADC_CH0_POS_SAMPLEA_AN11
|
           ADC_CH0_NEG_SAMPLEB_NVREF | ADC_CH0_POS_SAMPLEA_AN10);
   // configure to sample AN11
   OpenADC10( PARAM1, PARAM2, PARAM3, PARAM4, PARAM5 );
   // configure ADC using the parameters defined above


   EnableADC10(); // Enable the ADC


   // SCK2 is pin 26
   // SDO2 (MOSI) is in PPS output group 2, could be connected to RB5
which is pin 14
   PPSOutput(2, RPB5, SDO2);
   PPSOutput(4, RPB10, SS2);


   // control CS for DAC
   //ConfigIntTimer2(T2_INT_ON | T2_INT_PRIOR_2);
   mPORTBSetPinsDigitalOut(BIT_4 | BIT_10);
   mPORTBClearBits(BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5);
   mPORTBSetPinsDigitalOut(BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 |
BIT_5 | BIT_7 );    //Set port B as output
   mPORTBSetBits(BIT_4);

   mPORTAClearBits(BIT_0);          //Clear A bits to ensure light is
off.

   mPORTBSetPinsDigitalIn(BIT_14);


   // divide Fpb by 2, configure the I/O ports. Not using SS in this
example
   // 16 bit transfer CKP=1 CKE=1
   // possibles SPI_OPEN_CKP_HIGH;   SPI_OPEN_SMP_END;
SPI_OPEN_CKE_REV
   // For any given peripherial, you will need to match these
   // clk divider set to 2 for 20 MHz
   // end DAC setup
   OpenTimer23(T2_ON | T2_SOURCE_INT | T2_PS_1_1, 910);
```

```
    mT2ClearIntFlag(); // and clear the interrupt flag



    // === 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(&DAC);
    //PT_INIT(&adc);
    PT_INIT(&anim);


    // init the display
    // NOTE that this init assumes SPI channel 1 connections
    tft_init_hw();
    tft_begin();
    tft_fillScreen(ILI9340_BLACK);
    //240x320 vertical display
    tft_setRotation(0); // Use tft_setRotation(1) for 320x240
    tft_drawLine(0, 240, 60, 240, ILI9340_RED);
    tft_drawLine(180, 240, 240, 240, ILI9340_RED);
    tft_drawRect(0, 0, 240, 320, ILI9340_WHITE);

    int j = 0;
    for (j; j < sine_table_size; ++j){
         sin_table[j] =
(int)(2047*sin((float)j*6.283/(float)sine_table_size));
    }
    phase_accum_main;
    phase_incr_main = (int)(Fout*(float)two32/Fs);
    for(j=0; j < sine_table_size; ++j){
        DAC_DATA_PLUS[j] =
(int)(2047*sin((float)j*12.566/(float)sine_table_size));
        DAC_DATA_PLUS[j] = DAC_config_chan_B | (DAC_DATA_PLUS[j] +
2048);
        DAC_DATA_OVER[j] =
(int)(511*sin((float)j*12.566*2.0/(float)sine_table_size));
```

```
        DAC_DATA_OVER[j] = DAC_config_chan_B | (DAC_DATA_OVER[j] +
2048);
        DAC_DATA_MINUS[j] =
(int)(511*sin((float)j*12.566*4.0/(float)sine_table_size));
        DAC_DATA_MINUS[j] = DAC_config_chan_B | (DAC_DATA_MINUS[j] +
2048);
    }

    SpiChnOpen(spiChn,SPI_OPEN_ON | SPI_OPEN_CKE_REV | SPI_OPEN_MODE16
| SPICON_FRMEN | SPI_OPEN_MSTEN | SPICON_FRMPOL, 2);
    DmaChnOpen(0, 0, DMA_OPEN_AUTO);
    DmaChnSetTxfer(0, DAC_DATA_MINUS, (void*)&SPI2BUF, 512, 2, 2);
    DmaChnSetEventControl(0, DMA_EV_START_IRQ(_TIMER_3_IRQ));
    //DmaChnEnable(0);

    int num;
    inverseTable[0] = 1;
    for (num=0; num < 16; num++){
        inverseTable[num] = divfix16(int2fix16(1), int2fix16(num+1));
    }

    // generate random seed
    int curr_switch = mPORTAReadBits(BIT_4);
    tft_setCursor(20, 80);
    tft_setTextSize(2);
    tft_setTextColor(ILI9340_WHITE);
    tft_writeString("Toggle switch \n  to start game");
    while(mPORTAReadBits(BIT_4) == curr_switch) {};
    srand(PT_GET_TIME());

    // round-robin scheduler for threads
    while (1) {
        //disabled protothreads for color and animation
        //PT_SCHEDULE(protothread_timer(&timer));
        //PT_SCHEDULE(protothread_adc(&adc));
        PT_SCHEDULE(protothread_anim(&anim));
    }
} // main


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