DDR

Introduction

Dance Dance Revolution (DDR) is a classic rhythm game with intuitive controls. There are four possible inputs - up, down, left and right. The user needs to input the correct direction when the corresponding 'note' reaches the line, also known as the 'nowbar' (Figure 1). The user inputs these arrow by stepping on the correct button on a platform/pad that is placed on the floor. (Figure 2)

One downside of DDR is that the game requires the arcade console (platform), which most people would not have access to, especially during the pandemic. This project aims to create a more accessible version of such a console using simple materials and the PSoC to allow players to access this game without the console.

Overview

This project will be split into three main components - the controller, game body and music player - according to the hardware flowchart (Appendix 0.4). The controller will be responsible for taking in four digital input lines from four different switches and debouncing the signal. The state of the controller would then be outputted along with an interrupt to indicate state changes to the game body. The controller will be created with the 8051. The game body will be responsible for progressing the game state, displaying the state onto a TFT screen and signal the music player according to the software flowchart (Appendix 0.5). The music player will a hard coded music piece when it is signalled to do so. The game body will be created with the PSoC board and the music player will be created with the PSoC stick.

Figure 1: DDR UI

Figure 2: DDR gameplay

Controller

Hardware

To imitate the large surface area that players can step on in the console, aluminum foil was used to create conducting pads to create switches with a large surface ares. The floor pads were pulled up to 5V by 10k pull-up resistors, while the foot pads (taped under socks) are grounded (Figure 4). Together this creates a very responsive, but bouncy set of pull-up switches which are grounded when the foot pad contacts the floor pad(Figure 3). Each pad is then connected to one of P1.0-P1.3 on the 8051, where it will be debounced.

Software

There are two main concerns regarding reading the switch states directly from the game body. Firstly, since the switches are made up of aluminum foil coming in contact with each other, it is extremely bouncy due to the roughness of the contact surfaces and lack of pressure on the pads. Secondly, the game body would not be able to detect state changes of the switches, which means it would have to poll the switches in regular intervals that are relatively small, causing major delays in game physics.

The 8051 would solve both of the above problems by acting as an encoder that processes the raw switch states. The 8051 takes in raw switch inputs through P1.0-P1.3, then it would mask out the upper nibble with 0. It then compares the new

Figure 3: Physical appearance of pads

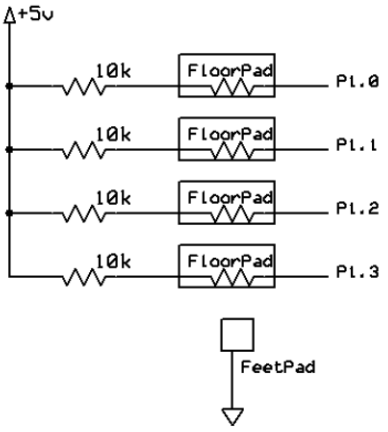


Figure 4: Schematic of pads

state with the previous button state, if they are the same, nothing is done and the routine loops. If they are different it stores this new state as old state, then SWAP the nibbles and mask off the LSB with 1s; this allows the snapshot of the state be stored in the MSB nibble. This snapshot would then be written to P1 and at the same time P3.4 would be pulsed, which would signal a state change to the PSoC. The LSBs have to be masked off with 1s so that those pins in P1 can continue to be used as inputs, while the MSB would contain a snapshot of the switch state for the PSoC regardless of how much the switch state fluctuates afterwards. Finally there is a delay of 20ms to debounce after a state change, then the loop continues. This code is able to detect state changes within 12us of the change, making it extremely responsive. It is also able to handle up to 50 state changes per second (debouncing). For code, refer to Appendix 0.1.

Figure 5 demonstrates this behaviour. The blue signal being a pin from P1.0-P1.3 and the yellow signal represents P3.4, or the state change indicator. In the figure, P3.4 gets pulsed high whenever there is a change in state of this specific pin, allowing access to snapshots of the switch states when it changes. The P3.4 signal here lasts for the entire debouncing period for demonstration purposes, while in reality P3.4 only pulses for 1 machine cycle.

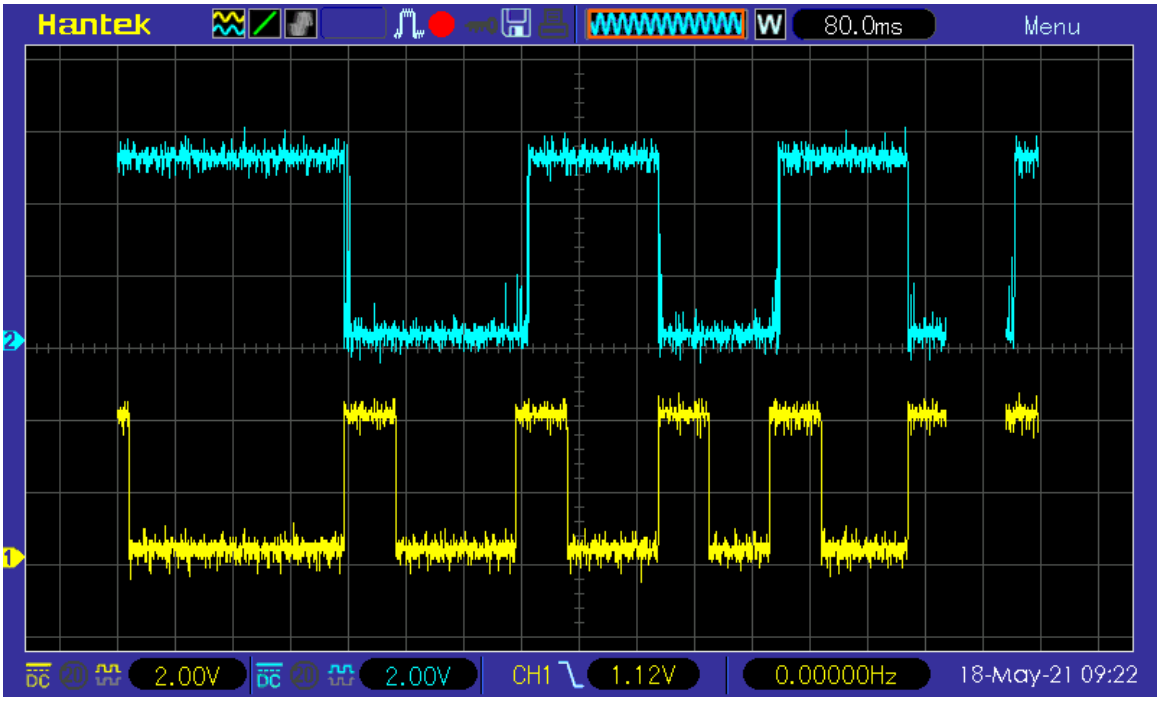


Figure 5: Controller input(blue) and P3.4 output(yellow)

Music Player

Hardware

The music player is created using a PSoC stick, a speaker and a LM386. They are wired according to Figure 6. The LM386 is powered by a 12V power source and configured such that it produces a 20x amplification of the input signal, which comes from pin 1.6 of the stick. The 0.05uF capacitor acts as a low pass filter and the 250uF capacitor blocks the DC voltage from the amplifier. The stick receives inputs from the psoc on pin 2.0, this would act as an interrupt pin that starts the playing of music. There is also a 10k potentiometer that allows the amplitude of the input signal coming from the stick to be reduced. This is important as the sound waveform coming from the VDAC in the stick has a range up to 1.020V (Figure 7), while the amplified sound should be within 12V (supply voltage into LM386) to avoid clipping of the waveform. The potentiometer here would reduce the amplitude range to around 500mV, such that the amplified waveform has a peak of around 10V, which will not be clipped.

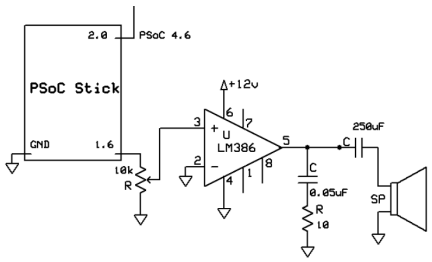


Figure 6: Schematic of stick

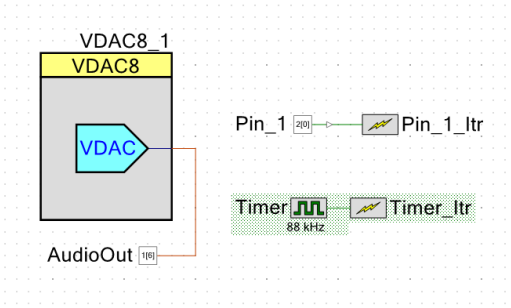


Figure 7: Creator schematic of stick

Software

Interrupts

The stick has two interrupts: the timer interrupt which is called at a frequency of 88000Hz according to a clock and a pin interrupt that gets called on a falling edge in P2.0. The stick also has an internal time tracker which is a long that gets incremented by 1 every time a timer interrupt is called. As a result, this internal timer/88000 is the time elapsed after 'game start' in seconds. The pin interrupt would initialize the internal time tracker to -3*88000, or -3 seconds, and start the clock, which would start producing timer interrupts. As a result, this allows synchronization between

the PSoC and the stick via this timer, as this pin interrupt would mark 3 seconds before the game 'begins'.

Music encoding

Since wav files are generally large, the stick uses a hard coded format of music which involves storing the frequencies of the notes playing at each time, such that these notes can be played as sine waves. The format of the song is an array of size 2 arrays, with each size 2 array representing a half beat in the song. The first element of a note array is the frequency of the note being played, while the second element represents whether this note is a 'new' note (encoded as 1), which would be played with a short silence in the beginning, or an 'old' note (encoded as 0) which would be played continuously. This encoding scheme allows the stick to play consecutive notes with the same frequency (440,1,440,1) and notes longer than half a beat (440,1,440,0).

Once the timer starts, the main loop of the code would first calculate the corresponding scaled integer value of the waveform at that time by using the sine function from math.h, then write it to the VDAC which outputs an analog value to the LM386, then the speaker. By separating the incrementing of the timer and the main loop, this allows for room for error such as when the main loop is behind, causing some time points to be skipped, or when the timer is behind, causing the main loop to output the same value multiple times. This is important since the sine function, time interrupts and discrete note checking takes non-zero time, it is expected that the stick would not be able to produce sine waves that are too high in frequency, since the update rate is limited by how fast the code executes the main loop.

As seen in Figure 8, 1000Hz is near the limit since the note is being quantized to around 6 steps, any higher frequency would lead to aliasing/incorrect notes, so the stored song (Everytime We Touch by Cascada) is transposed down an octave to fit this frequency range. For code, refer to Appendix 0.2.

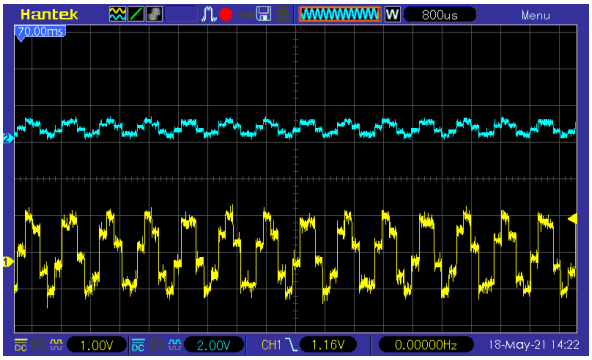


Figure 8: Amplified sine waves at 1000Hz

Game Body

Hardware

For the game body, the PSoC will act as a game engine that takes in input from the 8051 and controls the TFT display, music player (PSoC stick) and score display. It is wired up as shown in Figure 9 to the LED, TFT and stick, since the PSoC and TFT both operate on 3.3V logic. Note that in order for PSoC to directly talk to the stick (3.3V to 5V), it must share a ground with the stick, which is why the PSoC BB is grounded here. Refer to Appendix 0.6 for full schematics.

Software

Interrupts

The big board has two interrupts (Figure 10, Appendix 0.7): the timer interrupt is functionally the same as the timer interrupt in the stick, the only difference being that it is driven by a 60Hz clock, corresponding to the update rate of the TFT screen; the pin interrupt is on pin 5.0, which is connected to P3.4 on the 8051, meaning it is called when a state change occurs on the controller. On pin interrupt, the PSoC updates it's internal stored state of the controller and checks for notes hit, which will be explained in greater detail in the next section.

Notes and timing

The game first waits for a right switch press in *wait_start*(). When the switch is grounded, the PSoC will pulse the music player to indicate to start playing music after

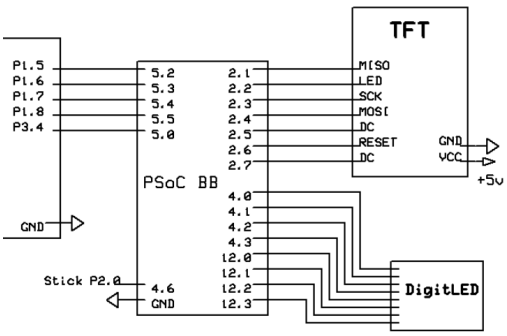


Figure 9: PSoCBB schematics

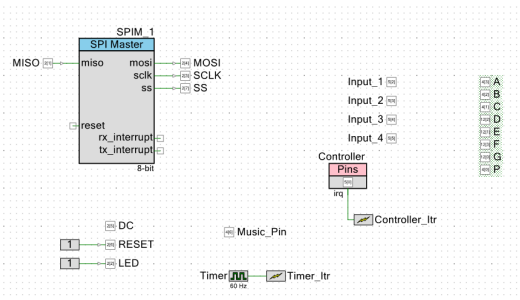


Figure 10: PSoCBB Creator

three seconds, at the same time the PSoC activates it's clock and starts incrementing it's own internal timer from -3 seconds. This allows the music to be synchronized with gameplay.

The falling notes in this game is hard coded into an array of size 2 arrays. The first element of a note array represents the time of when the note should hit the 'nowbar', or the horizontal line, on the screen at y=255; the second element of the array is which row the note should fall on, corresponding to the four different switches. This array is also sorted in chronological order, which would help optimize the runtime of the game loop. Each note also has a status that is stored at the same index of another array. A status of 0 means active, 1 means to be erased and 2 means inactive. All notes start as active.

Game Physics

The main game loop is iterates through all the notes in the hard coded array starting from *current_note_idx*. For each note, the y position of a note at *note_time* is calculated using $y = 255 + \frac{(CURRENT\_TIME - note\_time) * 255}{120}$. This allows the note to be at 255 when *note_time* equals current time, while controlling the velocity such that it take 2 seconds (120 game ticks) to travel from the top of the screen to the nowbar.

If the y position is above screen (negative), this means all notes afterwards would also be above the screen due to the note array being chronological, which allows the routine to save time by break out of the loop and not iterating through the whole array. If the y position of an active (status=0) note is below the screen, this means the note is missed and the player's score will be reduced by 1. The note's status will be set to 1 to indicate it should be erased. Regardless of the note's status, *current_note_idx* will be incremented here as the note is now 'off screen' and no longer in need to be updated. The note is then drawn if the status is active, or erased and deactivated if the status is to be erased.

During the main loop, the pin interrupt will be called when needed. Other than updating the controller state, it also checks all notes with *note_time* within 0.3 seconds of the current time, also known as the slop window as it allows some error, with the current controller state to check if notes are 'hit'. If a note is 'hit', the note status will be set to 1 (to be erased) and the score will be incremented by 1.

At the end of the game loop, the nowbar and lights corresponding to switch states are also drawn to the TFT and the game score is displayed on the digitLED. Note that the lights are updated here but not on interrupt. This is because the serial writing in the main loop might get interrupted and if the interrupt itself also writes to serial, this would produce interweaving of serial messages, therefore all serial writing is done in the main loop.

Rendering

Before drawing to the TFT, the frame of drawing must be specified. The *init_columns* and *init_page* commands help set the x and y range of the frame. Since writing to the whole screen is extremely slow (takes around 1 second), it is unrealistic to re render the screen on every game step. Instead whenever the PSoC draws a note, it simultaneously draws a black square of the same size right above, this erases the note in the last time step since the note only travels downwards, creating the illusion of the note falling downward (Figure 11).

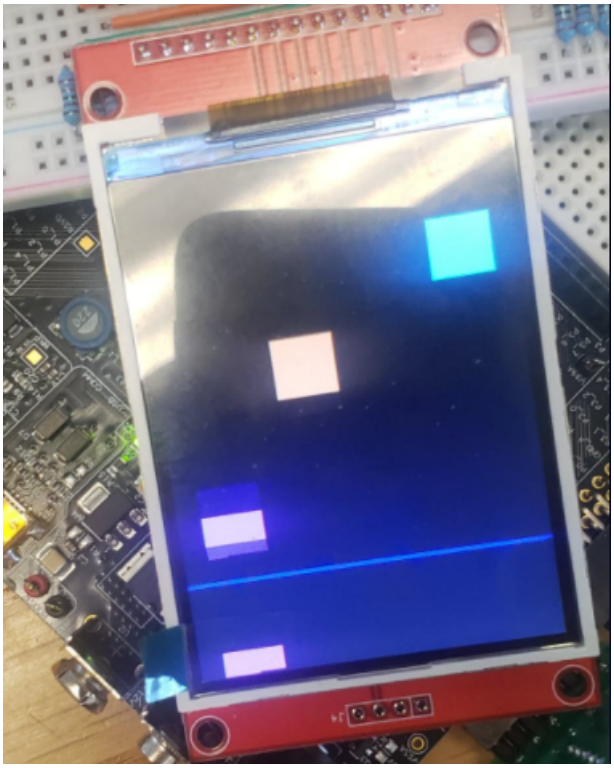


Figure 11: Game screen

Video demo

Here's a link to a video demo: https://youtu.be/lGGdTS5MYy0

Appendix

0.1 Controller ASM code

```
.org 00h
ljmp start

.org 100h
start:
mov P1, #0FFh     ; init as input pins
mov R0, #00h      ; last state memory
mov P3,#11101111b; P3.4 low

loop:
mov A, P1         ; read input
ANL A, #0Fh       ; mask off first 4 bit (bitwise AND)
mov R1, A         ; store to R1
XRL A, R0
CJNE A, #00h, NOTSAME     ; Jump if not same
SJMP LOOP

NOTSAME:
MOV A, R1
MOV R0, A             ; store old state
SWAP A
ORL A, #0Fh       ; mask off last 4 bits with 1s
MOV P1, A
SETB P3.4         ; pulse interrupt
CLR P3.4
LCALL DELAY       ; debouncing delay

SJMP LOOP

DELAY:
MOV R3, #80h
DEBOUNCE2:
    MOV R2, #0FFh
    DEBOUNCE:
        DJNZ R2, DEBOUNCE
    DJNZ R3, DEBOUNCE2
ret
```

0.2 PSoC stick code

```
#include <project.h>
#include <stdio.h>
#include <math.h>

long CURRENT_TIME = 0;
const int sample_rate = 88000;
float OMEGA = 2*M_PI/88000;

//Hard coded song  - with a note per half step
float freqs[][2] = {{830,1},{784,1},{784,1},{698,1},{698,1},{698,0}...};

int wave_gen(int current, float frequency){
    //Given t and f, generates integer value of sin(2(pi)(f)/Fs*t) scaled between 0 and 254
    int val = (int)((float)(127+ 127*sin(OMEGA*frequency*(float)current)));
    return val;
}

int nice_song(int current){
    return nicesong[current/11];
}

int hard_coded_song(int current){
    int halfbeat = sample_rate/4; // sample per half beat

    if (current <0 || current > 200*halfbeat){ // if current time is before 0, or after song end
        return 0;
    }

    int time_in_idx = current%(halfbeat); //time into current halfbeat
    int current_idx = current/halfbeat; // current halfbeat
```

```
    if (time_in_idx <1000 && freqs[current_idx][1]==1){ // if new note is played, dont play first
        5% of it to create pause
        return 0;
    }
    //play frequency of current note
    return wave_gen(current,freqs[current_idx][0]);
}

//Timer interrupt
CY_ISR( Timer_Itr_Handler){
       CURRENT_TIME+=1;
    }

//Pin 1 interrupt
CY_ISR( Pin_1_Itr_Handler){
       CURRENT_TIME=-3*88000;
        Timer_Start();       // start timer from -3 seconds
    }

int main()
{
    CyGlobalIntEnable; /* Enable global interrupts. */

    CyDelay(1);
    // start interrupts
    Timer_Itr_StartEx(Timer_Itr_Handler);
    Pin_1_Itr_StartEx(Pin_1_Itr_Handler);

     /* Start VDAC */
    VDAC8_1_Start();

    for(;;)
    {
        //Set DAC value to waveform value of song at current time
        VDAC8_1_SetValue(hard_coded_song(CURRENT_TIME));
    }
}
```

0.3 PSoC BB code

```
#include <project.h>
#include "tft.h"
#include <stdio.h>
#include <stdlib.h>

// x position of rows
int row_left[4] = {16,78,130,187};
// hard coded notes in chronological order
int notes[][2] = {{0,1},{30,1},{45,3},{60,2}...} ;
int num_notes = 79;
int notes_status[79] = {0};
// initial controller state
int controller_state[4] = {1,1,1,1};
int old_controller_state[4] = {1,1,1,1};

// color of nowbar
uint16 bar_color = 0xF8;
void DigitLED_Write( int);
void DigitLED_WriteHex( int);



void init_columns(uint16 x_start,uint16 x_len){
    // helper function to initialize column addresses from x_start to x_start + x_len for tft
    write8_a0(0x2A);                    // send Column Address Set command
    write8_a1(x_start >> 8);                // set SC[15:0]
    write8_a1(x_start & 0x00FF);
    uint16 x_end = x_start + x_len;
    write8_a1(x_end >> 8);                // set EC[15:0]
    write8_a1(x_end & 0x00FF);
}

void init_page(uint16 y_start,uint16 y_len){
    // helper function to initialize row addresses from y_start to y_start + y_len for tft
    write8_a0(0x2B);                    // send Page Address Set command
```

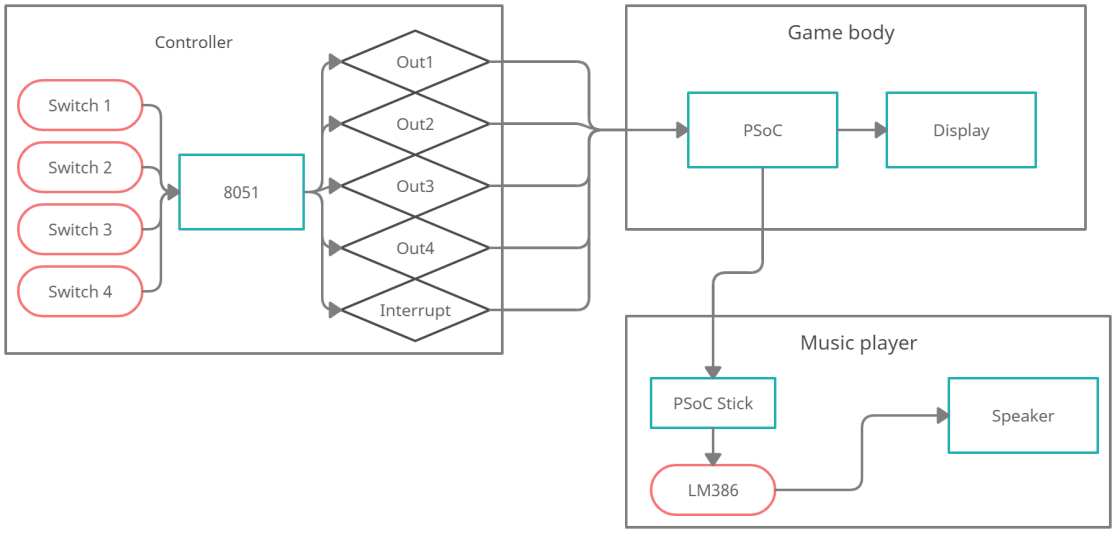
```
    uint16 y_end = y_start + y_len;
    write8_a1(y_start >> 8);                    // set SP[15:0]
    write8_a1(y_start & 0x00FF);
    write8_a1(y_end >> 8);                    // set EP[15:0]
    write8_a1(y_end & 0x00FF);
}

void draw_note(int row, uint16 y){
    // function to draw note at row and height y while erasing note above
    int x_start = row_left[row];
    int len = 39;            // length of square note
    //pick color according to row
    uint16 color = 0x0000;
    if (row ==0){
        color = 0xF0FF;
    }
    else if (row==1){
       color = 0xFF0F;
    }
    else if (row ==2 ){
        color = 0xFFF0;
    }
    else{
        color = 0x0FFF;
    }
    // initialize columns
    init_columns(x_start,len);

    //initialize rows such that tft doesnt draw pixels off screen
    int y_end = y +19;
    int y_start = y - len-19;
    if (y_end > 319){
        y_end = 319;
    }
    if (y_start < 0){
        y_start = 0;
    }
    init_page(y_start,y_end-y_start);


    int i;
    write8_a0(0x2C);                        // send Memory Write command
    for (i=0; i<(y-y_start-19)*len-1;i++)       //erasing old square
        {
           write8_a1(0x00);                              // 0x00 is the color black
           write8_a1(0x00);
        }
    for (i=0; i<(y_end-y+20)*len-1;i++)      // draw colored pixels to fill the square
        {
            write8_a1(color>>8);
                write8_a1(color & 0x00FF);
        }

    write8_a0(0x00);                        //done
}

void draw_black_note(int row, uint16 y){
    // function to erase note at row and height y and erasing note above
    int x_start = row_left[row];
    int len = 39;                  //length of square note
    uint16 color = 0x0000;
    //initialize x of draw range
    init_columns(x_start,len);

    //initialize y of draw range such that tft doesnt draw off screen
    int y_end = y +19;
    int y_start = y - len-19;
    if (y_end > 319){
        y_end = 319;
    }
    if (y_start < 0){
        y_start = 0;
    }
    init_page(y_start,y_end-y_start);

    int i;
    write8_a0(0x2C);                        // send Memory Write command

    //erase old square
    for (i=0; i<(y-y_start-19)*len-1;i++)
        {
```

```
            write8_a1(0x00);
            write8_a1(0x00);
        }
    //erase current square
    for (i=0; i<(y_end-y+20)*len-1;i++)
        {
            write8_a1(color>>8);
                write8_a1(color & 0x00FF);
        }
    write8_a0(0x00);
}


void draw_rectangle(uint16 x_start, uint16 x_len, uint16 y_start,uint16 y_len, uint16 color){
    //draws x_len by y_len colored rectangle with (x_start,y_start) as upper left corner

    //define boundaries
    init_columns(x_start,x_len);
    init_page(y_start,y_len);

    //fill boundaries with colored pixels
    int i;
    write8_a0(0x2C);                    // send Memory Write command
    for (i=0; i<x_len*y_len-1;i++)
        {
            write8_a1(color>>8);                                // 0xFF0F is the color orange
                write8_a1(color & 0x00FF);
        }
    write8_a0(0x00);
}

// function draws lights on active switches according to the current controller state
void draw_lights(){
    int i = 0;
    for (i=0; i<4; i++){
        // if no state change no change in display
        if (old_controller_state[i]!= controller_state[i]){
            if (controller_state[i]==0){
                // draw light rectangle on corresponding slot
                draw_rectangle(row_left[i],39,251,10,0x0FF0);
            }
            else{
                // erase light rectangle and redraw bar
                draw_rectangle(row_left[i],39,251,10,0x0000);
                draw_rectangle(row_left[i],39,255,3,bar_color);
            }
            // remember last drawn state
            old_controller_state[i] = controller_state[i];
        }
    }
}

int current_note_idx = 0;
int score = 0;
long CURRENT_TIME = -3*60;

//Timer interrupt
CY_ISR( Timer_Itr_Handler){
      CURRENT_TIME+=1;
    }

// Controller interrupt
CY_ISR( Controller_Itr_Handler){
    //get new state of controller
    controller_state[2] = Input_1_Read();
    controller_state[0] = Input_2_Read();
    controller_state[1] = Input_3_Read();
    controller_state[3] = Input_4_Read();

    // detect if any notes are hit
    int idx2 = 0;
    //iterate through all notes
    for (idx2=current_note_idx; idx2 < num_notes; idx2++){
        if (abs(CURRENT_TIME - notes[idx2][0]) < 17){ //hit on time
            if (controller_state[notes[idx2][1]]==0){
                notes_status[current_note_idx] = 1;  // note status 1 -> to be cleared
                if (score <9){
                    score+=1;           //increment score, cap score at 9
                }
            }
```

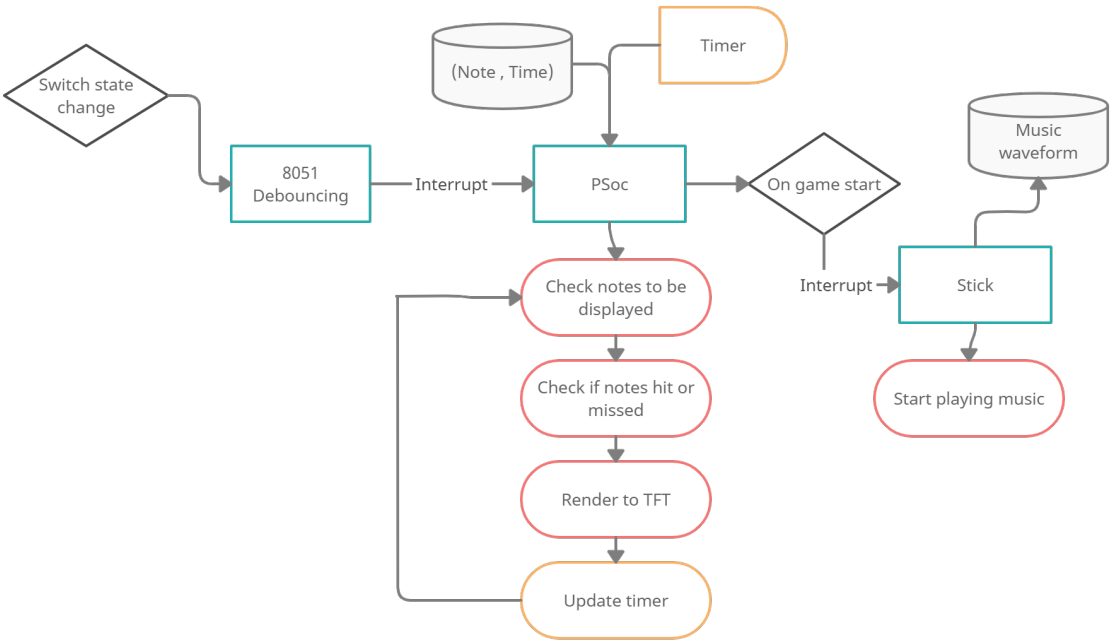
```
        }
    }
    Controller_ClearInterrupt();        // clear interrupt
    }

int time_to_y(int note_time){
    //function to convert note time to y position on TFT screen
    return 255+ ((CURRENT_TIME-note_time)*255)/120;
}


void wait_start(){
    //function that wait till a right switch press, then starts game timer and starts music player
    for(;;){
        if (controller_state[3]==0){
            Timer_Start();
            // pulse music player interrupt
            Music_Pin_Write(1);
            Music_Pin_Write(0);
            break;
        }
    }
}


void DigitLED_Write( int bi)
// write binary state to digitLED
{
    A_Write((bi >> 7) & 1);
    B_Write((bi >> 6) & 1);
    C_Write((bi >> 5) & 1);
    D_Write((bi >> 4) & 1);
    E_Write((bi >> 3) & 1);
    F_Write((bi >> 2) & 1);
    G_Write((bi >> 1) & 1);
    P_Write((bi >> 0) & 1);
}

// DigitLED_WriteHex will write the hexadicimal number "hex" to the LED screen
void DigitLED_WriteHex( int hex)
{
    if (hex == 0x00 || hex <= 0x00)
        DigitLED_Write(0b00000011);
    else if (hex == 0x01)
        DigitLED_Write(0b10011111);
    else if (hex == 0x02)
        DigitLED_Write(0b00100101);
    else if (hex == 0x03)
        DigitLED_Write(0b00001101);
    else if (hex == 0x04)
        DigitLED_Write(0b10011001);
    else if (hex == 0x05)
        DigitLED_Write(0b01001001);
    else if (hex == 0x06)
        DigitLED_Write(0b01000001);
    else if (hex == 0x07)
        DigitLED_Write(0b00011111);
    else if (hex == 0x08)
        DigitLED_Write(0b00000001);
    else if (hex == 0x09 || hex >= 0x09)
        DigitLED_Write(0b00011001);
    else
        DigitLED_Write(0b11111111);
}
int main()
{
    CyGlobalIntEnable;                  // Enable global interrupts
    SPIM_1_Start();                     // initialize SPIM component
    Timer_Itr_StartEx(Timer_Itr_Handler);   // timer interrupt enable

    tftStart();                         // initialize the TFT display
    Controller_Itr_StartEx(Controller_Itr_Handler);   // controller interrupt enable

    // screen is 239 by 319

    draw_rectangle(0,239,0,322,0);          //black screen
    draw_rectangle(0,239,255,3,bar_color);  // draw nowbar

    int idx=0;
    wait_start();       // wait until right switch press
    for(;;) {
```

```
        //iterate through all notes from current note onward
        for (idx=current_note_idx; idx < num_notes; idx++){
            //calculate y position of note
            int note_time = notes[idx][0];
            int y_pos = time_to_y(note_time);

            if (y_pos <0){  // note is above screen, all note after will also be above screen
                break;
            }
            if (y_pos > 350){  // note is below screen
                if (notes_status[idx]==0){
                    notes_status[idx] = 1;// miss
                    if (score >0){
                        score-=1;       // deduct score, flooring at 0
                    }
                }
                current_note_idx++;     // note passed, increment current note idx
                continue;
            }

            if (notes_status[idx]==0){
                // draw note if it has status 0
                draw_note(notes[idx][1], y_pos);
            }
            else if (notes_status[idx]==1){
                // clear note if it has status 1, set status to 2 (disregarded)
                draw_black_note(notes[idx][1], y_pos);
                notes_status[idx]+=1;
            }

        }

        // redraw bar
        draw_rectangle(0,239,255,3,bar_color);
        // draw lights
        draw_lights();
        // print score to DigitLED
        DigitLED_WriteHex(score);
    }
}
```

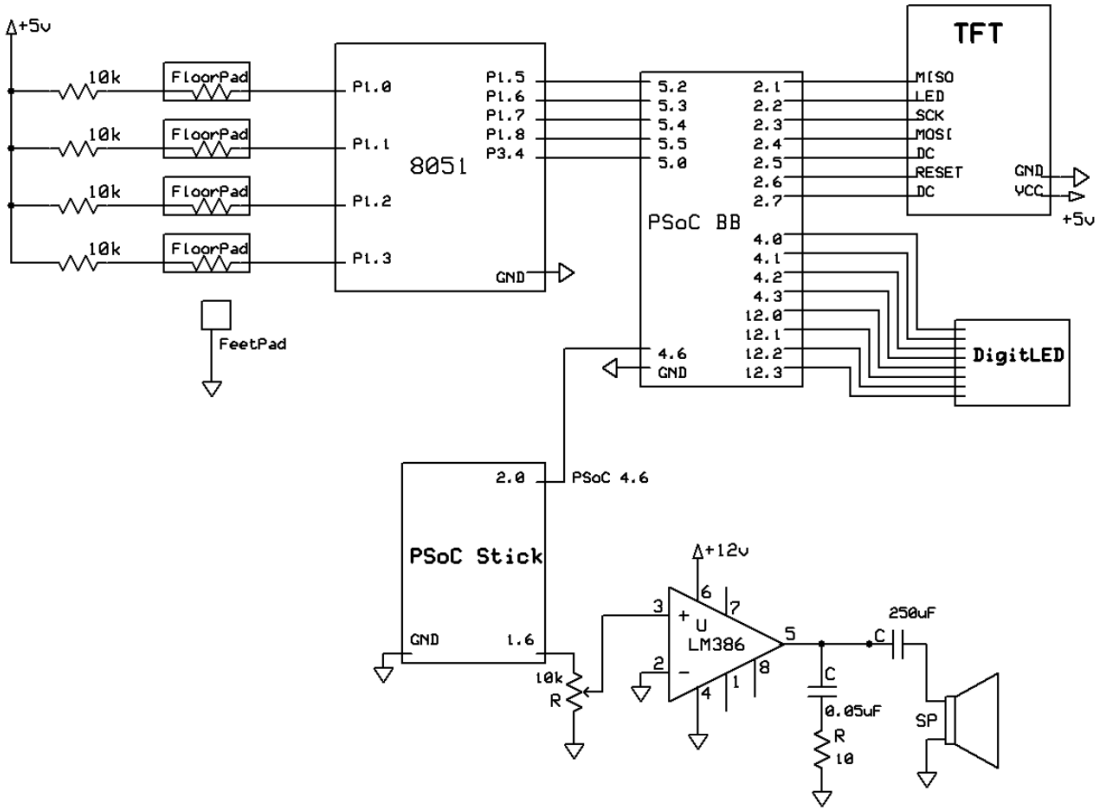
0.4 Hardware Flowchart



0.5 Software Flowchart



0.6 Schematics



0.7 PSoC BB creator