6.115 Final Project: Snake Game

Edward Jin

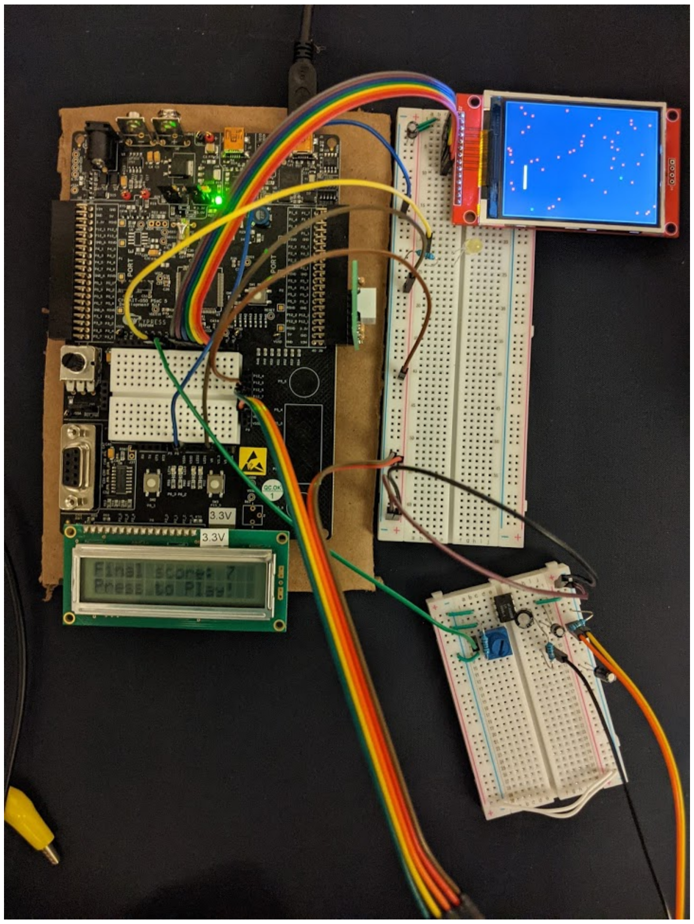


Figure 1: The game in action!

MIT Department of Electrical Engineering and Computer Science
6.115 Microcomputer Project Laboratory
May 20, 2021

1 Background and Introduction

For my final project, I decided to implement a variation of the extremely popular game Snake. The input for the system is simple - the only way a player can send signals to the game is through the joystick. However, this is enough to play the game in its entirety. This design was modelled on the Atari 2600, which was a widely used game console about 40 years ago which similarly relied on a single joystick for input, yet it was able to spawn some of the most iconic games of that generation.

In the project, I took the base concept of the game of snake and expanded on it significantly more over the original basic version. Specifically, I added more outputs, including sound, light, music, and text, as well as new features within the game itself, including traps and shields. Overall, these features make the game significantly more enjoyable than the base version, and add a new level of complexity to the simple game.

2 Hardware Description

The PSoC Big Board is the main driver of all the logic and functionality of the game, and all other components are connected to it. The main and sole source of input is the joystick, which sends analog signals of the x and y locations of the joystick, as well as button pressing signals (which can be pulled up directly in the PSoC software). On the output side, we have four sources of output as follows:

- A standard LCD screen for displaying text output, game status, and game score.
- A TFT screen for displaying the actual game output.
- An LED for displaying an indication of when the player collects the food.
- A speaker for playing game music.

2.1 Hardware Schematic and Documentation

As the PSoC Big Board runs on 3.3 V, all components listed above ran on the provided PSoC voltage and hence also 3.3 V. The LCD was connected directly to the PSoC on the LED slot (ports P2[6:0]), and the TFT screen and LED were also connected to the PSoC with jumper cables. The speaker was connected to the PSoC through the LM386 for a gain of around 20, as according to the datasheet of the LM386. Additionally, a variable potentiometer added to the output allows for volume control by the user. Finally, a difficulty switch can either be chosen to be high or low, causing respective changes in the difficulty of the game.

The following are connections made directly to components from the Big Board:

Type	PSOC Pin	Connection
Analog Input	6_0	Joystick VRx
Analog Input	6_6	Joystick VRy
Digital Input	12_7	Joystick SW
Digital Input	12_6	Difficulty Setting
SPI Output	0_1 - 0_7	TFT
Analog Output	3_0	LM386 Pin 3 (to speaker)
Digital Output	3_1	LED

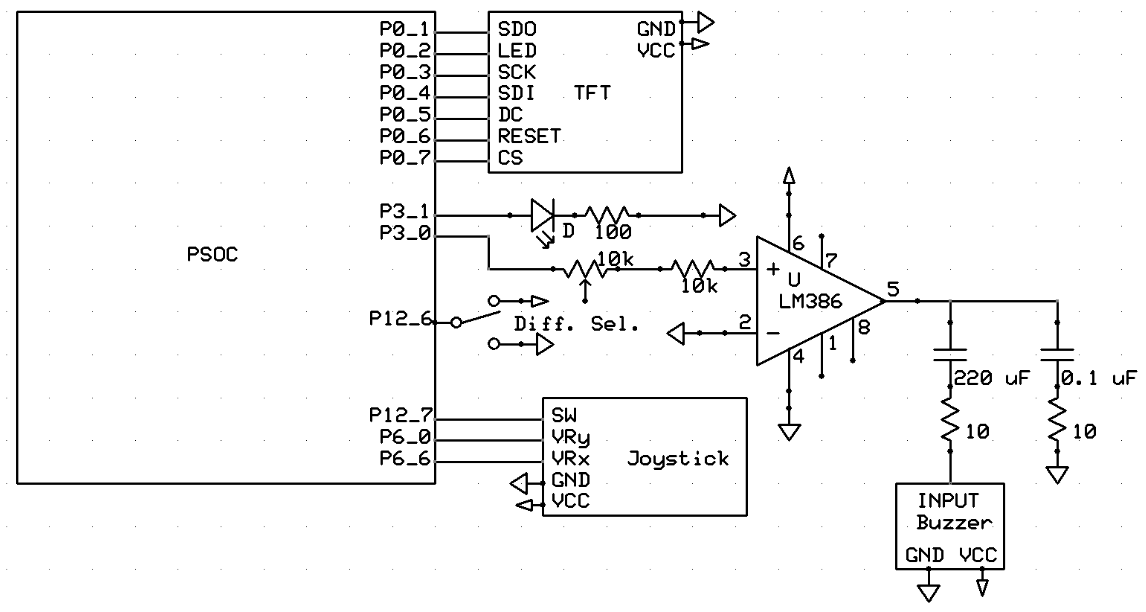


Figure 2: Hardware Schematic

3 Software Description

The software is the bulk of this project, and it involved a significant amount of coding in the PSoC C language. The functionality documentation is in section 4, and details how each of the functions in the program come together to make a playable game. The functions listed here are present within the code and a brief description of their function is listed below:

Function	Description
void InitializeVariables()	Initializes variables
void InitializeModules()	Initializes all PSoC modules
void InitializeTFT()	Initializes the TFT screen components
void InitializeTFTSnake()	Draws all the actual movable components on the screen
void snakePositionsPush(coordinate c)	Adds coordinate to the snakePositions deque (see below)
coordinate snakePositionsPop()	Removes coordinate from the snakePositions deque (see below)
int getJoystickState()	Checks the state of the joystick
int getJoystickX()	Gets the joystick X position as an int between 0 and 255
int getJoystickY()	Gets the joystick X position as an int between 0 and 255
void displayLCDNumeric(int x, int line)	Displays text on LCD
void displayLCDString(char* x, int line)	Displays text on LCD
void drawPixel(int x, int y)	Helper function, draws pixel on screen.
void moveSnake()	Moves the snake in a specified direction.
int PRNG(int max)	Implements random number generation (see below)
coordinate generateValidCoordinate()	Generates a coordinate which is not yet filled
void generateNewFoodPosition()	Generates new positions for food
void generateNewBarrierPositions()	Generates new positions for barriers
void generateNewShieldPosition()	Generates new position for shield
void updateDirection()	Updates the direction the snake is pointing

3.1 PSoC Schematic and Pins

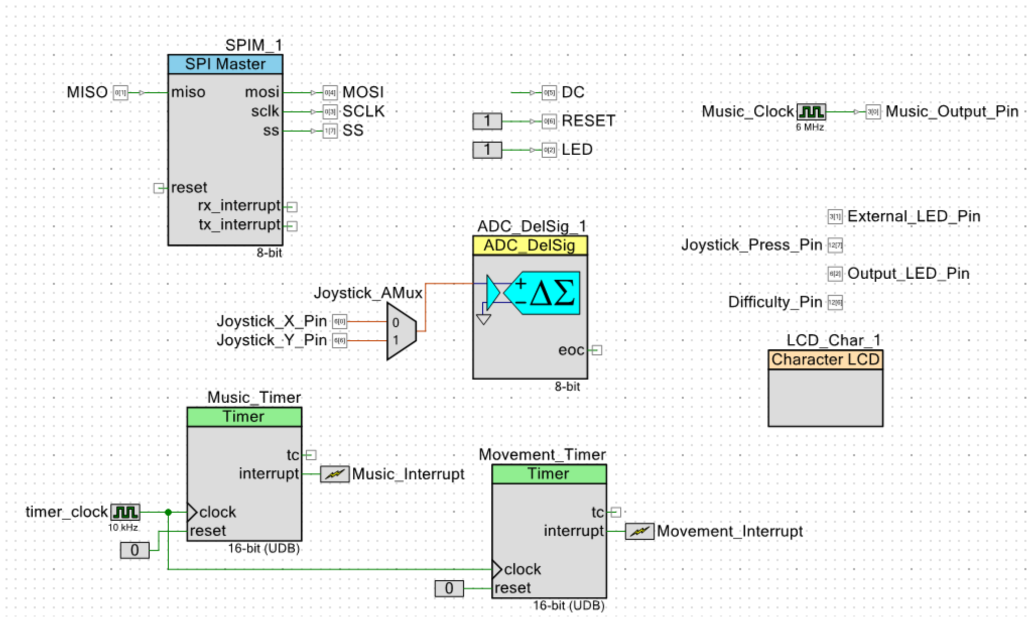
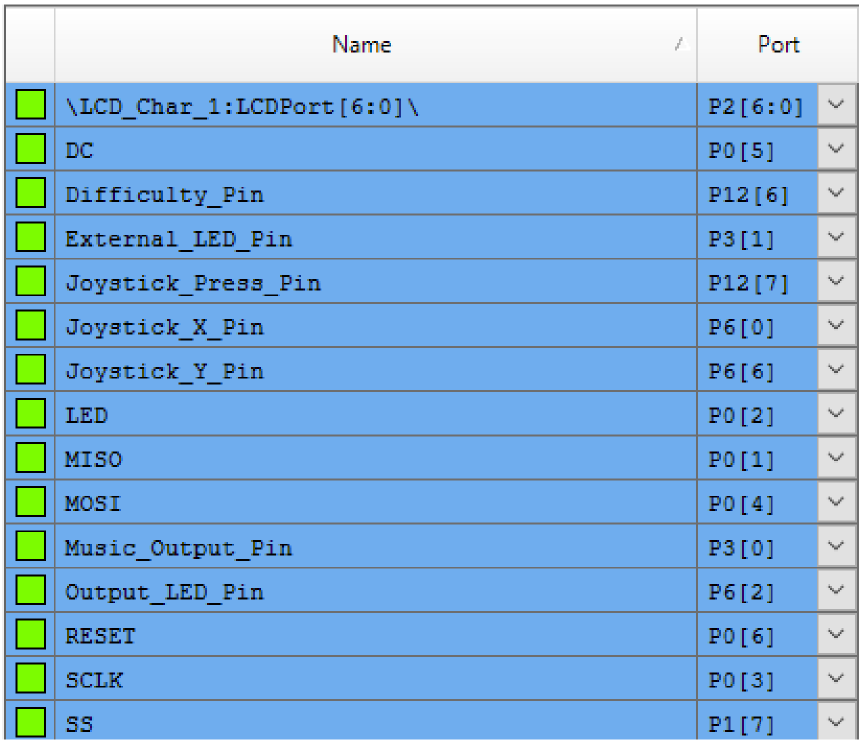


Figure 3: PSoC Schematic



	Name	Port
🟩	\LCD_Char_1:LCDPort[6:0]\	P2[6:0]
🟩	DC	P0[5]
🟩	Difficulty_Pin	P12[6]
🟩	External_LED_Pin	P3[1]
🟩	Joystick_Press_Pin	P12[7]
🟩	Joystick_X_Pin	P6[0]
🟩	Joystick_Y_Pin	P6[6]
🟩	LED	P0[2]
🟩	MISO	P0[1]
🟩	MOSI	P0[4]
🟩	Music_Output_Pin	P3[0]
🟩	Output_LED_Pin	P6[2]
🟩	RESET	P0[6]
🟩	SCLK	P0[3]
🟩	SS	P1[7]

Figure 4: PSoC Pins

3.2 Program Flow

The program itself generally flows as a simple state machine, with the output being updated by interrupts. It starts off by initializing everything, then waits for the joystick input to transition to the next state of playing, where interrupts are enabled. Once the snake dies, the interrupts are once again disabled and the program is allowed to go back to the initial state after the joystick is pressed.

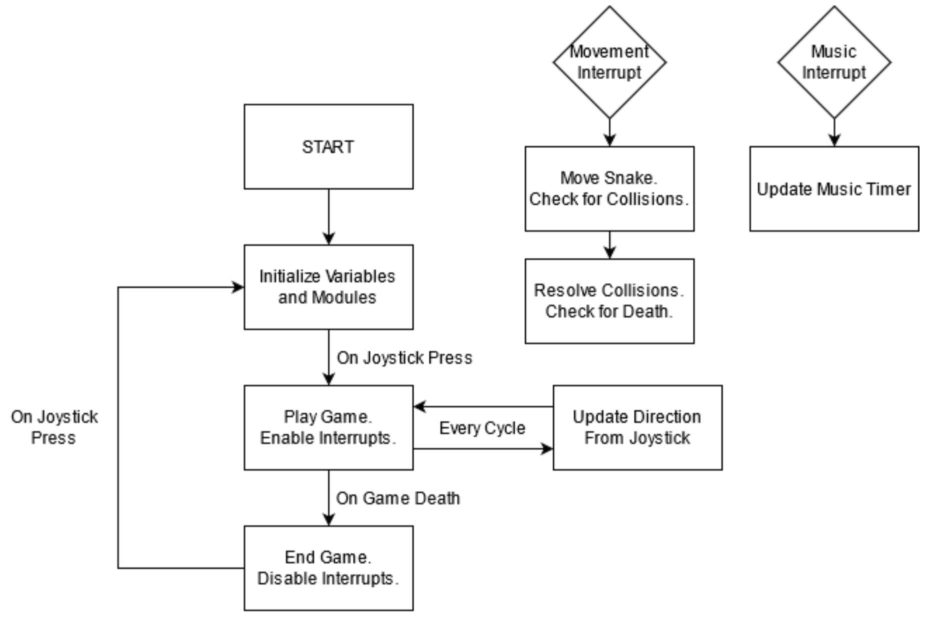

Movement Interrupt
Music Interrupt
START
Move Snake. Check for Collisions.
Update Music Timer
Initialize Variables and Modules
Resolve Collisions. Check for Death.
On Joystick Press
On Joystick Press
Play Game. Enable Interrupts.
Every Cycle
Update Direction From Joystick
On Game Death
End Game. Disable Interrupts.

Figure 5: Program Flow

4 Design, Documentation, and Implementation

4.1 Hardware Documentation and Testing

The process of testing all the hardware components of the system was incremental. Luckily, code for the LCD had already been written before in the form of the PSoC API, and the PSoC additionally already had a built-in slot for an onboard LCD. For the TFT screen, connections were made to a contiguous block of ports on the PSoC, following the TFT manual on the course website. The resistors mentioned in the TFT manual were not included, as the big board already uses 3.3 V signals and does not need to downconvert again.

For the LED, a resistor value was chosen experimentally such that the light would shine well, but not burn out. This turned out to be about 100 ohms with the input current from the PSoC. For the joystick, the outputs were first observed on the oscilloscope. It was found that the VRx and VRy signals were essentially analog signals ranging from 0 to the input voltage, with the rest position at half of the input voltage. The SW signal was unable to be seen on the scope, but testing with a pull-up resistor (implemented in the PSoC software) was enough to make it readable.

Finally, for the buzzer, there are quite a few components that needed to be implemented. First, it was found that the sound coming from the buzzer was quite low, so the input needed to be amped up, which was done with the LM386. Using the schematic provided in the datasheet, a gain of around 20 was able to be produced. Afterwards, it was thought that turning off or controlling the music would be useful, which is why a 10k potentiometer was added. The potentiometer is able to directly control the amount of sound that comes out of the buzzer, ranging from no sound at all to a relatively high amount.

4.2 Functionality Documentation

At the beginning, the screen will initialize and a message on the LCD will tell the player to press the joystick to start playing. Once this happens, the game then is started on the TFT screen, and the LCD keeps track of score and if the player is shielded or not (see below). The snake automatically starts moving, and the player has to control it with the joystick to not crash into walls or itself. The object of the game is to collect as many food particles as possible, which are denoted with a blue color. The music also starts playing here, and collisions with any type of particle are expressed to the player through a flashing of the LED. As mentioned above, the music volume can be controlled with a potentiometer located on the breadboard.

In my version of snake, I additionally added traps, which are denoted with a red color. The positions of the traps are randomly generated and the snake cannot run into them without dying. On the other hand, there is also a shield particle, which is green. When the snake collects a shield, it is able to run through a trap without dying, and the game continues. The possibility of using shields for trap removal also allows for getting around previously impossible situations, for example, when the food is surrounded by four traps. Once the snake is unshielded, the shield power up is once again regenerated on a random position in the map.

When the player dies, the music stops and the final score is displayed on the LCD screen.

If the input to pin 12_6 is changed, the difficulty can be changed. In the hard mode, there are 50 traps rather than 10 in the easy mode. Additionally, the snake moves twice as fast (once every 50 ms rather than 100 ms). This represents a significant ramp up in difficulty and is intended to be used for players who have already mastered the easy version.

4.3 Design: Random Number Generation

In the game of snake, randomness is decidedly necessary, in order to generate the positions of objects on the game board itself to create non-repetitive games maps. To this end, I implemented a pseudo-random number generator based on one presented by Park and Miller in 1988:

```
const long randomConst = 16807;
const long modulus = 2147483647;
long randomSeed = 350; // chosen randomly
int PRNG(int max)
{
    //uses fancy math to make a PRNG. returns results in the range [0, max)
    randomSeed = (randomConst * randomSeed);
    randomSeed %= modulus;
    int result = randomSeed % max;
    if(result < 0)
        result += max;
    return result;
}
```

While the initial seed is fixed, the gameplay of the player changes it from game to game, and so it is unlikely that any game after the first will be identical to previous games. Aside from this, the random number generator works well.

The random constant 16807 is not chosen randomly. Similarly, the modulus of $m = 2^{31} - 1$ is not chosen randomly either. The modulus is chosen as such since it is the largest possible 'long' value, but also because it is a prime number. The number 16807 is then chosen to complement this modulus, as it is a primitive root mod m. In other words, we have that

$$\mathrm{ord}_m(16807) = \min_{n>0}\{n|16807^n \equiv 1 \pmod m\} = m - 1.$$

So the random number generator here has a period of m, meaning that it is highly unlikely to repeat itself (even disregarding the fact that player inputs cause varying amounts of RNG change).

The function itself has an additional component which simply takes the result from the PRNG and takes it modulo max, hence generating random numbers in the range $[0, \max - 1]$. Note that the argument max is a 16-bit integer (rather than a 32-bit long). This is intentional, as having moduli close to the actual modulus m would result in a more skewed probability distribution (as the generated numbers mod m would not be close to uniform anymore). Further, we do not use such moduli in the code anyways.

4.4 Design: Snake Position Tracking

The snake's position itself is tracked with two components: a double-ended queue and a coordinate corresponding to where the snake head is. The snake head position is important as it controls all interactions and intersections with the game board, and is the one that it moved on each interrupt. On the other hand, the body of the snake is also important for checking self-intersections (which lead to a game loss), and also needs to be rendered every interrupt. Thus, both need to be tracked.

The snake head position is easy to track, as it can be modified from interrupt to interrupt simply by changing its position depending on the current direction. On the other hand, the position of the tail components is harder to track, since its length is variable, and additionally the coordinates of the positions are also changing on every interrupt.

Keeping track of these positions with python lists is easy - one would simply use an append command to add the new position of the snake head to the end of the snake, and remove the first position with a pop command. Unfortunately, C does not provide us with such abstract operations, and we have to work directly with arrays.

To this end, I implemented a double-ended queue that would provide for these two functions. It is implemented based on a circular array snakePositions, where the start and end positions are tracked. When an element is pushed into the end of the array, the end position is incremented and the new element is placed right after the previous end. When an element is popped from the front of the array, the start position is incremented, implying that the element originally there is no longer accessible.

All positions tracking is done modulo the array size of 1000, thus ensuring that we never run out of memory or run into bound errors, until the snake gets to length 1000. In testing, no person achieved a length of more than 25 on easy mode and more than 10 on hard mode, so suffice to say this length 1000 is essentially unreachable. Additionally, in theory, boundary errors could happen if the start position is greater than the ending position. However, in the current code impelmentation, this never happens, as there is always a corresponding push operation for every pop operation done.

```
coordinate snakeHeadPosition;
const int snakeMaxSize = 1000;
coordinate snakePositions[1000]; // size should be same as snakeMaxSize
int snakePositionsStart;
int snakePositionsEnd;
```

```
void snakePositionsPush(coordinate c)
{
    //c implementation of a deque. pushes new element at the end.
    snakePositionsEnd++;
    snakePositionsEnd %= snakeMaxSize;
    snakePositions[snakePositionsEnd] = c;
}

coordinate snakePositionsPop()
{
    //c implementation of a deque. pops element at the start and returns it.
    snakePositionsStart++;
    snakePositionsStart %= snakeMaxSize;
    return snakePositions[snakePositionsStart-1];
}
```

4.5 Design: Joystick Reading and Direction Funding

One major problem with finding the direction was that the joystick could be read every loop cycle, even though movement is only updated on every interrupt. The natural question then becomes how we can integrate the joystick information, if the input came within a loop cycle rather than on a movement cycle.

To solve this, it was decided to read the joystick input on every loop cycle. However, this didn't work out too well, since it would simply use the last joystick input right before movement to set the direction and inputs in succession could cancel out. Additionally, if the snake was moving north, and then two inputs of east, then south were inputted, the code would assume that both were valid direction inputs since the snake would change direction on both, and so on the actual movement update the snake would move directly south and die by running into itself.

To get around this problem, the previous direction that the snake actually moved in was recorded. Then, every new joystick input was compared to that previous direction, rather than the direction the last joystick input moved in. This allowed for much more robust movement and much fewer self-deaths. The joystick tolerances of 50 and 200 were found experimentally to work well with the joystick hardware.

```
    // previousDirection set by interrupt movement function
    if(previousDirection == DIR_NORTH || previousDirection == DIR_SOUTH)
    {
        //if snake currently going north or south, only change to either east or west
        if(x < 50)
            newDirection = DIR_WEST;
        else if(x > 200)
            newDirection = DIR_EAST;
        else
            newDirection = previousDirection; // if x movement not indicated, ignore (don't
                change direction)
    }
    else if (previousDirection == DIR_EAST || previousDirection == DIR_WEST)
```

```
{
    if(y < 50)
        newDirection = DIR_SOUTH;
    else if(y > 200)
        newDirection = DIR_NORTH;
    else
        newDirection = previousDirection;
}
currentDirection = newDirection;
```

4.6 Design: Timing and Interrupts

For both music and movement, interrupts were the most convenient tool to use, as documented here: `https://www.cypress.com/file/44256/download`. The generation of the interrupts themselves is achieved through the timer module of the PSoC, which is connected to a 10 kHz clock with variable period, which allows for fine tuning of how often interrupts occur. The actual code called on interrupt is modified from the generated interrupt files.

On an actual interrupt, no functions are called other than clearing the interrupt flag. Rather, global variables (as defined with the extern command) are changed, and these changes are then acted on in the main code. This design is so interrupts aren't acted on during the middle of a function call, which can cause undefined behavior. By only setting flags, the code is ensured to run in the proper order.

The current parameters for the movement and music were found experimentally, and the period for the movement timer in hard mode is half of that of easy mode.

4.7 Design: Music / Speaker

As mentioned above, the music itself is controlled by an interrupt. This interrupt simply advances a global variable music_timer by 1, which controls the tempo of the music. The most straightforward way I found to make music with the buzzer was to convert each musical note to a frequency, and then pass a square wave of said frequency into the buzzer, which would be updated as music_timer was also incremented. In order to actually generate this frequency, I chose to use a clock of 6 MHz frequency and then set the clock divider in software. I specifically chose a high frequency here in order to have better resolution, as the divisions with a larger input frequency are more fine. Finally, the original sound coming out of the buzzer was a bit high bitched, so I opted to halve the frequencies to lower the output by an octave.

5 Overall Remarks

Overall, this implementation of snake is a more intricate and exciting variant as compared to the original game, which seems to run robustly on the PSoC hardware (as well as other hardware components!). I was also able to get a few of my friends to test out the game, and they mentioned that it was quite cool and exciting, making this project a relative success.

6 Code Appendix

6.1 Main File (main.c)

```c
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/
#include <project.h>
#include "GUI.h"
#include "tft.h"
#include "math.h"
#include "stdlib.h"
#include "global_variables.h"

//Directions
const int DIR_EAST = 0;
const int DIR_NORTH = 1;
const int DIR_WEST = 2;
const int DIR_SOUTH = 3;
int previousDirection;
int currentDirection;

//TFT Screen
const int gridSize = 4;
const int borderSize = 2;
const int SCREEN_X = 240;
const int SCREEN_Y = 320;

//Music and LED Control
int musicTimer;
const int NUM_CYCLES_LED = 4;
int LEDTimer;

//Game State Control
const int STATE_NEW = 0;
const int STATE_DEAD = 1;
const int STATE_PLAYING = 2;
int state;
const int DIFFICULTY_EASY = 0;
const int DIFFICULTY_HARD = 1;
int difficulty;
bool timeToMove = false;
bool snakeShielded = false;
int currentScore;

//coordinates and positions
typedef struct coordinate {
    int x;
    int y;
} coordinate;
coordinate snakeHeadPosition;
const int snakeMaxSize = 1000;
coordinate snakePositions[1000]; // size should be same as snakeMaxSize
int snakePositionsStart;
int snakePositionsEnd;
coordinate barrierPositions[50];
int numBarriers; // changes depending on difficulty
coordinate foodPosition;
coordinate shieldPosition;

//PRNG
    https://stackoverflow.com/questions/9492581/c-random-number-generation-pure-c-code-no-libraries-or-functio
const long randomConst = 16807;
const long modulus = 2147483647;
long randomSeed = 350;

//Function Declarations
void InitializeVariables();
void InitializeModules();
void InitializeTFT();
void InitializeTFTSnake();
void snakePositionsPush(coordinate c);
coordinate snakePositionsPop();
int getJoystickState();
int getJoystickX();
int getJoystickY();
void displayLCDNumeric(int x, int line);
void displayLCDString(char* x, int line);
void drawPixel(int x, int y);
void moveSnake();
int PRNG(int max);
coordinate generateValidCoordinate();
bool isValidCoordinate(coordinate c);
bool intersectSnake(coordinate c);
bool intersectBarrier(coordinate c);
inline bool intersectFood(coordinate c);
inline bool intersectShield(coordinate c);
void generateNewFoodPosition();
void generateNewBarrierPositions();
void generateNewShieldPosition();
void updateDirection();
void InitializeVariables()
{
   //initializes all variables to default state. sets difficulty and game speed
   //called on bootup and reset
   snakePositionsStart = 0;
   snakePositionsEnd = -1;
   currentDirection = DIR_NORTH;
   previousDirection = DIR_NORTH;
   state = STATE_NEW;
   currentScore = 0;
   snakeShielded = false;
   int i;
   coordinate c = {0,0};
   for(i = 0; i < 50; i++)
      barrierPositions[i] = c;
   if(Difficulty_Pin_Read())
      difficulty = DIFFICULTY_HARD;
   else
      difficulty = DIFFICULTY_EASY;

   if(difficulty == DIFFICULTY_HARD)
   {
      Movement_Timer_WritePeriod(500); //set game speed
      numBarriers = 50;
   }
   else
   {
      Movement_Timer_WritePeriod(1000);
      numBarriers = 10;
   }
}

void InitializeTFT()
{
   //initializes all components on the TFT screeen, including the border, the actual
       snake, barrier, food, and shield positions
   GUI_Clear();
   InitializeTFTSnake();
   // draw borders
   GUI_SetPenSize(gridSize);
   GUI_SetColor(GUI_GRAY_3F);
   GUI_DrawRect(0, 0, 238, 318);
   GUI_DrawRect(1, 1, 238, 318);
   // generate and draw barrier and food position
   generateNewBarrierPositions();
   generateNewFoodPosition();
   generateNewShieldPosition();
}

void InitializeModules()
```

```
{
    //initializes all the PSoC modules as well as the other initializations. called on
        startup only
    Music_Clock_Start();
    Joystick_AMux_Init();
    LCD_Char_1_Start();
    LCD_Char_1_ClearDisplay();
    ADC_DelSig_1_Start();
    SPIM_1_Start();
    Music_Interrupt_Start();
    Music_Timer_Start();
    Movement_Interrupt_Start();
    Movement_Timer_Start();
    InitializeVariables();
    GUI_Init();
    displayLCDString("Press Joystick", 0);
    displayLCDString("to start Game!", 1);
    CyDelay(50);
    InitializeTFT();
}

void InitializeTFTSnake()
{
    //initializes the TFT snake onto the top left corner
    coordinate c = {2,2};
    int i;
    GUI_SetColor(GUI_WHITE);
    for(i = 0; i < 5; i++)
    {
        snakePositionsPush(c);
        drawPixel(c.x, c.y);
        c.x += 4;
    }
    snakeHeadPosition = snakePositions[snakePositionsEnd];
    CyDelay(50);
}

void snakePositionsPush(coordinate c)
{
    //c implementation of a deque. pushes new element at the end.
    snakePositionsEnd++;
    snakePositionsEnd %= snakeMaxSize;
    snakePositions[snakePositionsEnd] = c;
}

coordinate snakePositionsPop()
{
    //c implementation of a deque. pops element at the start and returns it.
    snakePositionsStart++;
    snakePositionsStart %= snakeMaxSize;
    return snakePositions[snakePositionsStart-1];
}

inline int min(int x, int y)
{
    return x > y ? y : x;
}

inline int max(int x, int y)
{
    return x > y ? x : y;
}

int getJoystickState()
{
    //returns 1 if Joystick is pressed and 0 otherwise
    return 1 - Joystick_Press_Pin_Read();
}

int getJoystickX()
{
    //returns an int between 0 and 255 corresponding to the x position of the joystick.
    Joystick_AMux_Select(0);
    CyDelay(1); // 1 ms delay for multiplexer to switch
   ADC_DelSig_1_StartConvert();

      ADC_DelSig_1_IsEndConversion(ADC_DelSig_1_WAIT_FOR_RESULT); // wait for ADC to
           finish converting
   return ADC_DelSig_1_GetResult16();  // read and return the adc value
}

int getJoystickY()
{
    //returns an int between 0 and 255 corresponding to the y position of the joystick.
    Joystick_AMux_Select(1);
    CyDelay(1); // 1 ms delay for multiplexer to switch
   ADC_DelSig_1_StartConvert();
      ADC_DelSig_1_IsEndConversion(ADC_DelSig_1_WAIT_FOR_RESULT); // wait for ADC to
           finish converting
   return ADC_DelSig_1_GetResult16();  // read and return the adc value
}

void displayLCDNumeric(int x, int line)
{
    LCD_Char_1_Position(line, 0);
    LCD_Char_1_PrintNumber(x);
}

void displayLCDString(char* x, int line)
{
    LCD_Char_1_Position(line, 0);
    LCD_Char_1_PrintString(x);
}

inline void drawPixel(int x, int y)
{
    //draws a pixel of size gridSize at the specified coordinates
    GUI_FillRect(x, y, x + gridSize - 1, y + gridSize - 1);
    CyDelay(3);
}

inline void flashLED()
{
    LEDTimer = NUM_CYCLES_LED;
    External_LED_Pin_Write(1);
}
void moveSnake()
{
    //computes the next position of the snake, given the current direction.
    //checks if the position is invalid. if so, changes snake state to dead.
    //if there is an intersection of the snake with a barrier, checks if the snake is
        shielded. if so, remove shield and remove the barrier impacted.
    //if there is an intersection with food or the shield, update game variables
        accordingly
    int bottomBoundary = SCREEN_Y - gridSize - borderSize;
    int rightBoundary = SCREEN_X - gridSize - borderSize;

    //update position of snake head
    previousDirection = currentDirection;
    if(currentDirection == DIR_NORTH)
        snakeHeadPosition.y += gridSize;
    else if(currentDirection == DIR_SOUTH)
        snakeHeadPosition.y -= gridSize;
    else if(currentDirection == DIR_EAST)
        snakeHeadPosition.x -= gridSize;
    else if(currentDirection == DIR_WEST)
        snakeHeadPosition.x += gridSize;

    //check for intersections
    if(snakeHeadPosition.y > bottomBoundary || snakeHeadPosition.y < 0 ||
        snakeHeadPosition.x > rightBoundary || snakeHeadPosition.x < 0 ||
        intersectSnake(snakeHeadPosition))
        state = STATE_DEAD;

    if(intersectBarrier(snakeHeadPosition))
    {
        if(snakeShielded)
        {
            snakeShielded = false;
            generateNewShieldPosition();
            flashLED();
            //clear out removed barrier
            int i;
            coordinate c = {0,0};
            for(i = 0; i < numBarriers; i++)
                if(snakeHeadPosition.x == barrierPositions[i].x && snakeHeadPosition.y ==
                     barrierPositions[i].y)
                    barrierPositions[i] = c;
        }
        else
            state = STATE_DEAD;
    }
    // push new position to GUI
    snakeHeadPosition.y = min(max(snakeHeadPosition.y, borderSize), bottomBoundary);
    snakeHeadPosition.x = min(max(snakeHeadPosition.x, borderSize), rightBoundary);
    GUI_SetColor(GUI_WHITE);
    snakePositionsPush(snakeHeadPosition);
    drawPixel(snakeHeadPosition.x, snakeHeadPosition.y);

    //check for more intersections
    if(intersectFood(snakeHeadPosition))
    {
        currentScore++;
        flashLED();
        generateNewFoodPosition();
    }
    else
    {
        coordinate c = snakePositionsPop();
        //limit time LED is on
        LEDTimer--;
        if(LEDTimer <= 0)
            External_LED_Pin_Write(0);

        //remove previous snake location if not intersected with food
        GUI_SetColor(GUI_BLACK);
        drawPixel(c.x, c.y);

        //process interaction with shield
        if(intersectShield(snakeHeadPosition))
        {
            flashLED();
            snakeShielded = true;
            coordinate c = {0,0};
            shieldPosition = c; // disable shield; can't be shielded more than once!
        }
    }
}

int PRNG(int max)
```

```
{
    //uses fancy math to make a PRNG. returns results in the range [0, max)
    randomSeed = (randomConst * randomSeed);
    randomSeed %= modulus;
    int result = randomSeed % max;
    if(result < 0)
        result += max;
    return result;
}

coordinate generateValidCoordinate()
{
    // returns a coordinate which has not yet been occupied.
    int max_x = (SCREEN_X - gridSize - borderSize)/gridSize + 1;
    int max_y = (SCREEN_Y - gridSize - borderSize)/gridSize + 1;
    coordinate c = {PRNG(max_x) * gridSize + borderSize, PRNG(max_y) * gridSize +
        borderSize};
    return isValidCoordinate(c) ? c : generateValidCoordinate();
}

bool isValidCoordinate(coordinate c)
{
    //returns true if there is no intersection with any other game element, and false
        otherwise.
    if(intersectSnake(c) || intersectBarrier(c) || intersectFood(c) || intersectShield(c))
        return false;
    return true;
}

bool intersectSnake(coordinate c)
{
    int i;
    for(i = snakePositionsStart; i <= snakePositionsEnd; i++) //check if position matches
        any of the current snake coordinates
        if(c.x == snakePositions[i].x && c.y == snakePositions[i].y)
            return true;
    return false;
}

bool intersectBarrier(coordinate c)
{
    int i;
    for(i = 0; i < numBarriers; i++) //check if position matches any barrier
        if(c.x == barrierPositions[i].x && c.y == barrierPositions[i].y)
            return true;
    return false;
}

inline bool intersectFood(coordinate c)
{
    return c.x == foodPosition.x && c.y == foodPosition.y;
}

inline bool intersectShield(coordinate c)
{
    return c.x == shieldPosition.x && c.y == shieldPosition.y;
}

void generateNewFoodPosition()
{
    //generate and output new food position to TFT screen
    coordinate c = generateValidCoordinate();
    GUI_SetColor(GUI_RED);
    drawPixel(c.x, c.y);
    foodPosition = c;
}

void generateNewShieldPosition()
{
    //generate and output new shield position to TFT screen
    coordinate c = generateValidCoordinate();
    GUI_SetColor(GUI_GREEN);
    drawPixel(c.x, c.y);
    shieldPosition = c;
}

void generateNewBarrierPositions()
{
    //generates numBarriers many barriers and draws them on the TFT screen
    GUI_SetColor(GUI_BLUE);
    int i;
    for(i = 0; i < numBarriers; i++)
    {
        barrierPositions[i] = generateValidCoordinate();
        drawPixel(barrierPositions[i].x, barrierPositions[i].y);
    }
}

void updateDirection()
{
    //updates the current direction heading, given the current joystick inputs
    int x = getJoystickX();
    int y = getJoystickY();
    int newDirection = -1;
    if(previousDirection == DIR_NORTH || previousDirection == DIR_SOUTH)
    {
        //if snake currently going north or south, only change to either east or west
        if(x < 50)
            newDirection = DIR_WEST;
        else if(x > 200)
```

```
            newDirection = DIR_EAST;
        else
            newDirection = previousDirection; // if x movement not indicated, ignore (don't
                change direction)
    }
    else if (previousDirection == DIR_EAST || previousDirection == DIR_WEST)
    {
        if(y < 50)
            newDirection = DIR_SOUTH;
        else if(y > 200)
            newDirection = DIR_NORTH;
        else
            newDirection = previousDirection;
    }
    currentDirection = newDirection;
}

int main()
{
    InitializeModules();
    double music[259] = {261.63, 261.63, 261.63, 261.63, 261.63, 261.63, 261.63, 261.63,
        261.63, 261.63, 261.63, 261.63, 261.63, 261.63, 261.63, 261.63, 349.23, 349.23,
        349.23, 349.23, 349.23, 349.23, 349.23, 349.23, 440, 440, 440, 440, 440, 440, 440,
        440, 0, 440, 440, 440, 440, 440, 440, 440, 440, 440, 440, 440, 440, 440, 440, 440,
        440, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66,
        293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 0, 293.66, 293.66, 293.66,
        293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66,
        293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 329.63, 329.63, 329.63,
        329.63, 349.23, 349.23, 349.23, 349.23, 392.0, 392.0, 392.0, 392.0, 349.23,
        349.23, 349.23, 349.23, 349.23, 349.23, 349.23, 349.23, 349.23, 349.23, 349.23,
        349.23, 349.23, 349.23, 349.23, 349.23, 329.63, 329.63, 329.63, 329.63, 329.63,
        329.63, 329.63, 329.63, 329.63, 329.63, 329.63, 329.63, 329.63, 329.63, 329.63,
        329.63, 261.63, 261.63, 261.63, 261.63, 261.63, 261.63, 261.63, 261.63, 261.63,
        261.63, 261.63, 261.63, 261.63, 261.63, 261.63, 261.63, 349.23, 349.23, 349.23,
        349.23, 349.23, 349.23, 349.23, 349.23, 440, 440, 440, 440, 440, 440, 440, 440,
        523.25, 523.25, 523.25, 523.25, 523.25, 523.25, 523.25, 523.25, 523.25, 523.25,
        523.25, 523.25, 523.25, 523.25, 523.25, 523.25, 466.16, 466.16, 466.16, 466.16,
        466.16, 466.16, 466.16, 466.16, 440, 440, 440, 440, 440, 440, 440, 440, 293.66,
        293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66,
        293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 293.66, 329.63,
        329.63, 329.63, 329.63, 349.23, 349.23, 349.23, 349.23, 392.0, 392.0, 392.0,
        392.0, 0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0,
        392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0,
        392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0, 392.0};
    //fire emblem main theme song; good repeating BGM
    CyGlobalIntEnable; // Enable global interrupts

    for(;;)
    {
        if(state == STATE_NEW) //initialization state.
    {
        if(getJoystickState()) //if joystick pressed, start playing and clear lcd screen
        {
            state = STATE_PLAYING;
            displayLCDString("              ", 0);
            displayLCDString("              ", 1);
            musicTimer = 0;
        }
    }
    else if (state == STATE_PLAYING)
    {
        //keep updating direction, and display information to LCD. update music and
             movement with interrupts
        updateDirection();
        if(timeToMove)
        {
            moveSnake();
            timeToMove = false;
            displayLCDString("Curr. score: ", 0);
            LCD_Char_1_PrintNumber(currentScore);
            if(snakeShielded)
                displayLCDString("Shielded :)", 1);
            else
                displayLCDString("Unshielded!", 1);
            double nextNote = music[musicTimer % 259];
            if(nextNote == 0)
                Music_Clock_SetDivider(65535);
            else
                Music_Clock_SetDivider(12000000 / nextNote - 1);
        }
    }
    else if (state == STATE_DEAD)
    {
        //game finished. display final score; ask to play again
        displayLCDString("Final score: ", 0);
        LCD_Char_1_PrintNumber(currentScore);
        Music_Clock_Stop();
        displayLCDString("Press to Play!", 1);
        if(getJoystickState()) //if joystick pressed again, reset everything
        {
            InitializeVariables();
            InitializeTFT();
            state = STATE_PLAYING;
            displayLCDString("              ", 0);
            displayLCDString("              ", 1);
            musicTimer = 0;
            Music_Clock_Start();
        }
    }
}
```

```
}

/* [] END OF FILE */
```

6.2 Shared Variables Header (global_variables.h)

```
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/

/* [] END OF FILE */
#include <stdbool.h>

extern int musicTimer;
    //https://stackoverflow.com/questions/1433204/how-do-i-use-extern-to-share-variables-between-source-files
extern bool timeToMove;
```

6.3 Music Interrupt (Music_Interrupt.c)

```
/*******************************************************************************
* File Name: Music_Interrupt.c
* Version 1.70
*
*  Description:
*   API for controlling the state of an interrupt.
*
*
*  Note:
*
********************************************************************************
* Copyright 2008-2015, Cypress Semiconductor Corporation. All rights reserved.
* You may use this file only in accordance with the license, terms, conditions,
* disclaimers, and limitations in the end user license agreement accompanying
* the software package with which this file was provided.
*******************************************************************************/


#include <cydevice_trm.h>
#include <CyLib.h>
#include <Music_Interrupt.h>
#include "cyapicallbacks.h"

#if !defined(Music_Interrupt__REMOVED) /* Check for removal by optimization */

/*******************************************************************************
*  Place your includes, defines and code here
********************************************************************************/
/* `#START Music_Interrupt_intc` */
#include "Music_Timer.h"
#include "global_variables.h"
/* `#END` */

#ifndef CYINT_IRQ_BASE
#define CYINT_IRQ_BASE   16
#endif /* CYINT_IRQ_BASE */
#ifndef CYINT_VECT_TABLE
#define CYINT_VECT_TABLE ((cyisraddress **) CYREG_NVIC_VECT_OFFSET)
#endif /* CYINT_VECT_TABLE */

/* Declared in startup, used to set unused interrupts to. */
CY_ISR_PROTO(IntDefaultHandler);


/*******************************************************************************
* Function Name: Music_Interrupt_Start
********************************************************************************
*
```

```
* Summary:
*  Set up the interrupt and enable it. This function disables the interrupt,
*  sets the default interrupt vector, sets the priority from the value in the
*  Design Wide Resources Interrupt Editor, then enables the interrupt to the
*  interrupt controller.
*
* Parameters:
*   None
*
* Return:
*   None
*
*******************************************************************************/
void Music_Interrupt_Start(void)
{
    /* For all we know the interrupt is active. */
    Music_Interrupt_Disable();

    /* Set the ISR to point to the Music_Interrupt Interrupt. */
    Music_Interrupt_SetVector(&Music_Interrupt_Interrupt);

    /* Set the priority. */
    Music_Interrupt_SetPriority((uint8)Music_Interrupt_INTC_PRIOR_NUMBER);

    /* Enable it. */
    Music_Interrupt_Enable();
}


/*******************************************************************************
* Function Name: Music_Interrupt_StartEx
********************************************************************************
*
* Summary:
*  Sets up the interrupt and enables it. This function disables the interrupt,
*  sets the interrupt vector based on the address passed in, sets the priority
*  from the value in the Design Wide Resources Interrupt Editor, then enables
*  the interrupt to the interrupt controller.
*
*  When defining ISR functions, the CY_ISR and CY_ISR_PROTO macros should be
*  used to provide consistent definition across compilers:
*
*  Function definition example:
*   CY_ISR(MyISR)
*   {
*   }
*   Function prototype example:
*   CY_ISR_PROTO(MyISR);
*
* Parameters:
```

```
*   address: Address of the ISR to set in the interrupt vector table.
*
* Return:
*   None
*
*******************************************************************************/
void Music_Interrupt_StartEx(cyisraddress address)
{
    /* For all we know the interrupt is active. */
    Music_Interrupt_Disable();

    /* Set the ISR to point to the Music_Interrupt Interrupt. */
    Music_Interrupt_SetVector(address);

    /* Set the priority. */
    Music_Interrupt_SetPriority((uint8)Music_Interrupt_INTC_PRIOR_NUMBER);

    /* Enable it. */
    Music_Interrupt_Enable();
}


/*******************************************************************************
* Function Name: Music_Interrupt_Stop
********************************************************************************
*
* Summary:
*   Disables and removes the interrupt.
*
* Parameters:
*   None
*
* Return:
*   None
*
*******************************************************************************/
void Music_Interrupt_Stop(void)
{
    /* Disable this interrupt. */
    Music_Interrupt_Disable();

    /* Set the ISR to point to the passive one. */
    Music_Interrupt_SetVector(&IntDefaultHandler);
}


/*******************************************************************************
* Function Name: Music_Interrupt_Interrupt
********************************************************************************
*
```

```
* Summary:
*   The default Interrupt Service Routine for Music_Interrupt.
*
*   Add custom code between the coments to keep the next version of this file
*   from over writting your code.
*
* Parameters:
*
* Return:
*   None
*
*******************************************************************************/
CY_ISR(Music_Interrupt_Interrupt)
{
    #ifdef Music_Interrupt_INTERRUPT_INTERRUPT_CALLBACK
        Music_Interrupt_Interrupt_InterruptCallback();
    #endif /* Music_Interrupt_INTERRUPT_INTERRUPT_CALLBACK */

    /*  Place your Interrupt code here. */
    /* `#START Music_Interrupt_Interrupt` */
    musicTimer += 1;
    Music_Timer_ReadStatusRegister();
    /* `#END` */
}


/*******************************************************************************
* Function Name: Music_Interrupt_SetVector
********************************************************************************
*
* Summary:
*   Change the ISR vector for the Interrupt. Note calling Music_Interrupt_Start
*   will override any effect this method would have had. To set the vector
*   before the component has been started use Music_Interrupt_StartEx instead.
*
*   When defining ISR functions, the CY_ISR and CY_ISR_PROTO macros should be
*   used to provide consistent definition across compilers:
*
*   Function definition example:
*   CY_ISR(MyISR)
*   {
*   }
*
*   Function prototype example:
*     CY_ISR_PROTO(MyISR);
*
* Parameters:
*   address: Address of the ISR to set in the interrupt vector table.
*
* Return:
```

```
*   None
*
*******************************************************************************/
void Music_Interrupt_SetVector(cyisraddress address)
{
    cyisraddress * ramVectorTable;

    ramVectorTable = (cyisraddress *) *CYINT_VECT_TABLE;

    ramVectorTable[CYINT_IRQ_BASE + (uint32)Music_Interrupt__INTC_NUMBER] = address;
}


/*******************************************************************************
* Function Name: Music_Interrupt_GetVector
********************************************************************************
*
* Summary:
*   Gets the "address" of the current ISR vector for the Interrupt.
*
* Parameters:
*   None
*
* Return:
*   Address of the ISR in the interrupt vector table.
*
*******************************************************************************/
cyisraddress Music_Interrupt_GetVector(void)
{
    cyisraddress * ramVectorTable;

    ramVectorTable = (cyisraddress *) *CYINT_VECT_TABLE;

    return ramVectorTable[CYINT_IRQ_BASE + (uint32)Music_Interrupt__INTC_NUMBER];
}


/*******************************************************************************
* Function Name: Music_Interrupt_SetPriority
********************************************************************************
*
* Summary:
*   Sets the Priority of the Interrupt.
*
*   Note calling Music_Interrupt_Start or Music_Interrupt_StartEx will
*   override any effect this API would have had. This API should only be called
*   after Music_Interrupt_Start or Music_Interrupt_StartEx has been called.
*   To set the initial priority for the component, use the Design-Wide Resources
*   Interrupt Editor.
*
```

```
*   Note This API has no effect on Non-maskable interrupt NMI).
*
* Parameters:
*   priority: Priority of the interrupt, 0 being the highest priority
*            PSoC 3 and PSoC 5LP: Priority is from 0 to 7.
*            PSoC 4: Priority is from 0 to 3.
*
* Return:
*   None
*
*******************************************************************************/
void Music_Interrupt_SetPriority(uint8 priority)
{
    *Music_Interrupt_INTC_PRIOR = priority << 5;
}


/*******************************************************************************
* Function Name: Music_Interrupt_GetPriority
********************************************************************************
*
* Summary:
*   Gets the Priority of the Interrupt.
*
* Parameters:
*   None
*
* Return:
*   Priority of the interrupt, 0 being the highest priority
*    PSoC 3 and PSoC 5LP: Priority is from 0 to 7.
*    PSoC 4: Priority is from 0 to 3.
*
*******************************************************************************/
uint8 Music_Interrupt_GetPriority(void)
{
    uint8 priority;


    priority = *Music_Interrupt_INTC_PRIOR >> 5;

    return priority;
}


/*******************************************************************************
* Function Name: Music_Interrupt_Enable
********************************************************************************
*
* Summary:
*   Enables the interrupt to the interrupt controller. Do not call this function
*   unless ISR_Start() has been called or the functionality of the ISR_Start()
*   function, which sets the vector and the priority, has been called.
*
* Parameters:
*   None
*
* Return:
*   None
*
*******************************************************************************/
void Music_Interrupt_Enable(void)
{
    /* Enable the general interrupt. */
    *Music_Interrupt_INTC_SET_EN = Music_Interrupt__INTC_MASK;
}


/*******************************************************************************
* Function Name: Music_Interrupt_GetState
********************************************************************************
*
* Summary:
*   Gets the state (enabled, disabled) of the Interrupt.
*
* Parameters:
*   None
*
* Return:
*   1 if enabled, 0 if disabled.
*
*******************************************************************************/
uint8 Music_Interrupt_GetState(void)
{
    /* Get the state of the general interrupt. */
    return ((*Music_Interrupt_INTC_SET_EN & (uint32)Music_Interrupt__INTC_MASK) != 0u) ?
        1u:0u;
}


/*******************************************************************************
* Function Name: Music_Interrupt_Disable
********************************************************************************
*
* Summary:
*   Disables the Interrupt in the interrupt controller.
*
* Parameters:
*   None
*
* Return:
```

```
*   None
*
*******************************************************************************/
void Music_Interrupt_Disable(void)
{
    /* Disable the general interrupt. */
    *Music_Interrupt_INTC_CLR_EN = Music_Interrupt__INTC_MASK;
}


/*******************************************************************************
* Function Name: Music_Interrupt_SetPending
********************************************************************************
*
* Summary:
*   Causes the Interrupt to enter the pending state, a software method of
*   generating the interrupt.
*
* Parameters:
*   None
*
* Return:
*   None
*
* Side Effects:
*   If interrupts are enabled and the interrupt is set up properly, the ISR is
*   entered (depending on the priority of this interrupt and other pending
*   interrupts).
*
*******************************************************************************/
void Music_Interrupt_SetPending(void)
{
    *Music_Interrupt_INTC_SET_PD = Music_Interrupt__INTC_MASK;
}


/*******************************************************************************
* Function Name: Music_Interrupt_ClearPending
********************************************************************************
*
* Summary:
*   Clears a pending interrupt in the interrupt controller.
*
*   Note Some interrupt sources are clear-on-read and require the block
*   interrupt/status register to be read/cleared with the appropriate block API
*   (GPIO, UART, and so on). Otherwise the ISR will continue to remain in
*   pending state even though the interrupt itself is cleared using this API.
*
* Parameters:
*   None
```

```
*
* Return:
*   None
*
*******************************************************************************/
void Music_Interrupt_ClearPending(void)
{
    *Music_Interrupt_INTC_CLR_PD = Music_Interrupt__INTC_MASK;
}

#endif /* End check for removal by optimization */


/* [] END OF FILE */
```

6.4 Movement Interrupt (Movement_Interrupt.c)

```
/*******************************************************************************
* File Name: Movement_Interrupt.c
* Version 1.70
*
*  Description:
*   API for controlling the state of an interrupt.
*
*
*  Note:
*
********************************************************************************
* Copyright 2008-2015, Cypress Semiconductor Corporation. All rights reserved.
* You may use this file only in accordance with the license, terms, conditions,
* disclaimers, and limitations in the end user license agreement accompanying
* the software package with which this file was provided.
*******************************************************************************/


#include <cydevice_trm.h>
#include <CyLib.h>
#include <Movement_Interrupt.h>
#include "cyapicallbacks.h"

#if !defined(Movement_Interrupt__REMOVED) /* Check for removal by optimization */

/*******************************************************************************
*  Place your includes, defines and code here
********************************************************************************/
/* `#START Movement_Interrupt_intc` */
#include "Movement_Timer.h"
#include "global_variables.h"

/* `#END` */

#ifndef CYINT_IRQ_BASE
#define CYINT_IRQ_BASE   16
#endif /* CYINT_IRQ_BASE */
#ifndef CYINT_VECT_TABLE
#define CYINT_VECT_TABLE ((cyisraddress **) CYREG_NVIC_VECT_OFFSET)
#endif /* CYINT_VECT_TABLE */

/* Declared in startup, used to set unused interrupts to. */
CY_ISR_PROTO(IntDefaultHandler);


/*******************************************************************************
* Function Name: Movement_Interrupt_Start
********************************************************************************
```

```
*
* Summary:
*  Set up the interrupt and enable it. This function disables the interrupt,
*  sets the default interrupt vector, sets the priority from the value in the
*  Design Wide Resources Interrupt Editor, then enables the interrupt to the
*  interrupt controller.
*
* Parameters:
*   None
*
* Return:
*   None
*
*******************************************************************************/
void Movement_Interrupt_Start(void)
{
    /* For all we know the interrupt is active. */
    Movement_Interrupt_Disable();

    /* Set the ISR to point to the Movement_Interrupt Interrupt. */
    Movement_Interrupt_SetVector(&Movement_Interrupt_Interrupt);

    /* Set the priority. */
    Movement_Interrupt_SetPriority((uint8)Movement_Interrupt_INTC_PRIOR_NUMBER);

    /* Enable it. */
    Movement_Interrupt_Enable();
}


/*******************************************************************************
* Function Name: Movement_Interrupt_StartEx
********************************************************************************
*
* Summary:
*  Sets up the interrupt and enables it. This function disables the interrupt,
*  sets the interrupt vector based on the address passed in, sets the priority
*  from the value in the Design Wide Resources Interrupt Editor, then enables
*  the interrupt to the interrupt controller.
*
*  When defining ISR functions, the CY_ISR and CY_ISR_PROTO macros should be
*  used to provide consistent definition across compilers:
*
*  Function definition example:
*   CY_ISR(MyISR)
*   {
*   }
*   Function prototype example:
*   CY_ISR_PROTO(MyISR);
*
```

```
* Parameters:
*   address: Address of the ISR to set in the interrupt vector table.
*
* Return:
*   None
*
*******************************************************************************/
void Movement_Interrupt_StartEx(cyisraddress address)
{
    /* For all we know the interrupt is active. */
    Movement_Interrupt_Disable();

    /* Set the ISR to point to the Movement_Interrupt Interrupt. */
    Movement_Interrupt_SetVector(address);

    /* Set the priority. */
    Movement_Interrupt_SetPriority((uint8)Movement_Interrupt_INTC_PRIOR_NUMBER);

    /* Enable it. */
    Movement_Interrupt_Enable();
}


/*******************************************************************************
* Function Name: Movement_Interrupt_Stop
********************************************************************************
*
* Summary:
*   Disables and removes the interrupt.
*
* Parameters:
*   None
*
* Return:
*   None
*
*******************************************************************************/
void Movement_Interrupt_Stop(void)
{
    /* Disable this interrupt. */
    Movement_Interrupt_Disable();

    /* Set the ISR to point to the passive one. */
    Movement_Interrupt_SetVector(&IntDefaultHandler);
}


/*******************************************************************************
* Function Name: Movement_Interrupt_Interrupt
********************************************************************************
```

```
*
* Summary:
*   The default Interrupt Service Routine for Movement_Interrupt.
*
*   Add custom code between the coments to keep the next version of this file
*   from over writting your code.
*
* Parameters:
*
* Return:
*   None
*
*******************************************************************************/
CY_ISR(Movement_Interrupt_Interrupt)
{
    #ifdef Movement_Interrupt_INTERRUPT_INTERRUPT_CALLBACK
        Movement_Interrupt_Interrupt_InterruptCallback();
    #endif /* Movement_Interrupt_INTERRUPT_INTERRUPT_CALLBACK */

    /*  Place your Interrupt code here. */
    /* `#START Movement_Interrupt_Interrupt` */
    timeToMove = true;
    Movement_Timer_ReadStatusRegister();

    /* `#END` */
}


/*******************************************************************************
* Function Name: Movement_Interrupt_SetVector
********************************************************************************
*
* Summary:
*   Change the ISR vector for the Interrupt. Note calling Movement_Interrupt_Start
*   will override any effect this method would have had. To set the vector
*   before the component has been started use Movement_Interrupt_StartEx instead.
*
*   When defining ISR functions, the CY_ISR and CY_ISR_PROTO macros should be
*   used to provide consistent definition across compilers:
*
*   Function definition example:
*   CY_ISR(MyISR)
*   {
*   }
*
*   Function prototype example:
*     CY_ISR_PROTO(MyISR);
*
* Parameters:
*   address: Address of the ISR to set in the interrupt vector table.
```

```
*
* Return:
*   None
*
*******************************************************************************/
void Movement_Interrupt_SetVector(cyisraddress address)
{
    cyisraddress * ramVectorTable;

    ramVectorTable = (cyisraddress *) *CYINT_VECT_TABLE;

    ramVectorTable[CYINT_IRQ_BASE + (uint32)Movement_Interrupt__INTC_NUMBER] = address;
}


/*******************************************************************************
* Function Name: Movement_Interrupt_GetVector
********************************************************************************
*
* Summary:
*   Gets the "address" of the current ISR vector for the Interrupt.
*
* Parameters:
*   None
*
* Return:
*   Address of the ISR in the interrupt vector table.
*
*******************************************************************************/
cyisraddress Movement_Interrupt_GetVector(void)
{
    cyisraddress * ramVectorTable;

    ramVectorTable = (cyisraddress *) *CYINT_VECT_TABLE;

    return ramVectorTable[CYINT_IRQ_BASE + (uint32)Movement_Interrupt__INTC_NUMBER];
}


/*******************************************************************************
* Function Name: Movement_Interrupt_SetPriority
********************************************************************************
*
* Summary:
*   Sets the Priority of the Interrupt.
*
*   Note calling Movement_Interrupt_Start or Movement_Interrupt_StartEx will
*   override any effect this API would have had. This API should only be called
*   after Movement_Interrupt_Start or Movement_Interrupt_StartEx has been called.
*   To set the initial priority for the component, use the Design-Wide Resources
*   Interrupt Editor.
*
*   Note This API has no effect on Non-maskable interrupt NMI).
*
* Parameters:
*   priority: Priority of the interrupt, 0 being the highest priority
*            PSoC 3 and PSoC 5LP: Priority is from 0 to 7.
*            PSoC 4: Priority is from 0 to 3.
*
* Return:
*   None
*
*******************************************************************************/
void Movement_Interrupt_SetPriority(uint8 priority)
{
    *Movement_Interrupt_INTC_PRIOR = priority << 5;
}


/*******************************************************************************
* Function Name: Movement_Interrupt_GetPriority
********************************************************************************
*
* Summary:
*   Gets the Priority of the Interrupt.
*
* Parameters:
*   None
*
* Return:
*   Priority of the interrupt, 0 being the highest priority
*    PSoC 3 and PSoC 5LP: Priority is from 0 to 7.
*    PSoC 4: Priority is from 0 to 3.
*
*******************************************************************************/
uint8 Movement_Interrupt_GetPriority(void)
{
    uint8 priority;


    priority = *Movement_Interrupt_INTC_PRIOR >> 5;

    return priority;
}


/*******************************************************************************
* Function Name: Movement_Interrupt_Enable
********************************************************************************
*
```

```
* Summary:
*   Enables the interrupt to the interrupt controller. Do not call this function
*   unless ISR_Start() has been called or the functionality of the ISR_Start()
*   function, which sets the vector and the priority, has been called.
*
* Parameters:
*   None
*
* Return:
*   None
*
*******************************************************************************/
void Movement_Interrupt_Enable(void)
{
    /* Enable the general interrupt. */
    *Movement_Interrupt_INTC_SET_EN = Movement_Interrupt__INTC_MASK;
}


/*******************************************************************************
* Function Name: Movement_Interrupt_GetState
********************************************************************************
*
* Summary:
*   Gets the state (enabled, disabled) of the Interrupt.
*
* Parameters:
*   None
*
* Return:
*   1 if enabled, 0 if disabled.
*
*******************************************************************************/
uint8 Movement_Interrupt_GetState(void)
{
    /* Get the state of the general interrupt. */
    return ((*Movement_Interrupt_INTC_SET_EN & (uint32)Movement_Interrupt__INTC_MASK) !=
         0u) ? 1u:0u;
}


/*******************************************************************************
* Function Name: Movement_Interrupt_Disable
********************************************************************************
*
* Summary:
*   Disables the Interrupt in the interrupt controller.
*
* Parameters:
*   None
```

```
*
* Return:
*   None
*
*******************************************************************************/
void Movement_Interrupt_Disable(void)
{
    /* Disable the general interrupt. */
    *Movement_Interrupt_INTC_CLR_EN = Movement_Interrupt__INTC_MASK;
}


/*******************************************************************************
* Function Name: Movement_Interrupt_SetPending
********************************************************************************
*
* Summary:
*   Causes the Interrupt to enter the pending state, a software method of
*   generating the interrupt.
*
* Parameters:
*   None
*
* Return:
*   None
*
* Side Effects:
*   If interrupts are enabled and the interrupt is set up properly, the ISR is
*   entered (depending on the priority of this interrupt and other pending
*   interrupts).
*
*******************************************************************************/
void Movement_Interrupt_SetPending(void)
{
    *Movement_Interrupt_INTC_SET_PD = Movement_Interrupt__INTC_MASK;
}


/*******************************************************************************
* Function Name: Movement_Interrupt_ClearPending
********************************************************************************
*
* Summary:
*   Clears a pending interrupt in the interrupt controller.
*
*   Note Some interrupt sources are clear-on-read and require the block
*   interrupt/status register to be read/cleared with the appropriate block API
*   (GPIO, UART, and so on). Otherwise the ISR will continue to remain in
*   pending state even though the interrupt itself is cleared using this API.
*
```

```
* Parameters:
*   None
*
* Return:
*   None
*
*******************************************************************************/
void Movement_Interrupt_ClearPending(void)
{
    *Movement_Interrupt_INTC_CLR_PD = Movement_Interrupt__INTC_MASK;
}

#endif /* End check for removal by optimization */


/* [] END OF FILE */
```