Advances in Engineering Innovation (2024) Volume 8 EWA Publishing
Published online: 28 June 2024 DOI: 10.54254/2977-3903/8/2024074

Creating a Parkour Game

Chen Yuanhao

Arizona State University, Tempe AZ85281, USA

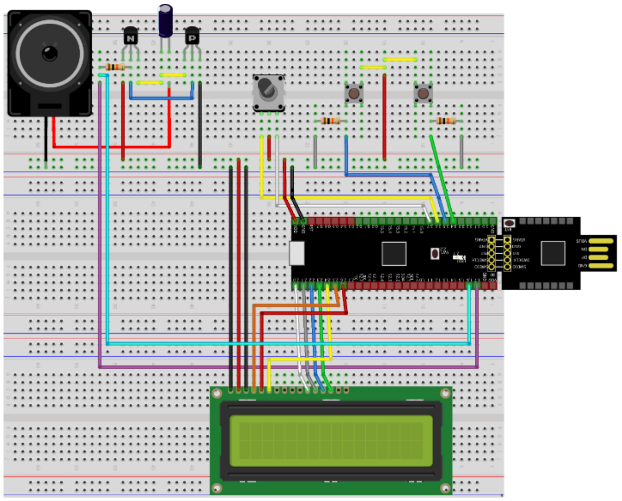
1810310168@qq.com

Abstract. This article primarily discusses how to use PWM, interrupt signals, and timers to create a parkour game. We also need two relatively independent PWMs to produce sounds: one for playing background music in a loop, and the other for playing collision, level completion, and prompt sounds at the end of the game.

Keywords: PWM, parkour game, CONTROL_ISR, TopDesign

1. Prelab Work

In the preparation work, we need to connect the circuits according to the figure below and ensure they are correct.



The following are all the materials needed:

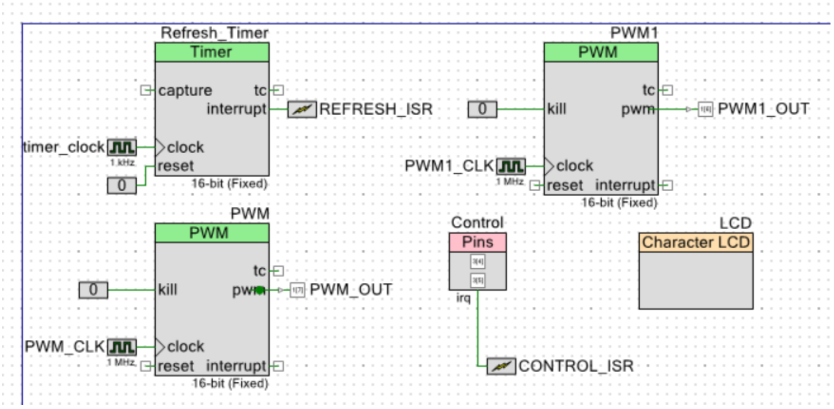
Materials	Quantity
PSoC 5LP	1
LCD Screen	1
Resistor (10kΩ±5%)	3
Capacitor (50V/100μF)	1
Transistor (2N2222A-D)	1
Transistor (2N3906)	1
Speaker (8Ω-1W)	1
Bread Board	3
Wires	Several
Button	2

2. Game

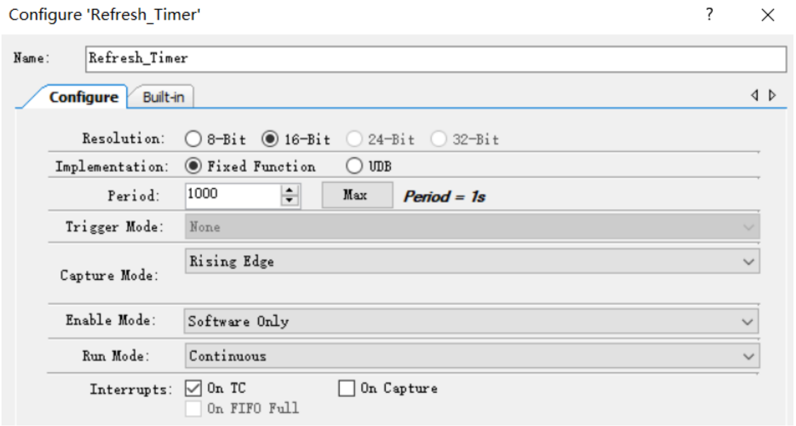
Description: This section focuses on how to use PWM, interrupt signals, and timers to create a parkour game. By pressing the button, the movement of the characters is controlled until they clear the level.

Requirement: Press the control button to move the character up and down, and then refresh the position when a new interrupt signal is given. Set a map with some obstacles in the system. When a character touches an obstacle, they lose one of their lives. When all three lives are lost, the game ends and the screen displays "Game Over". When the level is cleared successfully, the screen displays "Level Up!" and the map speed increases. Background music needs to be played in a loop during the entire game, and independent sounds are needed when crossing levels or hitting obstacles. The shapes of the character and obstacles should be defined by us.

Procedure: First, we need to identify the parts required for this experiment. Two buttons are needed to control the movement of the characters, so we need two "Input Pins". To keep the map moving to the left, we need a timer that runs throughout the game, constantly receiving interrupt signals and refreshing the screen. We also need two relatively independent PWMs to produce sounds: one for playing background music in a loop, and the other for playing collision, level completion, and prompt sounds at the end of the game.



Let's start with TopDesign. First, we design a refresh timer that runs throughout the game. To make the map move to the left more slowly, we adjusted the clock pulse of the timer to a lower 1kHz. Connect a logic low level to the reset interface and an interrupt signal named "REFRESH_ISR" to the interrupt interface. Then, right-click the configuration, change the resolution to 16-bit, change the implementation to a fixed function, and check "On TC" in Interrupts.


Configure 'Refresh_Timer'

Name: Refresh_Timer

Configure | Built-in

Resolution: ○ 8-Bit ● 16-Bit ○ 24-Bit ○ 32-Bit

Implementation: ● Fixed Function ○ UDB

Period: 1000 Max *Period = 1s*

Trigger Mode: None

Capture Mode: Rising Edge

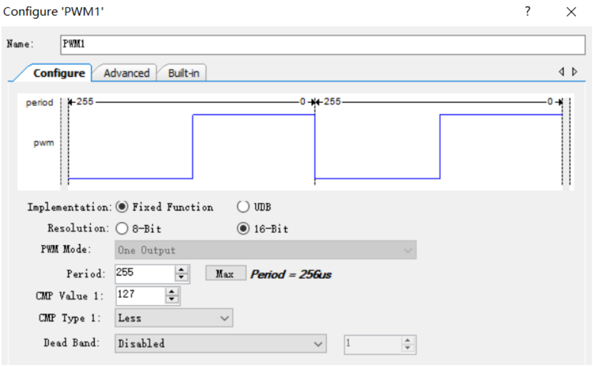
Enable Mode: Software Only

Run Mode: Continuous

Interrupts: ☑ On TC □ On Capture

□ On FIFO Full

As shown above, we have already set the timer. The next step is to design two PWMs for playing music. These two PWMs have the same parameters except for the names and the music they play. So, we create two PWMs, one called PWM, which is used to play the prompt tone, and the other named PWM1, which is used to play the background music in a loop. We connect a logic 0 to both PWM kill ports and a "Digital Output Pin" to the interrupt port, which are renamed PWM_OUT and PWM1_OUT. The clock pulses of both are adjusted to 1MHz. In the configuration, change both configurations to: Fixed Function and 16-bit.



Configure 'PWM1'

Name: PWM1

Configure | Advanced | Built-in

Implementation: ● Fixed Function ○ UDB

Resolution: ○ 8-Bit ● 16-Bit

PWM Mode: One Output

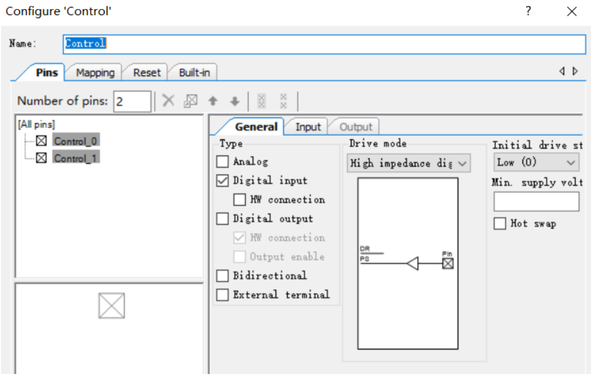
Period: 255 Max *Period = 256us*

CMP Value 1: 127

CMP Type 1: Less

Dead Band: Disabled 1

Then, we add two buttons to control the movement of the characters. We add a "digital input pin" and change the number of pins to 2 because we need one button to control upward movement and the other to control downward movement. Cancel the HW connection and change the driving mode to "High Impedance". Rename the button to Control, add an interrupt to it, rename it to CONTROL_ISR, and connect it to the IRQ port. Finally, we added an LCD and designed several icons to represent people and obstacles, completing all the previous designs.



Configure 'Control'

Name: Control

Pins | Mapping | Reset | Built-in

Number of pins: 2

[All pins]
- Control_0
- Control_1

General | Input | Output

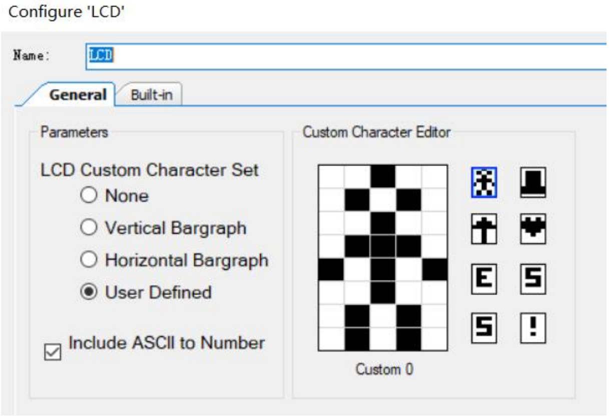
Type
- ☐ Analog
- ☑ Digital input
  - ☐ HW connection
- ☐ Digital output
  - ☑ HW connection
  - ☐ Output enable
- ☐ Bidirectional
- ☐ External terminal

Drive mode: High impedance dig

Initial drive st: Low (0)

Min. supply volt:

☐ Hot swap



Configure 'LCD'

Name: LCD

General | Built-in

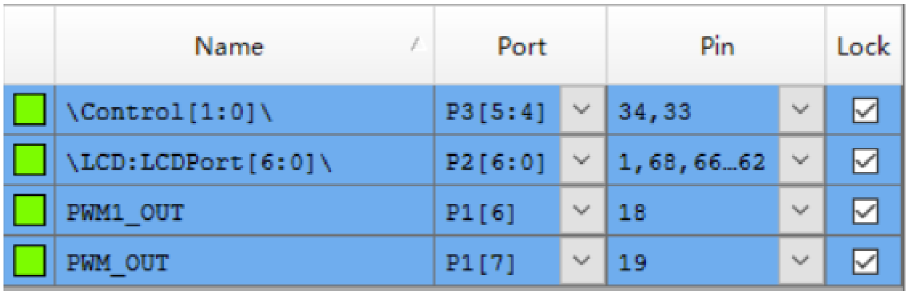
Parameters

LCD Custom Character Set
- ○ None
- ○ Vertical Bargraph
- ○ Horizontal Bargraph
- ● User Defined

☑ Include ASCII to Number

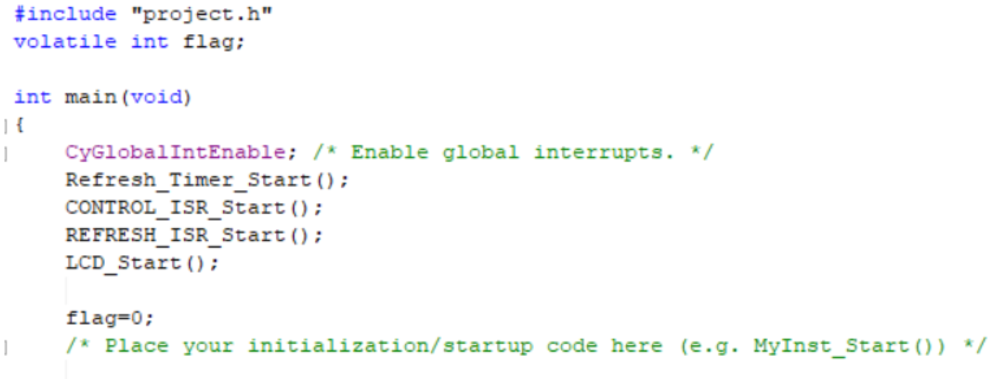
Custom Character Editor

Finally, we connect the control buttons to P3[5:4], the LCD to P2[6:0], PWM1 for playing background music to P1[6], and PWM for playing prompt tones to P1[7].



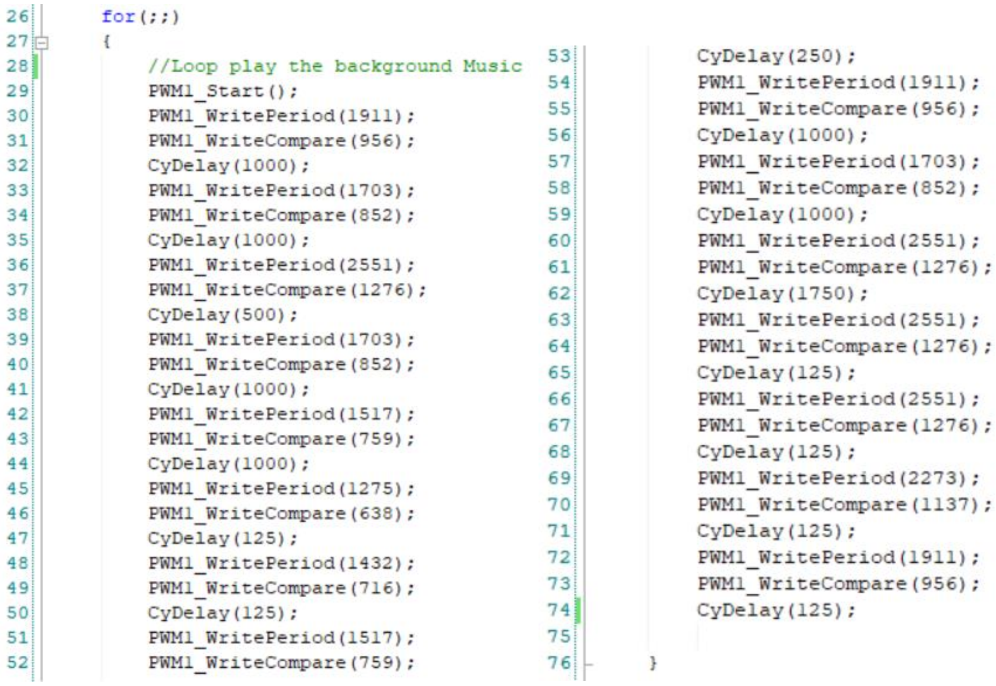
	Name	Port	Pin	Lock
🟩	\Control[1:0]\	P3[5:4]	34,33	☑
🟩	\LCD:LCDPort[6:0]\	P2[6:0]	1,68,66…62	☑
🟩	PWM1_OUT	P1[6]	18	☑
🟩	PWM_OUT	P1[7]	19	☑

Next is the most important part: organizing ideas and writing code. First, we believe that, except for the background music which is played in a loop, everything else is controlled by the corresponding interrupt signal. Therefore, the code for playing background music should be placed in the loop program of the main function. The main program only needs to turn on the LCD, timer, refresh interrupt signal, and Control interrupt signal. In the main program, it is also necessary to define a variable flag that can be recognized in any interrupt signal and initialize it to 0.



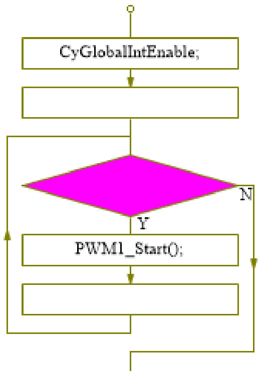
```
#include "project.h"
volatile int flag;

int main(void)
{
    CyGlobalIntEnable; /* Enable global interrupts. */
    Refresh_Timer_Start();
    CONTROL_ISR_Start();
    REFRESH_ISR_Start();
    LCD_Start();

    flag=0;
    /* Place your initialization/startup code here (e.g. MyInst_Start()) */
```

For the background music played in a loop, we first activate PWM1, and then write the frequency, compare, and duration of the tones to be played in sequence. The music I used here is "Never Gonna Give You Up".

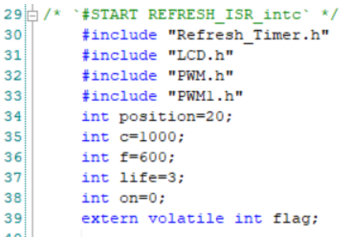


```
    for(;;)
    {
        //Loop play the background Music
        PWM1_Start();
        PWM1_WritePeriod(1911);
        PWM1_WriteCompare(956);
        CyDelay(1000);
        PWM1_WritePeriod(1703);
        PWM1_WriteCompare(852);
        CyDelay(1000);
        PWM1_WritePeriod(2551);
        PWM1_WriteCompare(1276);
        CyDelay(500);
        PWM1_WritePeriod(1703);
        PWM1_WriteCompare(852);
        CyDelay(1000);
        PWM1_WritePeriod(1517);
        PWM1_WriteCompare(759);
        CyDelay(1000);
        PWM1_WritePeriod(1275);
        PWM1_WriteCompare(638);
        CyDelay(125);
        PWM1_WritePeriod(1432);
        PWM1_WriteCompare(716);
        CyDelay(125);
        PWM1_WritePeriod(1517);
        PWM1_WriteCompare(759);
        CyDelay(250);
        PWM1_WritePeriod(1911);
        PWM1_WriteCompare(956);
        CyDelay(1000);
        PWM1_WritePeriod(1703);
        PWM1_WriteCompare(852);
        CyDelay(1000);
        PWM1_WritePeriod(2551);
        PWM1_WriteCompare(1276);
        CyDelay(1750);
        PWM1_WritePeriod(2551);
        PWM1_WriteCompare(1276);
        CyDelay(125);
        PWM1_WritePeriod(2551);
        PWM1_WriteCompare(1276);
        CyDelay(125);
        PWM1_WritePeriod(2273);
        PWM1_WriteCompare(1137);
        CyDelay(125);
        PWM1_WritePeriod(1911);
        PWM1_WriteCompare(956);
        CyDelay(125);

    }
```

The following is the logic diagram of the main function:

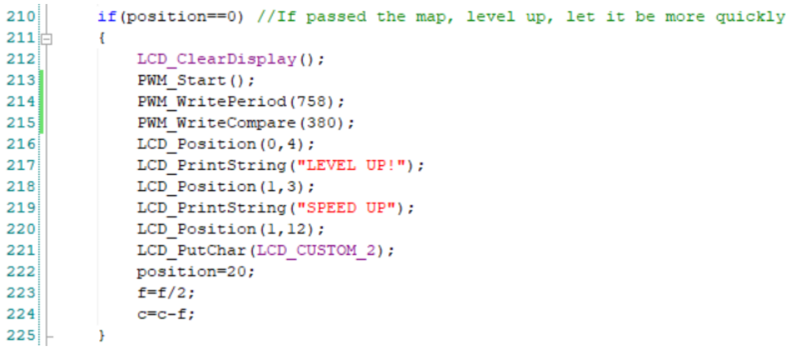


Then comes the code in REFRESH_ISR, which has many functions. First, we need header files including "Refresh_Timer.h", "LCD.h", "PWM.h", and "PWM1.h", indicating that this interrupt signal controls the timer, LCD, and two PWMs. We need to set the map. For example, here I set the length of the map to 20 frames, so I define an integer variable position=20. Let the frequency and compare of the initial map moving speed be F and C, and assign them initial values of 600 and 1000. Then, define a decision variable on for detecting whether a character hits an obstacle, initially set to 0. There is also a life value, life=3. Finally, an instant variable flag is defined, which controls and keeps the position of the characters.



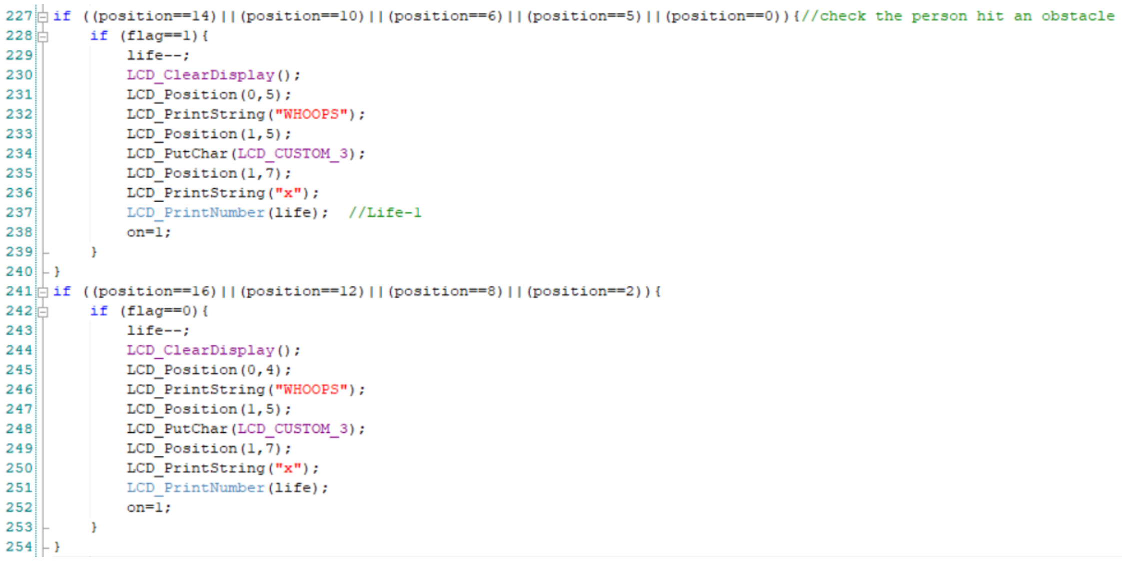
```
/* `#START REFRESH_ISR_intc` */
    #include "Refresh_Timer.h"
    #include "LCD.h"
    #include "PWM.h"
    #include "PWM1.h"
    int position=20;
    int c=1000;
    int f=600;
    int life=3;
    int on=0;
    extern volatile int flag;
```

First, we should speed up every cycle of the map. We defined the variable c before, and use c as the cycle of each run. Then, design the map. We clearly describe the position of obstacles on the map in the form of coordinates, using our homemade symbols. For example, if there is an obstacle at (0,3), our coordinate should be (0, position-17). With the step-by-step reception of interrupt signals, these obstacles need to be translated to the left in turn. So here, we let the position decrease by one in turn. When the game is cleared, that is, when the position moves to 0 on the left, we need to start over and speed up. Therefore, add the judgment condition: when position==0, clear the screen, PWM plays the level completion prompt tone, and display "Level Up!", reset the position to 20, and reduce the frequency and compare, to increase the speed.



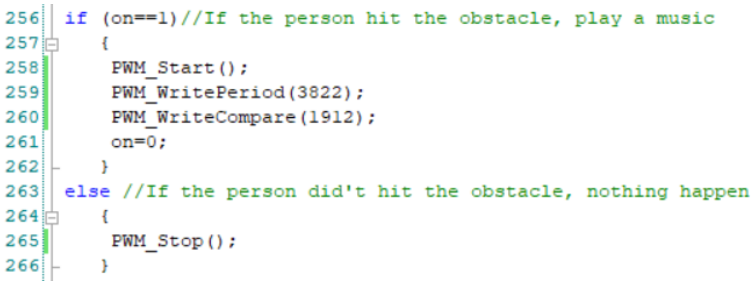
```
    if(position==0) //If passed the map, level up, let it be more quickly
    {
        LCD_ClearDisplay();
        PWM_Start();
        PWM_WritePeriod(758);
        PWM_WriteCompare(380);
        LCD_Position(0,4);
        LCD_PrintString("LEVEL UP!");
        LCD_Position(1,3);
        LCD_PrintString("SPEED UP");
        LCD_Position(1,12);
        LCD_PutChar(LCD_CUSTOM_2);
        position=20;
        f=f/2;
        c=c-f;
    }
```

Next, it is necessary to detect whether a character has hit an obstacle. The current position of the character, represented by flag, will collide if it overlaps with the coordinates of any obstacle when it is detected that the character is in the position of (flag,0). If there is no overlap, there is no collision, and the game continues to run normally. In case of a collision, clear the screen and display "Whoops!", reduce the life count by 1 and display the remaining life count, and set the variable on=1 to indicate a collision, and then continue the game.



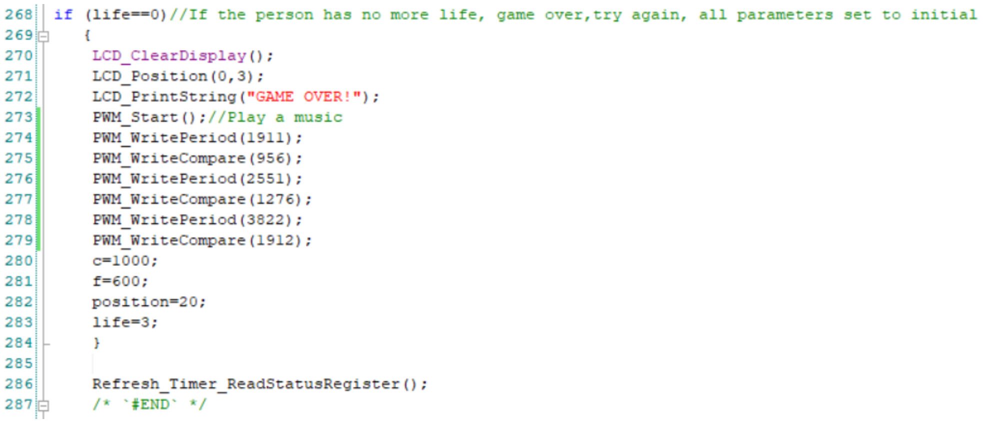
```
if ((position==14)||(position==10)||(position==6)||(position==5)||(position==0)){//check the person hit an obstacle
    if (flag==1){
        life--;
        LCD_ClearDisplay();
        LCD_Position(0,5);
        LCD_PrintString("WHOOPS");
        LCD_Position(1,5);
        LCD_PutChar(LCD_CUSTOM_3);
        LCD_Position(1,7);
        LCD_PrintString("x");
        LCD_PrintNumber(life);  //Life-1
        on=1;
    }
}
if ((position==16)||(position==12)||(position==8)||(position==2)){
    if (flag==0){
        life--;
        LCD_ClearDisplay();
        LCD_Position(0,4);
        LCD_PrintString("WHOOPS");
        LCD_Position(1,5);
        LCD_PutChar(LCD_CUSTOM_3);
        LCD_Position(1,7);
        LCD_PrintString("x");
        LCD_PrintNumber(life);
        on=1;
    }
}
```

When the character collides with obstacles, PWM needs to produce a prompt tone, controlled by the variable on. When on==1, PWM turns on and plays a specific prompt tone, then resets on to its initial value. When there is no collision, PWM should be turned off.



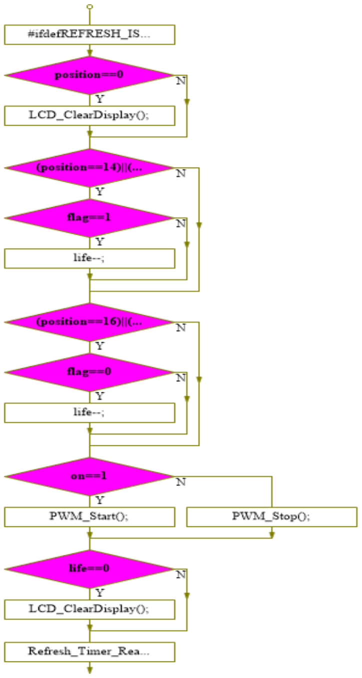
```
if (on==1)//If the person hit the obstacle, play a music
    {
     PWM_Start();
     PWM_WritePeriod(3822);
     PWM_WriteCompare(1912);
     on=0;
    }
else //If the person did't hit the obstacle, nothing happen
    {
     PWM_Stop();
    }
```

Then there is the code for detecting life. When life reaches 0, the game ends, the screen displays "Game Over", all parameters return to their initial values, PWM plays a prompt tone, and life is re-assigned to 3. At the end of the entire code, it is necessary to detect the current state of the timer, to maintain the original state and respond when each interrupt signal occurs.

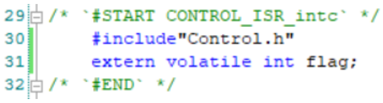


```
if (life==0)//If the person has no more life, game over,try again, all parameters set to initial
   {
    LCD_ClearDisplay();
    LCD_Position(0,3);
    LCD_PrintString("GAME OVER!");
    PWM_Start();//Play a music
    PWM_WritePeriod(1911);
    PWM_WriteCompare(956);
    PWM_WritePeriod(2551);
    PWM_WriteCompare(1276);
    PWM_WritePeriod(3822);
    PWM_WriteCompare(1912);
    c=1000;
    f=600;
    position=20;
    life=3;
    }

    Refresh_Timer_ReadStatusRegister();
    /* `#END` */
```

The following is the logic diagram of REFRESH_ISR:

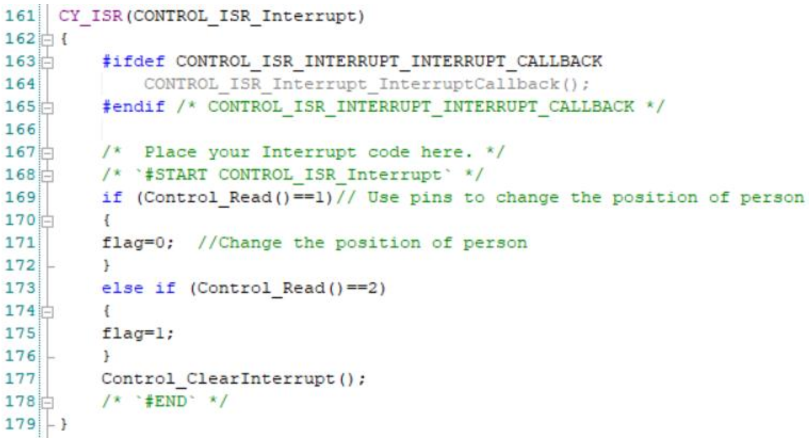


Finally, there is the code for the control button. This part is relatively simple, requiring the header file "Control.h" first, because it controls the interrupt signal of the control button. Then the previously defined variable flag is introduced, because the control button changes the value of flag, to achieve the function of changing the position of characters.



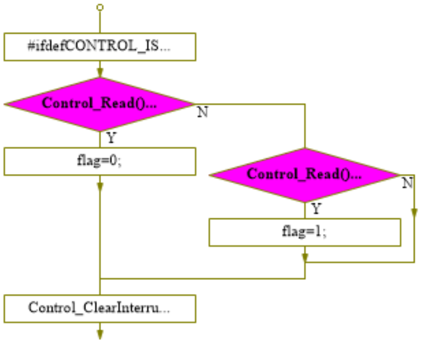
```
/* `#START CONTROL_ISR_intc` */
    #include"Control.h"
    extern volatile int flag;
/* `#END` */
```

When we press the left button, the character moves up, so flag=0, because flag represents the position of the line where the character is located. Similarly, when the right button is pressed, the character moves down. At the end of the interrupt signal, we need to clear the current interrupt to maintain the previous state.

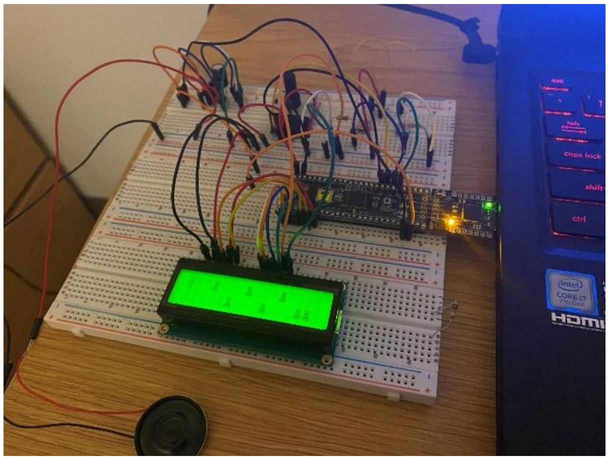


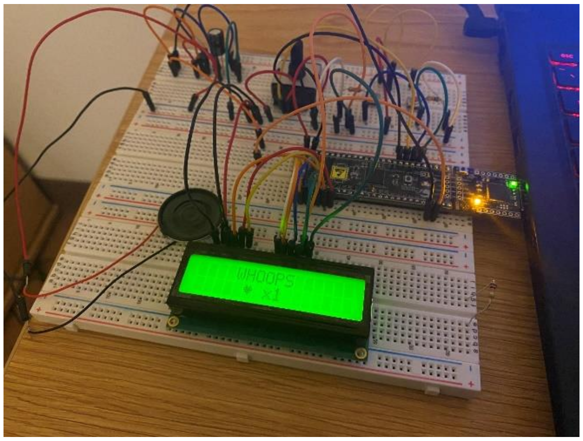
```
CY_ISR(CONTROL_ISR_Interrupt)
{
    #ifdef CONTROL_ISR_INTERRUPT_INTERRUPT_CALLBACK
        CONTROL_ISR_Interrupt_InterruptCallback();
    #endif /* CONTROL_ISR_INTERRUPT_INTERRUPT_CALLBACK */

    /*  Place your Interrupt code here. */
    /* `#START CONTROL_ISR_Interrupt` */
    if (Control_Read()==1)// Use pins to change the position of person
    {
    flag=0;  //Change the position of person
    }
    else if (Control_Read()==2)
    {
    flag=1;
    }
    Control_ClearInterrupt();
    /* `#END` */
}
```

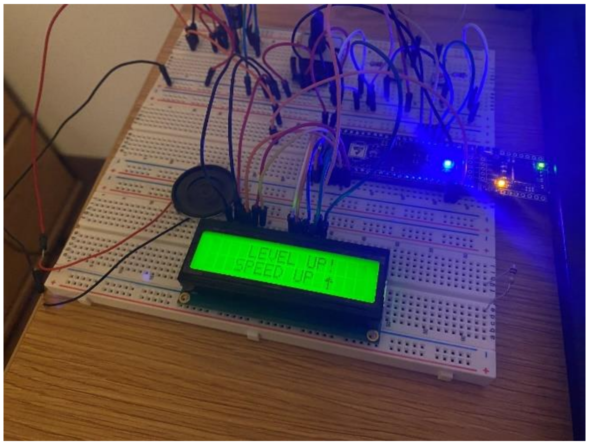
The following is the logic diagram of CONTROL_ISR:

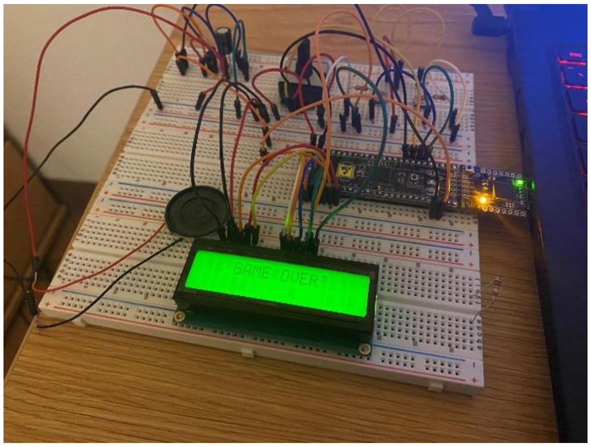


The final experimental results are as follows:









3. Summary

In this experiment, the application of speakers, position detection, and control signals is comprehensively investigated. Adding random functions and various props to the game would make it more complete, although more complicated.

References

[1] Semtech Corporation. (2016). Chirp Signal Processor: European, EP2975814A1. Retrieved January 20, 2016, from https://patents.google.com/patent/EP2975814A1/en

[2] Ephrat, A., & Peleg, S. (2017). Vid2speech: Speech reconstruction from silent video. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing* (pp. 5095-5099). IEEE.

[3] Strap Technologics. (2021, December 4). Retrieved from https://strap.tech/

Appendix: Complete Code

Part 1:

REFRESH_ISR.c

```
/* `#START REFRESH_ISR_intc` */
        #include "Refresh_Timer.h"
        #include "LCD.h"
        #include "PWM.h"
        #include "PWM1.h"
        int position=20;
        int c=1000;
        int f=600;
        int life=3;
        int on=0;
        extern volatile int flag;

    /* `#END` */
CY_ISR(REFRESH_ISR_Interrupt)
{
        #ifdef REFRESH_ISR_INTERRUPT_INTERRUPT_CALLBACK
                REFRESH_ISR_Interrupt_InterruptCallback();
        #endif /* REFRESH_ISR_INTERRUPT_INTERRUPT_CALLBACK */

        /*    Place your Interrupt code here. */
        /* `#START REFRESH_ISR_Interrupt` */
        Refresh_Timer_WritePeriod(c);
        LCD_ClearDisplay();// Define the position of person
        LCD_Position(0,position-17);// Design the game map
        LCD_PutChar(LCD_CUSTOM_1);
        LCD_Position(1,position-15);
        LCD_PutChar(LCD_CUSTOM_1);
        LCD_Position(1,position-14);
        LCD_PrintString(" ");
        LCD_Position(0,position-13);
        LCD_PutChar(LCD_CUSTOM_1);
        LCD_Position(1,position-11);
        LCD_PutChar(LCD_CUSTOM_1);
        LCD_Position(1,position-10);
        LCD_PrintString(" ");
        LCD_Position(0,position-9);
        LCD_PutChar(LCD_CUSTOM_1);
        LCD_Position(1,position-7);
        LCD_PutChar(LCD_CUSTOM_1);
        LCD_Position(1,position-6);
        LCD_PutChar(LCD_CUSTOM_1);
        LCD_Position(1,position-5);
        LCD_PrintString(" ");
        LCD_Position(0,position-3);
        LCD_PutChar(LCD_CUSTOM_1);
        LCD_Position(1,position-1);
        LCD_PutChar(LCD_CUSTOM_1);
        LCD_Position(1,position);
        LCD_PrintString(" ");
        LCD_Position(flag,0);
        LCD_PutChar(LCD_CUSTOM_0);
        position--;                         //Let the map go left each time
        if(position==0) //If passed the map, level up, let it be more quickly
        {
                LCD_ClearDisplay();
                PWM_Start();
                PWM_WritePeriod(758);
                PWM_WriteCompare(380);
                LCD_Position(0,4);
                LCD_PrintString("LEVEL UP!");
                LCD_Position(1,3);
                LCD_PrintString("SPEED UP");
                LCD_Position(1,12);
                LCD_PutChar(LCD_CUSTOM_2);
                position=20;
                f=f/2;
                c=c-f;
        }

if ((position==14)||(position==10)||(position==6)||(position==5)||(position==0)){//check the person hit an obstacle
        if (flag==1){
                life--;
                LCD_ClearDisplay();
                LCD_Position(0,5);
                LCD_PrintString("WHOOPS");
                LCD_Position(1,5);
                LCD_PutChar(LCD_CUSTOM_3);
                LCD_Position(1,7);
                LCD_PrintString("x");
                LCD_PrintNumber(life);    //Life-1
                on=1;
        }
}
if ((position==16)||(position==12)||(position==8)||(position==2)){
        if (flag==0){
                life--;
                LCD_ClearDisplay();
                LCD_Position(0,4);
                LCD_PrintString("WHOOPS");
                LCD_Position(1,5);
                LCD_PutChar(LCD_CUSTOM_3);
                LCD_Position(1,7);
                LCD_PrintString("x");
                LCD_PrintNumber(life);
                on=1;
        }
}

if (on==1)//If the person hit the obstacle, play a music
   {
        PWM_Start();
        PWM_WritePeriod(3822);
        PWM_WriteCompare(1912);
        on=0;
   }
else //If the person did't hit the obstacle, nothing happen
   {
        PWM_Stop();
   }

if (life==0)//If the person has no more life, game over,try again, all parameters set to initial
   {
        LCD_ClearDisplay();
        LCD_Position(0,3);
        LCD_PrintString("GAME OVER!");
        PWM_Start();//Play a music
        PWM_WritePeriod(1911);
        PWM_WriteCompare(956);
        PWM_WritePeriod(2551);
        PWM_WriteCompare(1276);
        PWM_WritePeriod(3822);
        PWM_WriteCompare(1912);
        c=1000;
        f=600;
        position=20;
        life=3;
        }

        Refresh_Timer_ReadStatusRegister();
        /* `#END` */
    }
```

CONTROL_ISR.c

```
/* `#START CONTROL_ISR_intc` */
        #include"Control.h"
        extern volatile int flag;
    /* `#END` */
CY_ISR(CONTROL_ISR_Interrupt)
{
        #ifdef CONTROL_ISR_INTERRUPT_INTERRUPT_CALLBACK
                CONTROL_ISR_Interrupt_InterruptCallback();
        #endif /* CONTROL_ISR_INTERRUPT_INTERRUPT_CALLBACK */

        /*    Place your Interrupt code here. */
        /* `#START CONTROL_ISR_Interrupt` */
        if (Control_Read()==1)// Use pins to change the position of person
        {
        flag=0;    //Change the position of person
        }
        else if (Control_Read()==2)
        {
        flag=1;
        }
        Control_ClearInterrupt();
        /* `#END` */
    }
```

Main.c

```
#include "project.h"
volatile int flag;

int main(void)
{
        CyGlobalIntEnable; /* Enable global interrupts. */
        Refresh_Timer_Start();
        CONTROL_ISR_Start();
        REFRESH_ISR_Start();
        LCD_Start();
```

```
    flag=0;
    /* Place your initialization/startup code here (e.g. MyInst_Start()) */

    for(;;)
    {
        //Loop play the background Music
        PWM1_Start();
        PWM1_WritePeriod(1911);
        PWM1_WriteCompare(956);
        CyDelay(1000);
        PWM1_WritePeriod(1703);
        PWM1_WriteCompare(852);
        CyDelay(1000);
        PWM1_WritePeriod(2551);
        PWM1_WriteCompare(1276);
        CyDelay(500);
        PWM1_WritePeriod(1703);
        PWM1_WriteCompare(852);
        CyDelay(1000);
        PWM1_WritePeriod(1517);
        PWM1_WriteCompare(759);
        CyDelay(1000);
        PWM1_WritePeriod(1275);
        PWM1_WriteCompare(638);
        CyDelay(125);
        PWM1_WritePeriod(1432);
        PWM1_WriteCompare(716);
        CyDelay(125);
        PWM1_WritePeriod(1517);
        PWM1_WriteCompare(759);
        CyDelay(250);
        PWM1_WritePeriod(1911);
        PWM1_WriteCompare(956);
        CyDelay(1000);
        PWM1_WritePeriod(1703);
        PWM1_WriteCompare(852);
        CyDelay(1000);
        PWM1_WritePeriod(2551);
        PWM1_WriteCompare(1276);
        CyDelay(1750);
        PWM1_WritePeriod(2551);
        PWM1_WriteCompare(1276);
        CyDelay(125);
        PWM1_WritePeriod(2551);
        PWM1_WriteCompare(1276);
        CyDelay(125);
        PWM1_WritePeriod(2273);
        PWM1_WriteCompare(1137);
        CyDelay(125);
        PWM1_WritePeriod(1911);
        PWM1_WriteCompare(956);
        CyDelay(125);

    }
}

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