Create a Parkour Game

Yuanhao Chen

Department of Robotics and Autonomous Systems, Arizona State University, Tempe, AZ, USA

Abstract—This project demonstrates the creation of a parkour game using the PSoC 5 microcontroller, focusing on the integration of Pulse Width Modulation (PWM), interrupt signals, and timers. The game leverages two independent PWMs to enhance the user experience: one PWM is dedicated to continuously playing background music, while the other handles sound effects such as collisions, game completion, and end-of-game prompts. This approach allows for dynamic audio feedback that aligns with the game's interactive elements, enriching the overall gameplay experience.

keywords—*microcontrollers, lab report, parkour game, PWM, PSoC 5*

Contents

1. Introduction

This project involves developing a parkour game using the PSoC 5 microcontroller, emphasizing custom-designed graphics for the character, obstacles, and life items. The game initiates when a button is pressed, and players use two additional buttons to navigate the character up and down to dodge obstacles. Upon collecting a life item, the player gains an extra life. Initially, the character starts with three lives; encountering an obstacle results in the loss of one life, but the game continues. Successfully completing a level triggers a "Level Up, Speed Up" message, followed by an automatic transition to the next level with increased obstacle speeds. After three successful levels, the game displays a "You Win" message, resets the life count, and returns to the start screen. Conversely, if all lives are depleted, the game displays a "Game Over" message, resets the lives, and returns to the start screen. Throughout the game, continuous background music plays, and additional sound effects are provided for passing levels, game failures, and collisions with obstacles.

2. Prelab Work

2.1. Materials Requirements

The following are all electronic components that need to be used in this project, shown in Table 1.

Table 1. Electronic components list

Component Name	Quantity
PSoC 5 LP Microcontroller	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
Button	2

2.2. Circuit Connection

The connection of the circuit is provided below in Figure 1.

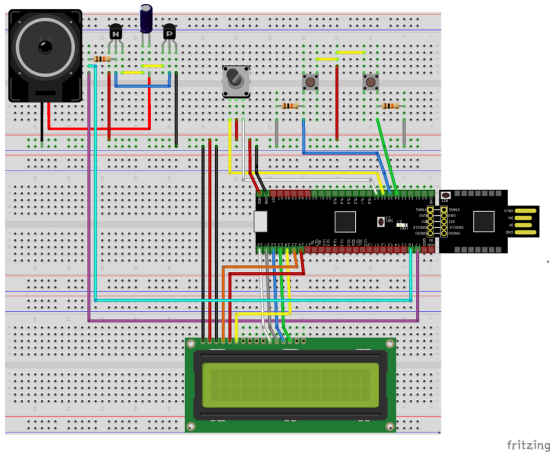

Figure 1. Wire connection of the circuit.

3. Frame Design of Analog Circuit

3.1. Top Design

For this experiment, we need to determine the necessary components to complete the setup. Specifically, two input pins are required to control the character's movements. To ensure the map continuously moves to the left, a timer is needed to run throughout the game, constantly receiving interrupt signals and refreshing the screen. Additionally, we need two relatively independent PWMs to generate sounds: one to loop background music and the other to play sounds for collisions, level completions, and game end prompts. By incorporating these components in the programming software, we can ensure the game functions as intended. The final schematic is shown in Figure 2 below.

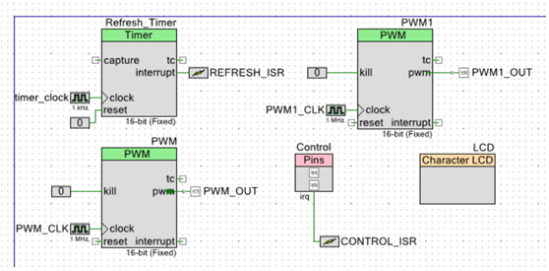

Figure 2. Top design of the circuit.

3.2. Timer Configuration

To manage the map's continuous leftward movement and facilitate regular screen updates, we configured a refresh timer named "REFRESH_ISR". The timer's clock pulse was deliberately set to a lower frequency of 1 kHz to moderate the map's movement speed. **[1]** We connected a logic low level to the reset interface and linked the interrupt signal to the "REFRESH_ISR" port. The timer's resolution was set to 16 bits, and it operates as a fixed function, with "On TC" enabled in the Interrupts settings. See Figure 3 below.

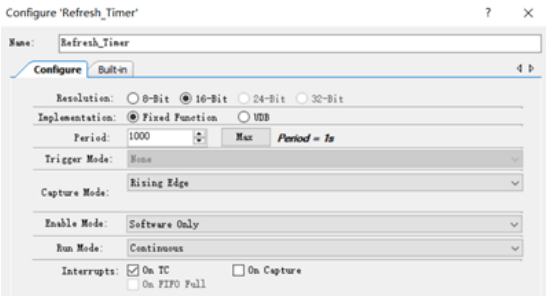

Figure 3. The timer setting.

3.3. PWMs Configuration

To create an immersive audio environment, two PWMs were designed. The first, "PWM", is dedicated to triggering prompt tones during specific game events. The second, "PWM1", plays the background music in a loop. Both PWMs are configured identically except for their naming and the soundtracks they control. Each has a logic 0 connected to its kill port and a 'Digital Output Pin' linked to the interrupt port, named "PWM_OUT" and "PWM1_OUT", respectively. The clock pulses for both PWMs are set to 1 MHz, with configurations fixed to 16 bits shown in Figure 4. **[2]**

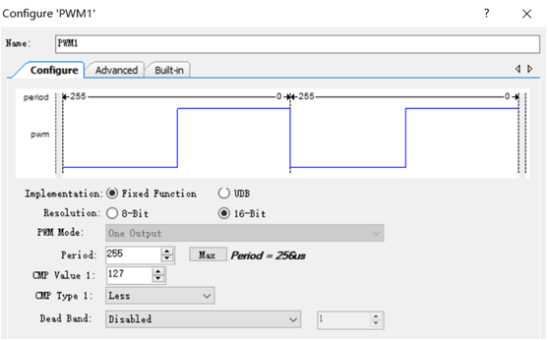

Figure 4. The PWMs setting.

3.4. Digital Inputs Configuration

To facilitate character control in the game, a digital input pin was introduced with two distinct functions: one button controls upward movement, and the other controls downward movement. Figure 5 shows that the hardware connection was disabled, and the driving mode was set to 'Highly Impairment'. The buttons were collectively named 'Control', with an associated interrupt renamed "CONTROL_ISR", connected to the "irq" port.

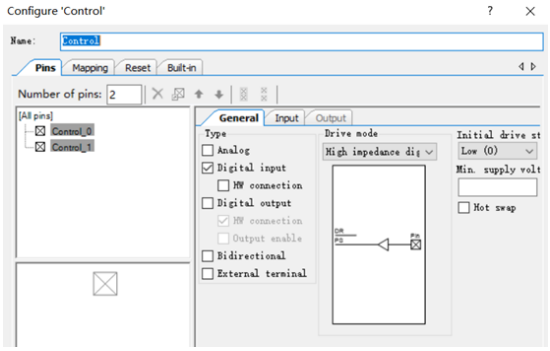

Figure 5. The digital inputs setting.

3.5. LCD and Icons Configuration

In the final phase of the design, an LCD was added to visually represent the game's characters and obstacles. Several icons were meticulously designed for this purpose, ensuring clarity and coherence in the game's graphical representation (See Figure 6). The control buttons were connected to P3[5:4], the LCD to P2[6:0], the PWM1 for background music to P1[6], and the PWM for prompt tones to P1[7]. This comprehensive setup completed the experimental design for the parkour game (See Figure 7).

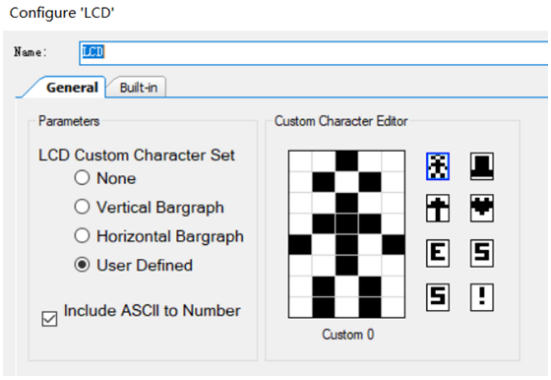

Figure 6. The LCD icons setting.

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	☑

Figure 7. Connecting ports setting.

4. Code Logic

4.1. The Main Function

The main function serves as the foundation of the parkour game, orchestrating the initial setup and continuous operation. In this function, essential components such as the LCD, timer, refresh interrupt signal, and control interrupt signal are activated. A pivotal element in this setup is the definition of a universally accessible variable "flag", initialized to 0, which plays a crucial role in coordinating activities across different interrupt signals. The background music, distinct from other sound effects and not governed by interrupt signals, is looped within the main function to ensure an uninterrupted auditory backdrop throughout the game.

4.2. The REFRESHER Interrupt

The REFRESH_ISR is responsible for dynamically managing the game's visual elements and interactions. This interrupt signal oversees the timer, LCD, and both PWM units, as indicated by the inclusion of "Refresh_Timer.h", "LCD.h", "PWM.h", and "PWM1.h". The initial map configuration starts with a length of 20 frames, and variables "position", "F", "C", "on", and "life" are established to control the game-play dynamics. The map's movement accelerates with each cycle, and obstacles are defined and translated based on these settings. Successful level completion triggers a sequence where the screen is cleared, the PWM plays a customs clearance tone, "Level Up!" is displayed, and game speed increases. Conversely, collisions result in the display of "Whoops!", a decrement in life, and activation of the PWM to signal the event. The game concludes when life reaches zero, displaying "Game Over", resetting all parameters, and playing a concluding tone. Throughout, the interrupt signal ensures the preservation of the original timer state for seamless reactivity. The logic of REFRESH_ISR can be understood through its specific logic diagram shown in Figure 8.

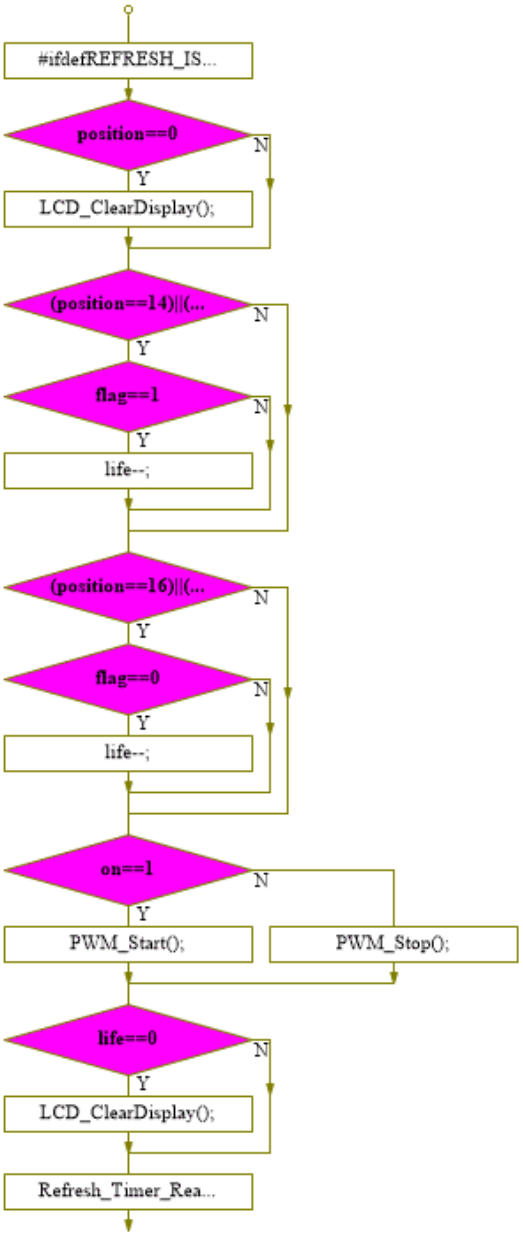

Figure 8. Logic diagram for REFRESH_ISR.

4.3. The CONTROL Interrupt

The CONTROL_ISR simplifies the interaction mechanism, focusing solely on character movement through the control buttons. By modifying the "flag" variable, the control buttons dictate the character's position, facilitating upward or downward movements corresponding to the left and right buttons, respectively. This approach maintains the character's line-based position and ensures that the game's flow remains consistent by clearing the interrupt after each activation. The operational logic of CONTROL_ISR is detailed in its associated logic diagram shwon in Figure 9.

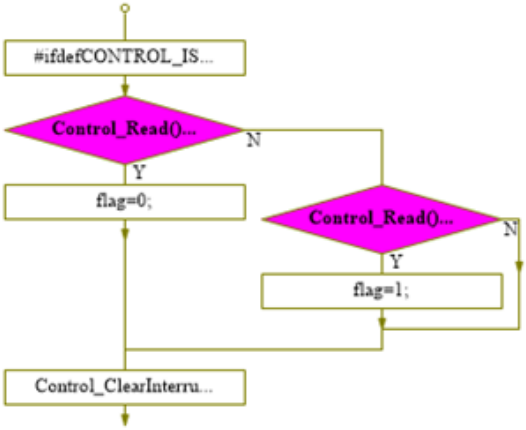

Figure 9. Logic diagram for CONTROL_ISR.

5. Summary

In this experiment, we delved into the multifaceted aspects of a parkour game designed using the PSoC 5 microcontroller, with a particular focus on the integration of speakers, position detection, and control signals. Position detection was critical for managing character navigation and obstacle interactions, while control signals ensured responsive and precise user input.

However, the potential for further enhancement remains. Incorporating random functions and additional in-game items could significantly enrich the game play, adding layers of complexity and unpredictability that would elevate the user experience. Such additions would not only increase the game's depth but also offer new challenges and excitement for programmers, reflecting a more comprehensive and sophisticated implementation of coding principles.

6. Appendix

6.1. Code for Main Function

Caution

The following code should be programmed from the first line of "main.c" file.

```
#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 */
```

Code 1. Complete code in "main.c".

6.2. Code for Refresher Function

> **Caution**
>
> The following code should be programmed from the line right after *"START_REFRESH_ISR_intc"* in "REFRESH_ISR.c" file.

```
\newmdenv[
  /* `#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` */
}

]{note}
```

Code 2. Complete code in "REFRESH_ISR.c".

6.3. Code for Controller Function

> **Caution**
>
> The following code should be programmed from the line right after "*START_CONTROL_ISR_intc*" in "CONTROL_ISR.c" file.

```
\newmdenv[
/* `#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` */

}

]{note}
```

Code 3. Complete code in "CONTROL_ISR.c".

6.4. Final Product Photos Showcase

The following four pictures were taken during the game, all interrupts are clearly shown.

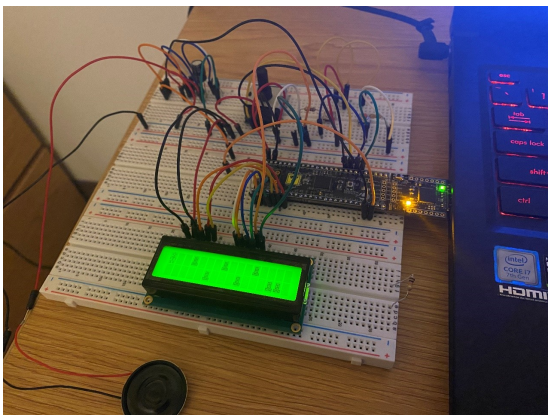

Figure 10. The little hero is crossing the obstacles.

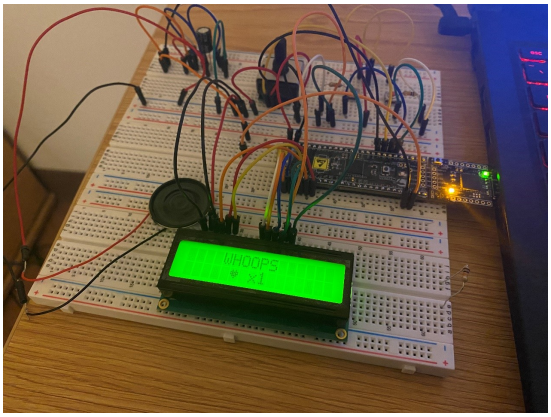

Figure 11. Whoops! Crashed on an obstacle, 1 HP left.

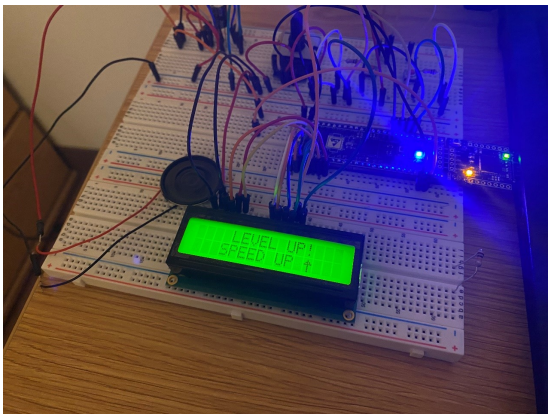

Figure 12. Stage clear, level up and speed up.

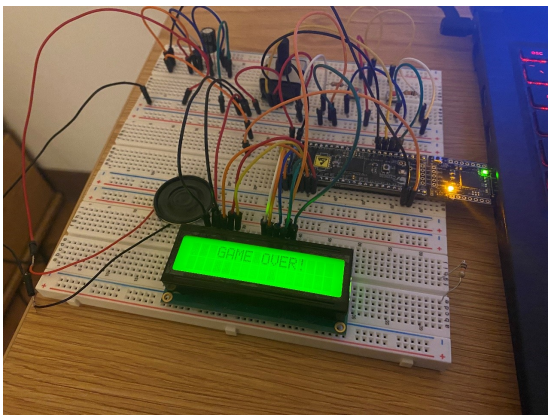

Figure 13. No life left, game over.

6.5. Importance of Hardware Timers

Often in microcontroller based programs, it is necessary to have the microcontroller do some repetitive, periodic task. While in some cases, a simple delay loop can suffice, delays are sub-optimal as then the microcontroller is unable to process other events during the delay, nor is the timing truly guaranteed if there are other interrupts! Use of a hardware timer solves these issues, and hardware timers are found nearly universally on any microcontroller. Most operate by some combination of an input clock, programmable dividers (sometimes called prescalers) for this clock, a counting register, and some kind of period value. The counting register counts up/down on every clock edge, and when it hits 0/period, the timer triggers an interrupt. Most timers also include the ability to capture events by storing the current value of the count to a register when such a capture event occurs. Input capture can be useful for measuring time between external events.

On the PSoC, the Timer module accomplishes this for us. We feed it with a clock whose target frequency we input (chosen by PSoC Creator from the PSoC clock system). On each rising edge of this clock, an internal Count register decrements by one. When this Count register hits 0 (or terminal count), the module can trigger an interrupt. The Count register then reloads with the value stored in the Period register, and the whole process repeats. This process thus produces interrupts at regular intervals. For example, suppose the clock input to the Timer is set to 1 MHz, and the timer has a value of 99 stored in the Period register. Then the interrupt would trigger every (0.000001 s)*(99+1) = 0.0001s or at 10 kHz.

6.6. The Pseudo-Random Sequence (PRS) Module

As we discussed in the summary section, can we make the elements of the game (such as increasing life, obstacle position, etc.) appear randomly? The answer is yes, and here is a short description of PRS module.

Since all processors are deterministic (they can only execute a pre-defined sequence of opcodes), true randomness is actually quite difficult to achieve on any CPU. Most computers skirt the issue by having some pool of entropy maintained by the operating system, whose source is a combination of non-deterministic things it can access (various component temperatures, disk access times/rates/block IDs/etc, mouse movements, keyboard keys, other environmental noise, etc.). The OS uses this pool to "seed" a sequence generator whose output produces what appears to be a sequence of random bytes, but actually they're not.

In the C standard library, there is a ***rand()*** function that is in ***stdlib.h***. It functions similarly to the OS pseudo-random generator, in that it uses a seed number (0 by default, though it can be set by the ***srand()*** function) to generate the sequence, and each successive call of the rand() function produces the next output of the sequence. Unfortunately, unless there is a good source of non-determinism available to seed it, calling the rand() function will produce the exact same sequence of numbers. Such is typically the case in microcontrollers with no OS available to pool non-deterministic sources into a source of entropy. Thus, alternative methods are needed, or some partially non-deterministic way of seeding it (ie. reading a floating ADC pin, measuring the time for a button press, sampling a voltage, etc.) has to be used.

In hardware, a common way to generate a pseudo-random sequence is through a device called a ***Linear-Feedback Shift Register (LFSR)***. A LFSR is a shift register whose next input bit is a function of some combination, typically using XOR gates, of current bits in the register. Which bits are used (they are sometimes called the taps since you are "tapping" that bit's output in the register) and how they are XORed are often expressed as a polynomial. **[3]** See example below (From Wikipedia article on LFSRs):

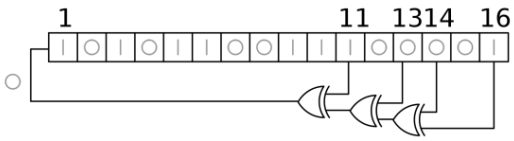

Figure 14. An example of LFSR.

The polynomial for this example is:

$$x^{16} + x^{14} + x^{13} + x^{11} + 1$$

Since the 16th, 14th, 13th, 11th, and bits are tapped to produce the next input, and 1 to indicate the input is the 1st bit. The current state of the generator is 0xACE1. On the next clock cycle, the value, in binary, would be 0101011001110000 or, in hex 0x5670. Therefore, reading the register at different points in time will produce, what appears to be, the "random" sequence of values. Seeding an LFSR is merely setting the value of the initial state. The input bit to position 1 can also be seen as a "random bitstream when looking on the outside. Just note, though, that if you know the current state and polynomial, the output sequence is entirely deterministic, meaning I could tell you which number comes next (or in 10 or 17 or 100 iterations, it doesn't matter), hence the name pseudo-random as it only appears random if all you see is the output and the rest is concealed. There are other downsides, too. Eventually, based on the size of the register, the sequence will repeat, which is called the period of the generator. The bigger the register in bits, more generally, the longer the output "random" sequence before this occurs, to a maximum of 2^{N-1} values, where N is the register size in bits. The period is also based on the polynomial, and not all polynomials are equal. Some will repeat the sequence before all 2^{N-1} values have been output. Thus, there are some predefined polynomials that are commonly used to ensure the max. period is reached.

Now, on the PSoC, there is a module that can generate a PRS in hardware. It takes as input a clock, and as output produces the bitstream. If you go to configure it, you can punch in the resolution, in bits, of the sequence, and it will produce the optimal polynomial for that size by default. It also has an API that allows you to $\_Read()$ the current value (or state) of the generator, to $\_WriteSeed()$ if you want to reseed it, or even to $\_WritePolynomial()$ if you want to change it from the pre-generated one. Pretty nifty, though you would still have to make sure that you don't call the $\_Read()$ function at the exact same point in time in your code after calling $\_Start()$! If you need to generate numbers within a specific range, you use modular arithmetic! For example, say you were trying to do a 6 sided die. You could do something like: **uint8 die = (PRS_Read() % 6)+ 1** to emulate the throw. The %6 returns a number between 0 and 5, and the +1 makes it between 1 and 6 like a die. Weighted random numbers can be done with thresholds/mods or both. For example if I want an event to only have a 1/100 chance, I could %100 and check if it's equal to 0 (or really any other target number, they are all equally likely with a rate of 1/100!)

References

[1] Infineon Technologies CY8CKIT-059 PSoC 5LP Prototyping Kit User Guide. https://www.mouser.com/pdfDocs/CY8CKIT_059_UG.pdf

[2] Cypress Programmable System-on-Chip (PSoC) 5LP: CY8C58LP Family Datasheet. https://cdn.sparkfun.com/assets/e/6/1/8/4/PSoC_5LP_CY8C58LP_Family_DS.pdf

[3] Cypress Pseudo Random Sequence (PRS). https://www.infineon.com/dgdl/Infineon-Component_Pseudo_Random_Sequence_(PRS)_V2.20-Software%20Module%20Datasheets-v02_04-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e8cf9e01b76