Laboratory 03 – Simon Game

By: **Caleb Probst**

Spring 2023 ECE 381 Microcontrollers

Introduction

In this laboratory, the main concept was PWM signals. To show this concept, we recreated the game Simon from the 70's. This game is a memory game that blinks a sequence of lights, and if you follow the sequence correctly, it will add another to the sequence. This shows the concept of PWM's in two ways. First, we have to be able to change the brightness of the LEDs and be able to play sound through a speaker. Additionally, we also learned about pseudo-random numbers, and we implemented a hardware implementation of this through the PRS (Pseudo Random Sequence) hardware.

Goal

The goal of this laboratory is to recreate the Simon memory game. To do this, we first need to be able to play sound through the speaker at the correct frequencies, use LEDs to signal which color is which, and have the correct game logic. First, the program waits on the user input to start the game. The LEDs then pulse 3 times with the speaker, and then the random sequence begins. After each sequence, there is a 3 second countdown timer per button click, that if it goes off, the game ends. If the user is to incorrectly press a button in the sequence, the game also ends. For each round, the LEDs get dimmer and the sequence gets longer. This goes up to 20 rounds. If all 20 rounds are completed, a win sequence is played. If the game is lost, a lose sequence plays. After this, the game waits for user input and restarts.

Materials

PsoC4 4200M Microcontroller	1
Red LED	1
Blue LED	1
Yellow LED	1
Green LED	1
Push Button	4
1KΩ Resistor	1
470Ω Resistor	5
10μF Capacitor	1
P2N2222 Transistor	1

Procedure and Results

Part 1 – Setup in PSoC Creator

For the Top Design in this laboratory, we need 5 different modules. First, we need the PRS module to be able to generate our random numbers using hardware. Second, we need 2 different PWM signals (which are hardware controlled). One is for the speaker and one is for the LEDs. However, in order to make all 4 LEDs blink individually or together, we need a de-multiplexer (DeMUX) to split the signal into 4 different pins. To control the signal to the LEDs, we need a control register module as well. Finally, we need an input timer to keep track of the timeout for the user input and fail sequence. With the input timer, we need a 1KHz clock signal to keep track of time, and for the LED PWM, Speaker PWM and PRS, we need a larger clock signal, so we use the default 1 MHz signal. Finally, we add the correct output pins for the LEDs and speaker, as well as an interrupt for the input timer. The LED outputs and speaker output are both in strong drive mode, and the button inputs are in Resistive Pull Down (this means we do not need an external resistor). Additionally, on the PRS, we need the enable to be on, so we simply force it to VCC using a constant "1". *Figure 1* shows the full Top Design. *Table 1* also shows the pins that we connected the modules to.

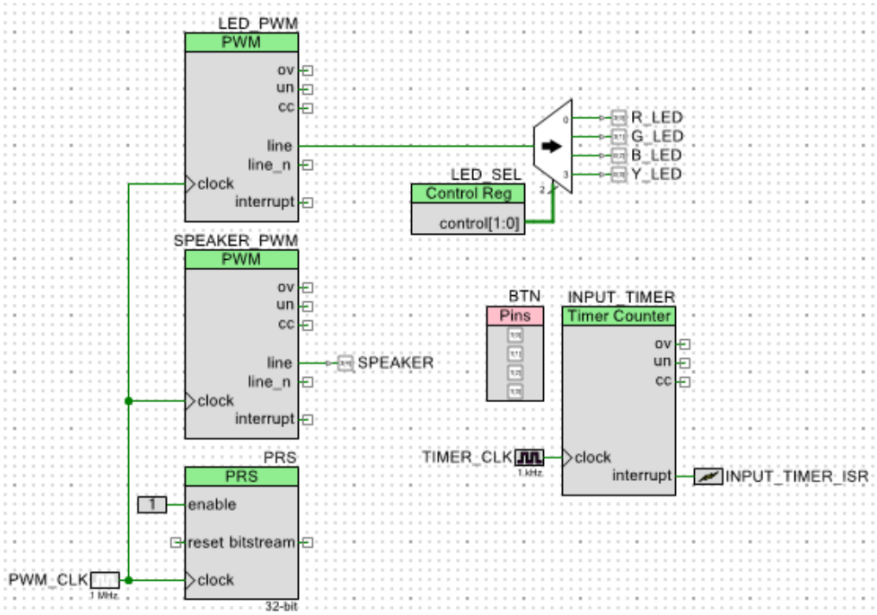


Figure 1: Top Design Diagram

BTN[3:0]	P1[3:0]
B_LED	P0[2]
G_LED	P3[1]
R_LED	P3[0]
Y_LED	P0[3]
SPEAKER	P3[4]

Table 1: Pin Outs of Hardware on PSoC4

Part 2 – Hardware and Wiring

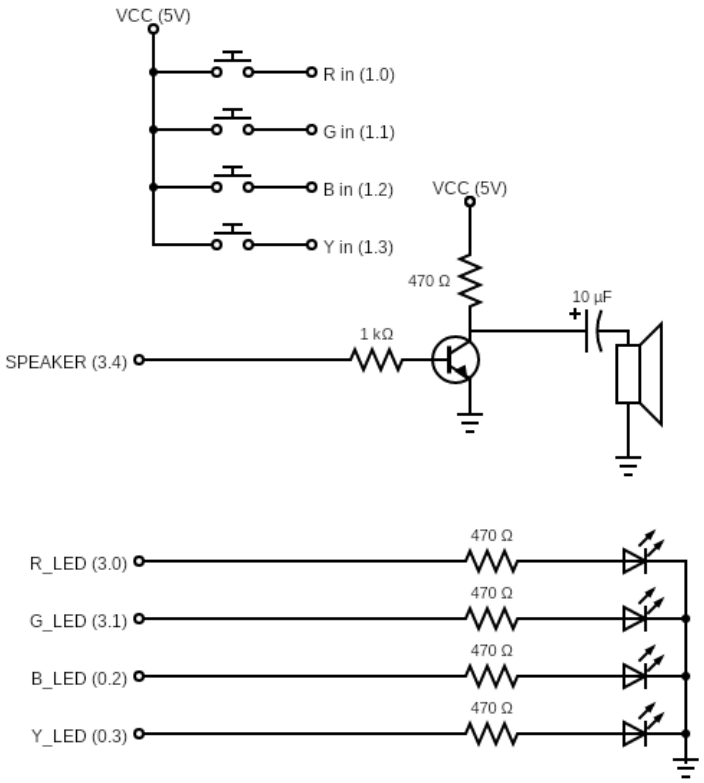


Figure 2: Wiring Diagram

Above in *Figure 2* is the wiring diagram corresponding to the configured Top Design. Some changes that can be made is if you do not use Resistive Pull Down on the input pins, you may use external resistors to limit the current going into the GPIO pin. For the transistor, we used a P2N2222 transistor. The data sheet for this is given in the Appendix under *P2N2222*. Following

this data sheet, we must know that looking at the flat side of the transistor, the left most pin will go to 470Ω resistor, capacitor and speaker, the middle pin will go to the speaker output and 1KΩ resistor, and the right pin will go to ground.

Part 3 – Software

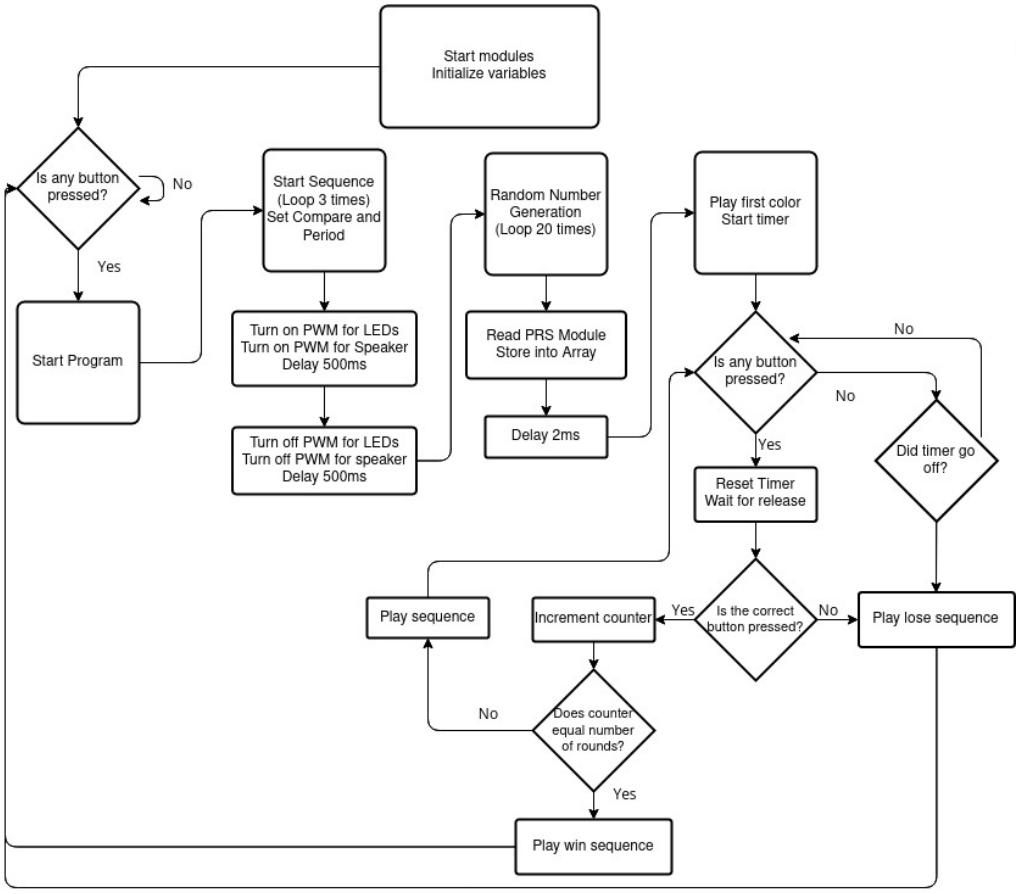


Figure 3: Flowchart of Program

In this program, the majority of the game logic was written for us. Since this program was focusing on PWM signals, that is the majority of what we needed to write. *Figure 3* shows the entire flow diagram of the program as well.

poll_button() Function

The *poll_button* function is a very simple function. It basically is used inside a polling loop for each time it waits for the user to press a button. All it does is check if the button is pressed, and if it is a value we recognize (0, 1, 2, 4, and 8 in this case) and returning the number of the button that is pressed.

play_color() Function

The *play_color()* function takes 3 arguments, the color, the DC value (0-100%) and a delay of how long in between each color turning on and off. First, it stores a dummy variable for the

period of the PWM for the speaker. Then it selects the color that we defined from the arguments and selects the correct period for the frequency we desire for that color (Red =3225 [310Hz], Green = 3174 [415Hz], Blue = 4784 [209Hz], Yellow = 3968 [252Hz]). We then write the PERIOD value as the period defined, and the COMPARE value as half of the period, so that we have a 50% duty cycle. For the majority of the math involved, the value was rounded up, so the addition of one for the frequency is not very necessary, but can be done if the frequency is not close enough to what you want it to be. The LED PWM PERIOD value is also set to 1000, and the COMPARE value is set to the *dc* argument times 10 so that we can have 0 to 1000. This gives us a dimming function that for each sequence it gets dimmer. It then starts the PWM for both the LEDs and selects the color that we defined, with a delay from the argument, and then stops with another delay after to have a pre-defined space in between each tone.

play_start() Function

This function is very similar to the *play_color* function as it simply sets the period of the PWMs to their respective values and the compare to the values we want, and then turns all LED's on through a loop and then turns them all off. This happens while the speaker is playing a 600Hz tone. This repeats 3 times.

play_fail() Function

This function is similar to the *play_start* function except without the looping of 3 times and at a different frequency on the speaker PWM.

play_win() Function

This function is similar to the *play_start* function except with different frequencies on the speaker PWM and repeats differently. All the concepts are the same.

main Function

This function controls the majority of the game logic, which was written for us. First, it waits for a button to be pressed which will then begin the game, using the *play_start* function. Next, we generate a sequence of random values 0 through 3 that gets stored in the array. We use the PRS module for this with a small delay in order to account for the delay needed for the shift register to shift bits. We then set some initial variables and start the loop. In the loop, it first plays the sequence at the number of steps that you have gotten so far, and then starts the timeout timer and waits for button press. If the button is not pressed in 3 seconds, it will play the fail. If the button is pressed, and it is incorrect, it will play the fail. If the button is pressed and is correct, the timer is reset and continues to the next press until the full sequence is correct. After that, the counter is incremented and then loops until the number of rounds is 20 and is correctly guessed. If the full sequence is guessed after 20, the *play_win* sequence is played. After each win or fail, the game restarts and waits for user input to begin.

Testing Methodology

For the testing of this laboratory, we simply play the game. *Figure 4* shows the device before any input is pressed.

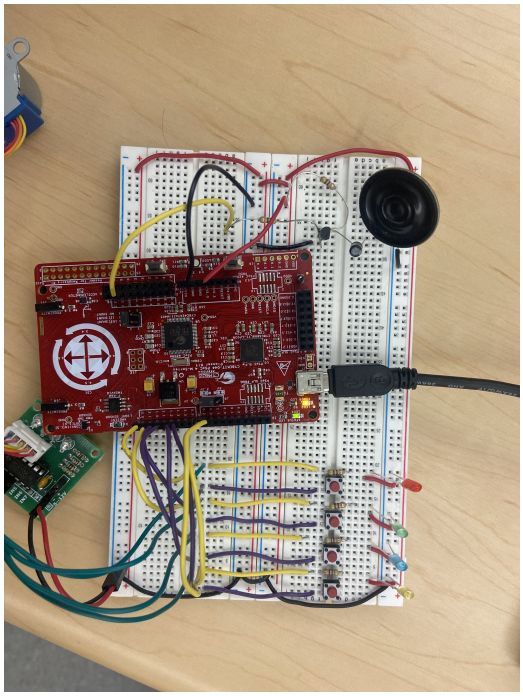


Figure 4: Before User Interaction

As shown no LEDs light up, which means it is waiting for input. After a button is pressed, then the start sequence will show as shown by *Figure 5*.

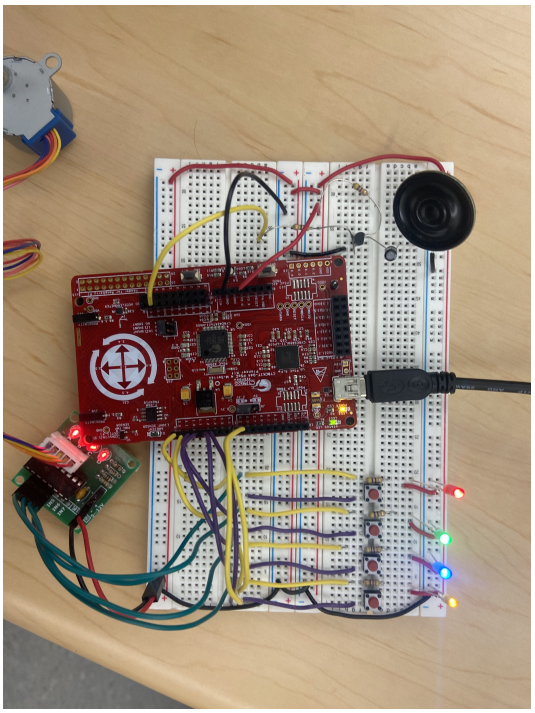


Figure 5: Start Sequence

As shown, all LEDs light up and the speaker sounds. This repeats 3 times. Next, it will play the first one in the sequence and wait for button press as shown by *Figure 6*.

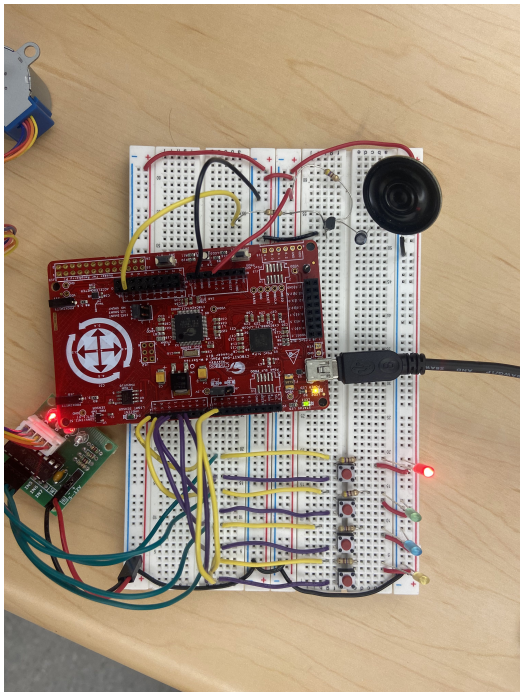


Figure 6: Playing Sequence

This should be repeated and add one to the sequence every time the sequence is guessed correctly. If the button is pressed incorrectly, a low frequency noise should play and the lights should go off and the game should repeat.

Conclusion

This lab is much more hands on and also include most of the previous laboratories. Overall, the laboratory made the concept of PWMs much simpler. Being able to see both the change of frequency and the change in average voltage makes the concepts much clearer. This lab clears up the misconception of analog outputs in digital devices as well.

Appendix

P2N2222

P2N2222A

Amplifier Transistors

NPN Silicon

Features

- These are Pb−Free Devices*

MAXIMUM RATINGS (T_A = 25°C unless otherwise noted)

Characteristic	Symbol	Value	Unit
Collector − Emitter Voltage	V_{CEO}	40	Vdc
Collector − Base Voltage	V_{CBO}	75	Vdc
Emitter − Base Voltage	V_{EBO}	6.0	Vdc
Collector Current − Continuous	I_C	600	mAdc
Total Device Dissipation @ T_A = 25°C Derate above 25°C	P_D	625 5.0	mW mW/°C
Total Device Dissipation @ T_C = 25°C Derate above 25°C	P_D	1.5 12	W mW/°C
Operating and Storage Junction Temperature Range	T_J, T_{stg}	−55 to +150	°C

THERMAL CHARACTERISTICS

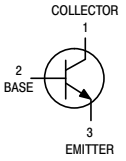
Characteristic	Symbol	Max	Unit
Thermal Resistance, Junction to Ambient	$R_{\theta JA}$	200	°C/W
Thermal Resistance, Junction to Case	$R_{\theta JC}$	83.3	°C/W

Stresses exceeding Maximum Ratings may damage the device. Maximum Ratings are stress ratings only. Functional operation above the Recommended Operating Conditions is not implied. Extended exposure to stresses above the Recommended Operating Conditions may affect device reliability.

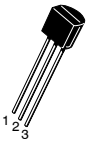
*For additional information on our Pb−Free strategy and soldering details, please download the ON Semiconductor Soldering and Mounting Techniques Reference Manual, SOLDERRM/D.

ON Semiconductor®

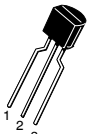
http://onsemi.com



TO−92
CASE 29
STYLE 17

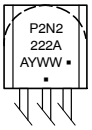


STRAIGHT LEAD
BULK PACK



BENT LEAD
TAPE & REEL
AMMO PACK

MARKING DIAGRAM



A = Assembly Location
Y = Year
WW = Work Week
▪ = Pb−Free Package
(Note: Microdot may be in either location)

ORDERING INFORMATION

Device	Package	Shipping†
P2N2222AG	TO−92 (Pb−Free)	5000 Units/Bulk
P2N2222ARL1G	TO−92 (Pb−Free)	2000/Tape & Ammo

†For information on tape and reel specifications, including part orientation and tape sizes, please refer to our Tape and Reel Packaging Specification Brochure, BRD8011/D.

Main Code

```
#include "project.h"

static const int ROUNDS = 20;
volatile int timeout, pressed;

// This function implements a non-blocking polling loop.
// If neither of the buttons is pressed, it should return 0,
// otherwise it should return a value that corresponds
// to one of the colored buttons. To be consistent, this
// should match the color used in the sequence
int poll_button()
{
    int button = 0;

    // XXX: Polling loop for button goes here

    // For each iteration of polling loop, check value of BTN. If bit
change (0,1,2,4,8)
    // are on, change the button value based on this value
    switch(BTN_Read()){
    case 1:
        button = 1;
        break;
    case 2:
        button = 2;
        break;
    case 4:
        button = 3;
        break;
    case 8:
        button = 4;
        break;
    default:
        button = 0;
        break;
    }

    return button;
}

// The play color function illuminates an led
// using a PWM at a duty cycle given by dc for
// a number of milliseconds equal to delay and
// plays the correpsonding note.
//
//  INPUTS:
//      color (int): A mapping to determine the
//                    LED to play.
//                  Suggested Colors and Notes are
//                  0: Red (310 Hz)
//                  1: Green (415 Hz)
//                  2: Blue (209 Hz)
//                  3: Yellow (252 Hz)
```

```
//
//      dc (int): The duty cycle of the chosen LED's
//                PWM. Only numbers 0-100 should be
//                valid.
//
//      delay (uint32): How long to play the LED and
//                      note, in milliseconds

void play_color(int color, int dc, uint32 delay)
{
    // Set period to initial value of 0
    int period = 0;

    switch(color){
    case 0:
        // Period = 3225 for Red
        // 1000000 / 310 = 3225
        period = 3225;
        break;
    case 1:
        // Period = 3174 for Green
        // 1000000 / 415 = 3174
        period = 3174;
        break;
    case 2:
        // Period = 4784 for Blue
        // 1000000 / 209 = 4784
        period = 4784;
        break;
    case 3:
        // Period = 3968 for Yellow
        // 1000000 / 252 = 3968
        period = 3968;
        break;
    }
    // Write speaker frequency from switch statement
    SPEAKER_PWM_WritePeriod(period);
    // 50% duty cycle, period/2
    SPEAKER_PWM_WriteCompare((int) (period/2));
    // Write LED period to 1000 for easy math
    LED_PWM_WritePeriod(1000);
    // Compare is duty cycle% * 10
    LED_PWM_WriteCompare(dc * 10);
    // Start PWM's'
    LED_PWM_Start();
    SPEAKER_PWM_Start();
    // Select color
    LED_SEL_Write(color);
    // Delay the amount desired
    CyDelay(delay);
    // Stop PWM's'
    SPEAKER_PWM_Stop();
    LED_PWM_Stop();
    // Small delay
    CyDelay(100);
```

```
}

// This should play the start sequence of Simon, which is
// to turn all 4 LEDs on and play a 600Hz tone for 500 ms,
// then turn them off for 500ms, then repeat 2 more times
void play_start()
{

    // 50% duty cycle, 600 Hz

    /*
    How to calculate Hz
    Take clock rate (1MHz) and divide by frequency. (in microsec) -
WritePeriod
    Duty cycle : multiply %duty cycle to the period - WriteCompare
    Period = (1000000/600) + 1
    Compare = .5 * Period



    */
    // Write period of 1667 for frequency of 600Hz (math above)
    SPEAKER_PWM_WritePeriod(1667);
    // 50% duty cycle
    SPEAKER_PWM_WriteCompare(833);
    // Basically DC voltage
    LED_PWM_WritePeriod(1000);
    LED_PWM_WriteCompare(999);
    for (int i = 0; i < 3; i++){
        // Half a second delay
        CyDelay(500);
        // Start PWM modules
        SPEAKER_PWM_Start();
        LED_PWM_Start();
        // 1 ms * 4 * 125 = 500 ms (time that LEDs are on)
        for(int i = 0; i < 125; i++){
            // Switch between LEDs and delay
            LED_SEL_Write(0);
            CyDelay(1);
            LED_SEL_Write(1);
            CyDelay(1);
            LED_SEL_Write(2);
            CyDelay(1);
            LED_SEL_Write(3);
            CyDelay(1);
        }
        // Stop PWM modules
        SPEAKER_PWM_Stop();
        LED_PWM_Stop();
    }
    // Not needed, but stopping Speaker PWM module again.
    SPEAKER_PWM_Stop();
}
```

```
// This should play the fail sequence, which is to turn on
// all 4 LEDs and play a 42Hz tone for 1.5 seconds
void play_fail()
{
    // Set speaker frequency to 42 Hz
    // 1000000 / 42 = 23810
    SPEAKER_PWM_WritePeriod(23810);
    // 50% duty cycle
    SPEAKER_PWM_WriteCompare(11904);
    // Basically DC voltage
    LED_PWM_WritePeriod(1000);
    LED_PWM_WriteCompare(999);
    // Non needed for loop, only runs once, but is copied from
play_start
    for (int i = 0; i < 1; i++){
        // Start PWM modules
        SPEAKER_PWM_Start();
        LED_PWM_Start();
        // Play LEDs for 1.5 seconds
        // 1ms * 4 * 375 = 1500 ms
        for(int i = 0; i < 375; i++){
            LED_SEL_Write(0);
            CyDelay(1);
            LED_SEL_Write(1);
            CyDelay(1);
            LED_SEL_Write(2);
            CyDelay(1);
            LED_SEL_Write(3);
            CyDelay(1);
        }
        // Stop PWM modules'
        SPEAKER_PWM_Stop();
        LED_PWM_Stop();
    }
    // Not needed, copied from the last.
    SPEAKER_PWM_Stop();
}

// This should play back the victory sequence. In the original
// Simon, that plays:
// R, Y, B, G in sequence 0.1s apart, 3 times
// then RY,
// then play failure tone for 0.8s, while the lights then
// continually flash R Y B G 0.1s apart.
//
// You can implement whatever victory sequence you want if
// you don't feel like doing the original one.
void play_win()
{
    // Basically DC voltage
    LED_PWM_WritePeriod(1000);
    LED_PWM_WriteCompare(999);
    // Repeat this sequence 3 times
    for(int i = 0; i < 3; i++){
        // Repeat this inner sequence 4 times (per color)
```

```
        for(int j = 0; j < 4; j++){
            // Write the correct LED
            LED_SEL_Write(j);
            // Blink LED and play speaker at 600Hz with 100ms delay
between blinks
            for(int k = 0; k < 3; k++){
                SPEAKER_PWM_WritePeriod(1667);
                SPEAKER_PWM_WriteCompare(833);
                LED_PWM_Start();
                SPEAKER_PWM_Start();
                CyDelay(100);
                LED_PWM_Stop();
                SPEAKER_PWM_Stop();
                CyDelay(100);
            }
            // Blink LED and play speaker at 540Hz with 100ms delay
            SPEAKER_PWM_WritePeriod(1852);
            SPEAKER_PWM_WriteCompare(926);
            SPEAKER_PWM_Start();
            LED_PWM_Start();
            CyDelay(100);
            SPEAKER_PWM_Stop();
            LED_PWM_Stop();
            CyDelay(100);
        }
    }
}

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

    /* Variables */
    int ix = 0, num_correct = 0, errors = 0, seq_ctr = 0;
    int simon_sequence[ROUNDS];


    /* Place your initialization/startup code here (e.g.
MyInst_Start()) */
    PRS_Start();
    SPEAKER_PWM_Start();
    SPEAKER_PWM_Stop();
    LED_PWM_Start();
    LED_PWM_Stop();
    INPUT_TIMER_Start();
    INPUT_TIMER_Stop();
    INPUT_TIMER_ISR_Start();

    for(;;)
    {
        /* Place your application code here. */

        // Wait for any button press
        while (poll_button() == 0);
```

```
        // Play the start sequence
        play_start();

        // Generate random sequnce of 20 rounds. Store it
        // in simon_sequence

        // Use the PRS module to generate random numbers.
        // % 4 will create numbers 0 - 3
        for(int i = 0; i < ROUNDS; i++){
            simon_sequence[i] = PRS_Read() % 4;
            // Delay for PRS module
            CyDelay(1);
        }



        // Zero the sequence counter. We use this variable to keep
        // track of how far along we are in the game
        seq_ctr = 1;

        // These variables keep track of how many are correct each
round and
        // if any errors are input.
        num_correct = 0, errors = 0;

        // Start the counter at 1 (Why not zero? Due to the nature of
        // for loops bounds checking! See below)
        seq_ctr = 1;


        // The main game begins here. As long as there are no errors,
        // and we haven't gone through all rounds, we are stuck in
here
        while (seq_ctr < ROUNDS + 1 && errors == 0)
        {

            // For each round of the game, we start with having no
            // correct buttons pressed
            num_correct = 0;

            // Playback current sequence. The LED should get
            // progressively dimmer for each color in the sequence
            for (ix = 0; ix < seq_ctr; ix++)
            {
                play_color(simon_sequence[ix], (100/ROUNDS)*(ROUNDS -
seq_ctr + 1), 420);
            }

            // Since seq_ctr keeps track of the current number we have
            // gotten correct so far, for each round we have to keep
looping
            // until we get all correct (or break if an error)
            while (num_correct < seq_ctr)
            {
```

```
                // Pressed keeps track of the current (if any) button
that
                // was pressed. Timeout starts at 0 and gets set from
                // INPUT_TIMER_ISR if a button hasn't been pressed in
                // 3s
                pressed = 0;
                timeout = 0;


                // This is the main input loop. When poll_button
returns
                // something non-zero, that means a button is pressed.
                // If timeout gets set to 1, that means 3s have
elapsed
                // without a button press

                // Before button poll *initialize* the Timer (restarts
the timer)
                INPUT_TIMER_Init();
                INPUT_TIMER_Start();
                // Start ISR for timeout
                INPUT_TIMER_ISR_Start();
                // 3 second timer
                INPUT_TIMER_WritePeriod(3000);
                // Poll button
                while(pressed == 0 && timeout == 0)
                {
                    pressed = poll_button();
                }
                // Wait for button release
                while(BTN_Read()≠0 && timeout==0);
                // Stop timer and ISR
                INPUT_TIMER_Stop();
                // Not needed, but here anyway
                INPUT_TIMER_WritePeriod(3000);
                INPUT_TIMER_ISR_Stop();


                // If the main input loop breaks due a timeout, then
                if (timeout == 1)
                {
                    // Play the fail sequence

                    play_fail();

                    // Set the error counter
                    errors = 1;

                    // Break out of the loop for the current round.
                    // This combined with setting errors should get
                    // back to start of the gam
                    break;
                }
                // otherwise, the main input loop broke due to a
button press.
```

```
                else
                {
                    // Whether right or wrong, playback the correct
color
                    play_color(pressed - 1, (100/ROUNDS)*(ROUNDS -
seq_ctr + 1), 420);

                    // Check here to see pressed button is correct
                    if (pressed == simon_sequence[num_correct] + 1)
                    {
                        // Increment the number of correctly input
                        // buttons for the sequence
                        num_correct++;
                    }
                    // Otherwise, the button pressed didn't match the
current sequence color
                    else
                    {
                        // Play the fail sequence
                        play_fail();

                        // Set the error counter
                        errors = 1;

                        // Break out of the loop for the current
round.
                        // This combined with setting errors should
get
                        // back to start of the game
                        break;
                    }
                }

            }

            // If we get here, all the buttons entered for the current
round
            // matched the sequence! We can increment the sequence
counter
            // and go on to the next round. The original Simon delayed
0.8s
            // between rounds.
            if (errors == 0)
            {
                seq_ctr++;
                CyDelay(800);
            }
        }

        // If we get here, we have completed the game! Play a
celebration!
        if (errors == 0)
        {
            play_win();
        }
```

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

INPUT_TIMER_ISR.c

```
/* `#START INPUT_TIMER_ISR_intc` */
#include "INPUT_TIMER.h"
extern volatile int timeout;
/* `#END` */
...
...
...
CY_ISR(INPUT_TIMER_ISR_Interrupt)
{
    #ifdef INPUT_TIMER_ISR_INTERRUPT_INTERRUPT_CALLBACK
        INPUT_TIMER_ISR_Interrupt_InterruptCallback();
    #endif /* INPUT_TIMER_ISR_INTERRUPT_INTERRUPT_CALLBACK */

    /*  Place your Interrupt code here. */
    /* `#START INPUT_TIMER_ISR_Interrupt` */
INPUT_TIMER_ClearInterrupt(INPUT_TIMER_TC_INTERRUPT_MASK);
timeout = 1;


    /* `#END` */
}
```