ENS2257/6155: Microprocessor Systems

Project – Frogger Implementation

Atmega328 Interfacing and Programming

GROUP MEMBERS

Blake Schoppe – 10474767

Tshering Wangchuk- 10616192

Table of Contents

Table of Figures 3

Introduction 4

Task 1: Sound Generation 5

Sound Production: 5

Pseudocode 7

Assumptions and Special Features: 8

Task 2: Indefinite Frogger Game 9

Frogger Game Mechanics and Functionality: 9

Principles of Operation: 10

Algorithms and Pseudo-code: 11

Assumptions and Special Features: 13

Task 3: Competitive Frogger Game 14

Scoring Mechanics and Differences from Basic Frogger: 14

Algorithms and Pseudo-code: 14

Assumptions and Special Features: 16

Hardware Description 17

Core Components: 17

Audio Circuit Components: 17

Calculations for Circuit 18

Conclusion 20

References 21

Appendix 22

Appendix 2 – C Code 25

Table of Figures

Figure 1: 10K Ohm Potentiometer 22
Figure 2: The mentioned speaker 22
Figure 3: Keypad in use 22
Figure 4: Bypass Capacitor on OpAmp 23
Figure 5: Gain increase Capacitor 23
Figure 6: Provided Amplifier 23
Figure 7: Circuit Diagram for Speaker Configuration 24

Introduction

The evolution of gaming technology has been swift and remarkable, with its applications spanning across platforms, from colossal gaming consoles to tiny microcontroller chips. One such endeavor has been undertaken using the ATmega328PB microcontroller, bringing gaming to an unexpectedly minimalistic yet captivating platform. This report presents the intricate workings and design considerations of a two-pronged project: a sound generation module for a game and a recreation of the classic arcade game, "Frogger," in two different versions.

The first task focuses on the utilization of the ATmega328PB's digital pins to produce varied tones, manipulating the frequencies of square waves. This sound generation mechanism brings to life not just audio feedback but adds an essential immersive layer to the gaming experience. The underlying principle involves toggling the digital pins, especially PB0, at specific frequencies to produce recognizable sounds. Moreover, user interaction is facilitated via a keypad, which serves as an input device, further supplemented by auditory feedback through sound effects.

The subsequent tasks plunge into the world of 'Frogger', a timeless arcade game that requires players to safely navigate a frog across numerous hazards to its sanctuary. The first version provides an infinite gameplay experience, encapsulating the fundamental mechanics of the original game. In contrast, the second version offers a competitive edge, incorporating a dynamic scoring system that adds complexity and excitement.

While the project's essence captures the allure of retro gaming, the technicalities underlying it are advanced and meticulous. This report dives deep into the design, algorithms, pseudo-codes, and hardware components that come together to manifest this nostalgic gaming venture on a compact microcontroller platform.

Task 1: Sound Generation

Sound Production:

The game utilizes sound effects by manipulating the frequency of square waves produced by a digital pin, specifically PB0. By toggling this pin at specific frequencies, it generates tones that the user perceives as distinct sounds. The actual tone produced is determined by the duration the pin remains in its HIGH or LOW state, which is influenced by the desired note's frequency. Essentially, a higher frequency corresponds to a shorter delay and vice versa. The primary function that orchestrates this sound generation is play_tone, which requires the intended frequency as a parameter. In the code, PORTB |= (1 << PB0); sets the PB0 pin to a high state, while PORTB &= ~(1 << PB0); sets it to a low state. The variable delayTime plays a crucial role in defining the duration the pin stays in either state, thus dictating the tone's frequency. It's worth noting that, although AVR provides an in-built function, _delay_us(), for introducing delays, it restricts the use of variables since its argument must be predefined at compile-time. As a workaround, a custom delay_ms function was crafted to offer more flexibility in controlling delays.

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void play_tone(uint16_t frequency) {

	// Calculate delay time in microseconds

	uint16_t delayTime = 500000 / frequency / 2;


	// Play the tone for a duration,

	for (uint32_t i = 0; i < (frequency / 2)/4; ++i) {

		// Toggle PB0 high

		PORTB |= (1 << PB0);

		delay_us(delayTime);

		// Toggle PB0 low
```

```
        PORTB &= ~(1 << PB0);

        delay_us(delayTime);

    }

}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Keypad for Input:

The game features a keypad with a layout of 4 rows by 3 columns, designed for user input. For detection purposes, the columns are configured as outputs, while the rows act as inputs, further enhanced with pull-up resistors. To identify a specific keypress, the microcontroller systematically sets each column to a LOW state and then assesses the row values. If any row reads as LOW during this process, it indicates that a key corresponding to the active column has been activated. Two essential functions manage this interaction: the keypressed function verifies the presence of any keypress, while the keypad_read function discerns the exact key that has been pressed. In addition to this, a speaker or buzzer is integrated with the PB0 pin of the microcontroller to generate sound effects. When this pin is toggled at frequencies within human hearing capacity, the connected speaker emanates a sound corresponding to that specific frequency.

Pseudocode

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Function play_tones:

    Display tone grid on the LCD

    loop continuously:

        Capture user input from the keypad

        if input equals '#':

            Display the main menu on LCD

            Exit the function

        else:

            Based on the input value (from 1 to 8):

                Call the `hold_tone` function with the corresponding frequency

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Function hold_tone(frequency):

    Compute the delay duration using the provided frequency

    while a key remains pressed:

        Activate PB0 (set to HIGH)

        Pause for the calculated delay duration

        Deactivate PB0 (set to LOW)

        Pause for the calculated delay duration

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Function play_tone(frequency):

    Compute the delay duration using the provided frequency
```

Repeat for a set duration based on the frequency:

Activate PB0 (set to HIGH)

Pause for the calculated delay duration

Deactivate PB0 (set to LOW)

Pause for the calculated delay duration

//

Assumptions and Special Features:

The system incorporates a buzzer or speaker, which is directly linked to the PB0 pin of the microcontroller. This microcontroller, specifically an ATmega328PB, runs at a clock frequency of 16 MHz. For user input, a 3x4 matrix layout characterizes the keypad design. A unique feature of this setup is its ability to produce tones consistently as long as a related keypad button remains pressed, providing a continuous auditory feedback. Furthermore, the keypad's initialization process has been designed for adaptability, accommodating various pin configurations by allowing explicit setting adjustments.

Task 2: Indefinite Frogger Game

Frogger Game Mechanics and Functionality:

Frogger is a classic arcade game that was first introduced in 1981 by Konami. The primary objective of the game is to guide a frog across a busy road and a treacherous river to reach one of the several lily pads or safe havens situated on the other side. Players navigate the frog through various levels of increasing difficulty, avoiding a myriad of obstacles along the way. Frogger requires players to guide a frog from the screen's bottom, bypassing diverse obstacles, to safely reach its home at the top.

Essential gameplay mechanics are:

- Movement Control: Players direct the frog up, down, left, and right.
- Hazards: Dodging vehicles to cross a road to Home.
- Scoring System: Accumulation of points hinges on time efficiency, various bonuses, and other gameplay dynamics.
- Timer: A continuously ticking clock pressures players, adding urgency to complete the goal.

Game Components:

a. Frog:

Symbol: "&" Functionality: The player's in-game avatar begins its journey at the screen's bottom, moving based on player commands.

b. Traffic:

Cars:

Symbol: "##8"

Description: Cars travel horizontally, populating specific rows, each with a distinct pace.

Trucks:

Symbol: "####T"

Description: With their extended length, trucks present a heightened challenge as they move horizontally.

c. Home Goal:

Position: Top row of the LCD.

Symbol: "█████████".

Purpose: The terminal point for the frog's journey. Advanced levels might introduce new challenges within these segments.

d. Timer:

Description: A visible timer counts down, adding urgency to the player. The player must guide the frog to its home before time runs out. Running out of time results in Game Over, making efficient navigation crucial.

Principles of Operation:

Game Operation and Mechanics:

The game starts with a frog at the bottom of a grid. Vehicles, classified as trucks ("####T") or cars ("##8"), spawn randomly in two rows and move either left or right based on their assigned row. The objective is to get the frog to the top without hitting any vehicles. This game is developed for an LCD display, utilizing functions like lcd_goto_position and lcd_write_data. The frog is represented as ‘&’ on the LCD. The top row has designated spots marked "GOAL" and obstacles shown as solid blocks. The frog has four movement directions: Up, Down, Left, and Right. Vehicles' movement direction is determined by their row. A game reset is triggered if the frog collides with a vehicle or wall or reaches the goal.

Algorithms and Pseudo-code:

//

```
Initialize game:
    Position frog at the bottom center of the grid
    Initialize vehicles to inactive
    Clear the grid and set goal positions

Game Loop:
    while game is running:
        Get player input
        if input is the exit key:
            Exit game
        Update game state based on input:
            Chance to spawn a new vehicle
            Update vehicle positions
            Update frog position based on player input
            Check collisions
            Render vehicles on the grid
            Update the frog's position on the grid
        Render the entire game on the LCD
        Delay for a short period
```

Update game state:

  Spawn vehicles occasionally

  Update vehicle positions:

    If vehicle is in top row, move right

    If vehicle is in bottom row, move left

    Deactivate vehicle if it's off the screen

  Check for collisions:

    For each vehicle:

      If vehicle position overlaps with frog position:

        Handle collision (reset game)

    If frog has reached the top and is in an empty space:

      Handle victory (reset game)

Handle collision:

  Decrease lives

  Play collision sound

  Reset game

Handle victory:

  Display victory message

  Play victory sound

  Reset game

//

Assumptions and Special Features:

The game utilizes keypad input, featuring directional keys and an Exit key. Vehicles, determined by their initial row, move either left or right. These vehicles are represented by either 5 or 3 characters, with second-row vehicles displaying inverted representations from the first row. Gameplay does not restrict players with a time limit or life count; a collision simply resets the game. Audio feedback for various game events is implied by the play_tone function. Players can view live and final scores on the LCD. A seeding method, activated upon game entry or exit, generates pseudo-random numbers for global application.

Task 3: Competitive Frogger Game

Scoring Mechanics and Differences from Basic Frogger:

In the "Scored Frogger" version, a dynamic scoring system enhances the basic Frogger gameplay. Players earn 20 points for each new row the frog enters. Upon reaching the top row's empty space, points are awarded based on the remaining time, with 10 points given for every 0.5 seconds left. A bonus of 500 points is added if the frog reaches home three times in one game. The primary objective is to guide the frog safely to the top grid position. Players use a keypad for direction, navigating a grid filled with moving vehicles and potential collision hazards. This version also integrates a scoring mechanism and incorporates sound cues for different game events.

Algorithms and Pseudo-code:

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Function play_scored_frogger(score):

    Prepare game (seed, display countdown, initialize game state)


    While game is running:

        Capture user input

        Update game state based on the input and current game scenario

        Render the updated game state on LCD

        Check if the game is over

        Update the timer on the display

        Decrease the timer

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Function update_game_state_scored(input, score reference):

Occasionally spawn vehicles

Update vehicles' positions

Clear grid cells except 'X' on the top row (score row)

Reset home positions

Update frog's position based on the input

Check for advancement in rows for score

Check for collisions with walls or special characters in home row

Update frog's position

Check if frog reached a goal

Check for other collisions

Render vehicles on grid

Display current score

Update grid with frog's new position

///

Function render_game_scored(score):

Render home row with special characters or spaces

Render other rows with game elements (vehicles, frog, etc.)

Display the current score

Indicate frog's lives using 'X' markers on the LCD

Render frog last, so it appears on top

//

Function check_game_over:

Check if the timer has run out or if the player has no lives left (Game Over)

Check if the frog has reached home 3 times (Victory)

Continue the game otherwise

//

Assumptions and Special Features:

The game concludes when either the timer depletes, the frog successfully reaches its home three times, or all frog lives are lost. An advanced scoring system has been implemented: players earn points for advancing rows, reaching the home, and gain extra bonuses for any residual time. The game also boasts improved audio feedback, providing tones in response to various game activities, such as movements and reaching objectives. Live scores and remaining lives, represented by 'X' markers, are visibly displayed on the LCD. This iteration of Frogger demonstrates enhanced gameplay with the integration of a scoring mechanism, auditory cues, and refined game visuals and checks.

Hardware Description

Core Components:

Microcontroller - ATmega328PB:

The ATmega328PB serves as the brain of the setup. It controls the generation of audio signals, manages user input, and displays information. Its versatile nature and a rich set of peripherals make it suitable for a range of applications, including audio processing.

Amplifier - LM386N-1 Low Voltage Audio Power Amplifier IC:

This amplifier is used to amplify the audio signals generated by the microcontroller. Its low voltage operation is compatible with the 5V microcontroller, and it can drive small speakers directly.

Speaker - Mylar Cone Speaker (0.25 WRMS, 8Ω, 27mm Diameter):

The selected speaker converts amplified electrical audio signals into sound. Mylar cone speakers are known for their clear sound reproduction in compact sizes, making them ideal for small audio devices.

Input - Keypad:

The keypad allows the user to interact with the system. It can be used to select modes, change settings, or input data, making the system interactive and user-friendly.

Audio Circuit Components:

Potentiometer - 10kΩ Logarithmic Taper Potentiometer:

The logarithmic taper potentiometer is used for volume control in the audio pathway. Its logarithmic nature ensures a more natural and linear volume change perception to the human ear when adjusted.

Resistors - R1: 4.7kΩ:

R1 acts as a current-limiting resistor. It ensures that the current flowing from the microcontroller's output pin (PB0) to the amplifier's input does not exceed safe limits, protecting both components.

Capacitors

C1 (220µF): This capacitor is used in the feedback loop of the amplifier to set the gain. Its value determines the amplification level of the audio signal.

C2 (10µF): Connected as a bypass capacitor, C2 helps filter out unwanted noise and interferences, ensuring clearer audio output.

Power Supply - 5v USB (external source to Microcontroller):

The circuit is powered using a 5v USB power supply. This ensures compatibility with standard USB chargers and provides a stable and regulated power source for the entire system.

Calculations for Circuit

Amplifier Gain Calculation (Using the LM386):

The gain of the LM386 can be adjusted by placing a capacitor between pins 1 and 8 (which are generally the Gain set pins).

Where: $Gain(A) = 20 \times (1 + \frac{R_1}{R_2} + \frac{C_1}{C_2})$

$R1$ and $R2$ are the internal resistances of the LM386, which are 1.35 kΩ and 150 Ω respectively.

$C1$ is the capacitor placed between the pins (220µF in this).

$C2$ is the internal capacitor of the LM386, which is around 15pF.

$$A = 20 \times \left(1 + \frac{1.35k\Omega}{150\Omega} + \frac{220\mu F}{15pF}\right)$$

$$A \approx 20 \times (10 + 14666.67) = 20 \times 14676.67 = 293533.4$$

While this seems high, the actual gain of the LM386 is limited to 200 due to saturation, so in effect we have just maximized the gain the LM386 can provide.

Bypass Capacitor:

The 10µF capacitor connected to the bypass pin helps reduce internal IC noise, thus improving the sound quality. There isn't a straightforward mathematical calculation for this, but it's a recommended practice in the LM386 datasheet.

Current Limiting Resistor:

The resistor (R1) between the microcontroller and the amplifier helps limit the current. The value of the resistor, combined with the impedance at the amplifier's input, creates a voltage divider.

If we assume the output impedance of the microcontroller to be negligible and the amplifier's input impedance to be much larger than 4.7kΩ:

$$V_{out} = V_{in} \times \frac{R_{input}}{R_{input}+R1}$$

Where:

- *V in* is the microcontroller's output voltage.
- *R input* is the amplifier's input impedance.
- *R*1 is 4.7kΩ.

Given the conditions mentioned above, *Vout* will be almost equal to *Vin*.

Volume Control:

The 10kΩ potentiometer allows for adjusting the amplitude of the input signal before amplification by modulating the resistance vs voltage. Generally for audio a logarithmic potentiometer will be used due to the change in volume perception also being logarithmic to human ears.

Conclusion

The journey through the intricacies of utilizing the ATmega328PB microcontroller for gaming has been both challenging and rewarding. By successfully designing a sound generation module, we have demonstrated that even minimalistic hardware platforms can be tuned to produce engaging auditory feedback. This capability not only adds depth to the gaming experience but also showcases the versatility of microcontroller applications.

Recreating the iconic arcade game, "Frogger," in two distinct versions on such a compact platform was no small feat. The infinite gameplay version pays homage to the game's classic mechanics, offering players the raw essence of "Frogger." Meanwhile, the competitive variant elevates the experience, pushing players to strive for higher scores and showcasing the adaptability of the ATmega328PB.

In synthesizing software algorithms with hardware functionalities, this project illuminates the profound possibilities lying within microcontrollers. The successful integration of sound and gaming on the ATmega328PB stands as a testament to the union of creativity and technical prowess. As technology continues to evolve, this project offers a nostalgic yet forward-looking glimpse into the future of gaming, where boundaries are constantly redefined and potential is only limited by one's imagination.

References

ATMEGA328PB datasheet ATmega328PB - Microchip Technology. (n.d.). https://ww1.microchip.com/downloads/en/DeviceDoc/40001906A.pdf

Frogger -- by Cindy Jih and Esther Jun. (n.d.). http://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2003/ctj4eej2/Project%20Website/appendix.html

GLBasic - multiplatform development in basic. RSS. (n.d.). http://www.glbasic.com/main.php?site=microprocessor&lang=en

Appendix

Figure 1: 10K Ohm Potentiometer

10k Potentiometer =

https://www.jaycar.com.au/medias/sys_master/images/images/9959413612574/RP8610-dataSheetMain.pdf

Figure 2: The mentioned speaker

27mm All Purpose Replacement Speaker (store page as data sheet not available)

https://www.jaycar.com.au/27mm-all-purpose-replacement-speaker/p/AS3002

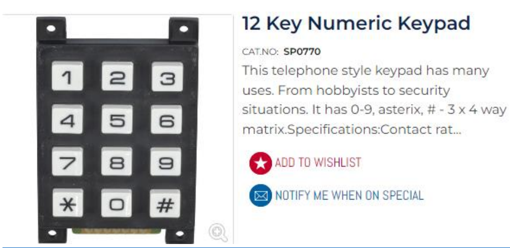

Figure 3: Keypad in use

12 Key Numeric Keypad

https://www.jaycar.com.au/medias/sys_master/images/images/9960835285022/SP0770-manualMain.pdf

Figure 4: Bypass Capacitor on OpAmp

10uF 16VDC Electrolytic RB Capacitor (store page as data sheet not available)

https://www.jaycar.com.au/10uf-16vdc-electrolytic-rb-capacitor/p/RE6066

Figure 5: Gain increase Capacitor

220uF 25VDC Low ESR Electrolytic Capacitor

https://www.jaycar.com.au/220uf-25vdc-low-esr-electrolytic-capacitor/p/RE6324?pos=9&queryId=bd3a64eecd0bf6e5a66b9952733c7bfc&sort=relevance

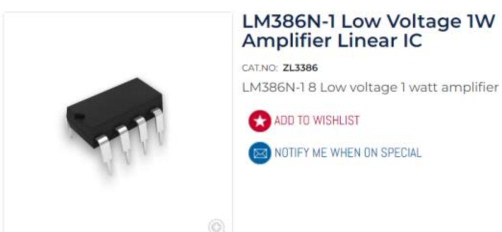

Figure 6: Provided Amplifier

LM386N-1 Low Voltage 1W Amplifier Linear IC

https://www.jaycar.com.au/medias/sys_master/images/images/9965288488990/ZL3386-dataSheetMain.pdf

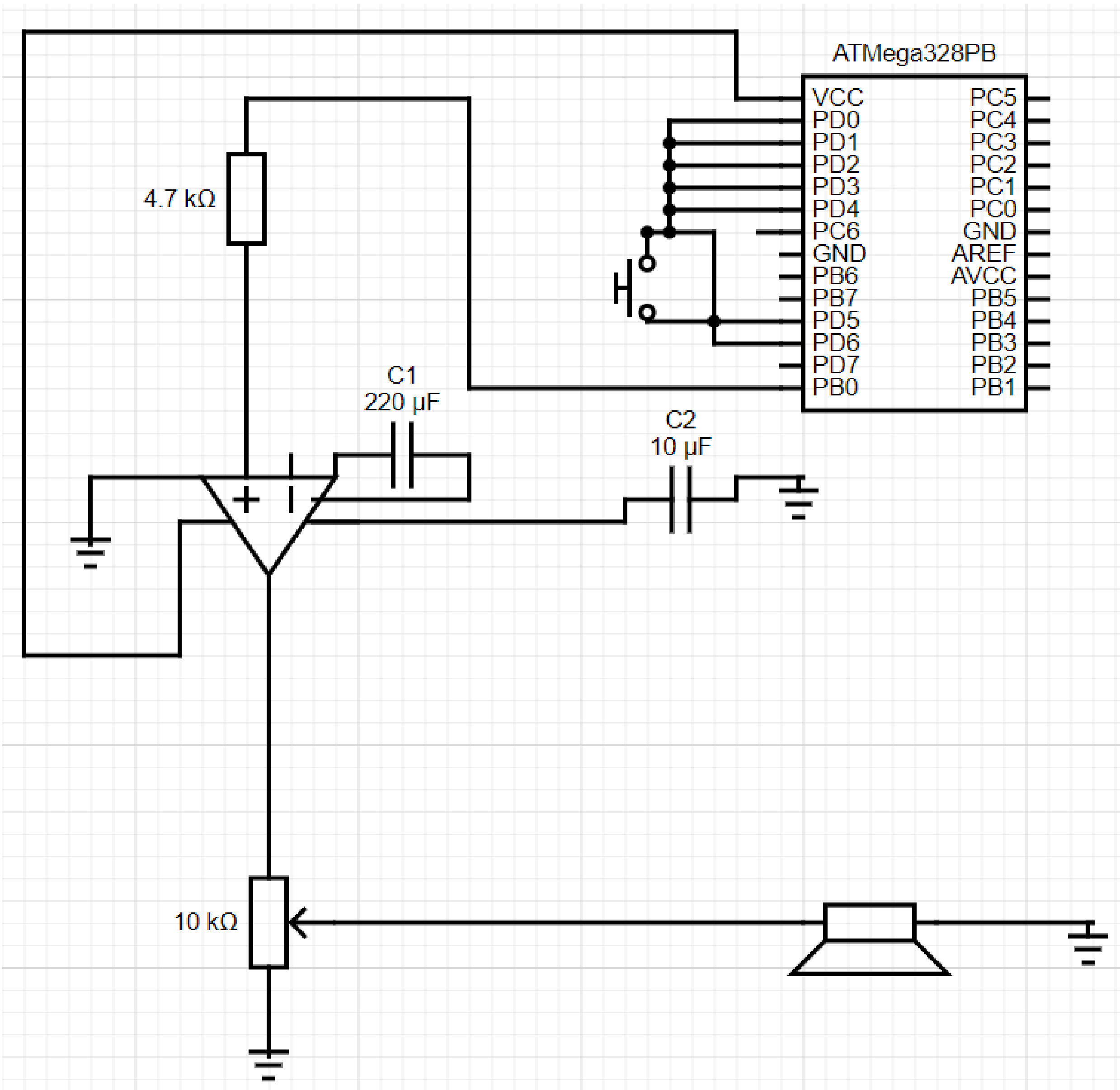

Figure 7: Circuit Diagram for Speaker Configuration

Appendix 2 – C Code

```
// Frogger Functionality
// for ENS2257/ENS6155 with Lab Kits with ATmega328PB

#define F_CPU 16000000UL

#include <atmel_start.h>
#include <stdint-gcc.h>
#include <stdio.h>
#include <avr/pgmspace.h>
#include <avr/iom328pb.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <lcd.h>
#include <math.h>
#include <string.h>
#include <avr/io.h>

// PIN DEFINITIONS:
// PD2 -- KEYPAD COL1 (left column)
// PD0 -- KEYPAD COL2 (center column)
// PD4 -- KEYPAD COL3 (right column)
// PD1 -- KEYPAD ROW1 (top row)
// PD6 -- KEYPAD ROW2 (second row)
// PD5 -- KEYPAD ROW3 (third row)
// PD3 -- KEYPAD ROW4 (fourth row)

#define GRID_WIDTH 20          //Grid Parameters for LCD Screen
#define GRID_HEIGHT 4

#define MAX_VEHICLES 5     // Maximum Vehicles to be generated at one time

// Note frequencies (in Hz) for C4 to B4
```

```
#define freq_C4 261
#define freq_D4 294
#define freq_E4 330
#define freq_F4 349
#define freq_G4 392
#define freq_A4 440
#define freq_B4 494
#define freq_C5 523

// Global variables
char grid[GRID_HEIGHT][GRID_WIDTH];
int frogX, frogY;
int frogMarkY, frogMarkX;
int frogMarkY2, frogMarkX2;
int lives = 3;
int previousFrogY = GRID_HEIGHT - 1;
int score = 0;
int frogsAtHome = 0;
float timer = 20;
int seed;

// Struct for Vehicles (Trucks and Cars)
typedef struct {
    int x;
    int y;
    int length;
    char representation[6];
    char reversed_representation[6];
    bool active;
    int type;
} Vehicle;
```

```
Vehicle vehicles[MAX_VEHICLES];

//*****************************************************************************************
//*                         KeyPad Functions and init - Used from Lab 3                   *
//*****************************************************************************************

//**************************************************
void keypad_init() {
  // Subroutine to set ports to required input and output states and enable pull up resistors for inputs

  // Set the columns as output
  DDRD |= (1<<PD0) | (1<<PD2) | (1<<PD4);

  // Set row pins to input mode
  DDRD &= ~(1<<PD1) & ~(1<<PD3) & ~(1<<PD5) & ~(1<<PD6);

  // Turn on the internal resistors for the input pins
  PORTD |= (1<<PD1) | (1<<PD3) | (1<<PD5) | (1<<PD6);

  // Initialise the column output pins low, so input low if contact made
  PORTD &= ~(1<<PD0) & ~(1<<PD2) & ~(1<<PD4);
}  // END keypad_init

//**************************************************

uint8_t keypressed() {
  uint8_t rowval;
  uint8_t kp;

  PORTD &= ~(1<<PD0) & ~(1<<PD2) & ~(1<<PD4);
  _delay_us(10);

  rowval = PIND & 0x6A;
```

```
    kp = (rowval != 0x6A);

    return kp;
}

//**************************************************

char keypad_read(char lastchar) {
    uint8_t rowval;
    char keych;

    keych = '$';

    PORTD &= ~(1<<PD2);
    PORTD |= (1<<PD0) | (1<<PD4);
    _delay_us(10);

    rowval = PIND & 0x6A;

    switch(rowval) {
        case 0x68: keych = '1'; break;
        case 0x2A: keych = '4'; break;
        case 0x4A: keych = '7'; break;
        case 0x62: keych = '*'; break;
        default: keych = '$'; break;
    }

    if (keych=='$') {
        PORTD &= ~(1<<PD0);
        PORTD |= (1<<PD2) | (1<<PD4);
        _delay_us(10);
```

```
        rowval = PIND & 0x6A;

        switch(rowval) {
            case 0x68: keych = '2'; break;
            case 0x2A: keych = '5'; break;
            case 0x4A: keych = '8'; break;
            case 0x62: keych = '0'; break;
            default: keych = '$'; break;
        }
    }

    if (keych=='$') {
        PORTD &= ~(1<<PD4);
        PORTD |= (1<<PD0) | (1<<PD2);
        _delay_us(10);

        rowval = PIND & 0x6A;

        switch(rowval) {
            case 0x68: keych = '3'; break;
            case 0x2A: keych = '6'; break;
            case 0x4A: keych = '9'; break;
            case 0x62: keych = '#'; break;
            default: keych = '$'; break;
        }
    }

    if (keych != '$') {
        lastchar = keych;
    }

    return lastchar;
} // END keypad_read
```

```
//*************************************************

uint8_t get_keypad_input() {
    if (keypressed()) {
        char keychar = keypad_read('$');
        return keychar;
    }
    return '$';
}

//*************************************************

void init_interrupts() {
    PCMSK0 = (1<<PCINT1);
    PCICR &= ~(1<<PCIE1) & ~(1<<PCIE2);
    PCICR |= (1<<PCIE0);
    sei();
}  // END init_interrupts

//*************************************************

//*****************************************************************************************
//*                                    Menu Functions                                     *
//*****************************************************************************************

//Displays Welcome Menu

void display_menu(void) {
    lcd_clear_and_home();
    lcd_write_string(PSTR("1: Play Tones"));
    lcd_goto_position(1, 0);
    lcd_write_string(PSTR("2: Play Frogger"));
```

```
    lcd_goto_position(2, 0);
    lcd_write_string(PSTR("3: Scored Frogger"));
    lcd_goto_position(3, 0);
    lcd_write_string(PSTR("4: C Major"));
}


//Displays 3x4 grid for tone generation choice
void display_tone_grid(void) {
    lcd_clear_and_home();
    lcd_goto_position(0, 8);
    lcd_write_string(PSTR("CDE"));
    lcd_goto_position(1, 8);
    lcd_write_string(PSTR("FGA"));
    lcd_goto_position(2, 8);
    lcd_write_string(PSTR("BC."));
    lcd_goto_position(3, 8);
    lcd_write_string(PSTR("..X"));
}

//******************************************************************************************
//*                         Task 1 - Function 1 - Sound Production                         *
//******************************************************************************************

//Custom delay_us to allow for dynamic variables to be used (unlike AVR Library _delay_us

void delay_us(uint16_t us) {
    while (us--) {
        _delay_us(1);
    }
}

void play_tone(uint16_t frequency) {
```

```
    uint16_t delayTime = 500000 / frequency / 2;

    for (uint32_t i = 0; i < (frequency / 2)/4; ++i) {
        PORTB |= (1 << PB0);
        delay_us(delayTime);
        PORTB &= ~(1 << PB0);
        delay_us(delayTime);
    }
}

void hold_tone(uint16_t frequency) {
    uint16_t delayTime = 500000 / frequency / 2;

    while (keypressed()){
        PORTB |= (1 << PB0);
        delay_us(delayTime);
        PORTB &= ~(1 << PB0);
        delay_us(delayTime);
    }
}

void play_tones(void) {
    display_tone_grid();

    while(1) {
        char input = get_keypad_input();

        if (input == '#') {
            display_menu();
            return;
        }

        switch(input) {
```

```
            case '1':
                hold_tone(freq_C4);
                break;
            case '2':
                hold_tone(freq_D4);
                break;
            case '3':
                hold_tone(freq_E4);
                break;
            case '4':
                hold_tone(freq_F4);
                break;
            case '5':
                hold_tone(freq_G4);
                break;
            case '6':
                hold_tone(freq_A4);
                break;
            case '7':
                hold_tone(freq_B4);
                break;
            case '8':
                hold_tone(freq_C5);
                break;
            default:
                break;
        }
    }
}

void c_scale() {
    DDRB |= (1 << PB0);
```

```
    play_tone(freq_C4);
    _delay_ms(500);
    play_tone(freq_D4);
    _delay_ms(500);
    play_tone(freq_E4);
    _delay_ms(500);
    play_tone(freq_F4);
    _delay_ms(500);
    play_tone(freq_G4);
    _delay_ms(500);
    play_tone(freq_A4);
    _delay_ms(500);
    play_tone(freq_B4);
    _delay_ms(500);
    play_tone(freq_C5);
    _delay_ms(500);

    display_menu();
    return;
}

//**************************************************************************************
//*                     Task 1 and 2 - Shared Functions Frogger                        *
//**************************************************************************************

//Checks if space is clear for vehicle spawn
bool is_space_clear(int x, int y, int length) {
    for (int i = 0; i < length; i++) {
        if (grid[y][x + i] != ' ') {
            return false;
        }
    }
    return true;
```

```
}

//Reverse Vehicle string for both directions
void reverse_string(char* str, char* reversed, int length) {
    for (int i = 0; i < length; i++) {
        reversed[i] = str[length - i - 1];
    }
    reversed[length] = '\0';
}

void spawn_vehicle() {
    for (int i = 0; i < MAX_VEHICLES; i++) {
        if (!vehicles[i].active) {
            vehicles[i].active = true;
            vehicles[i].y = rand() % 2 + 1;
            int length;
            if (rand() % 2 == 0) {
                length = 5;
                strcpy(vehicles[i].representation, "####T");
            } else {
                length = 3;
                strcpy(vehicles[i].representation, "##8");
            }
            reverse_string(vehicles[i].representation, vehicles[i].reversed_representation, length);
            vehicles[i].length = length;
            if (vehicles[i].y == 1) {
                vehicles[i].x = 0;
                if (!is_space_clear(vehicles[i].x, vehicles[i].y, length)) {
                    vehicles[i].active = false;
                    continue;
                }
            } else {
                vehicles[i].x = GRID_WIDTH - length;
```

```
                if (!is_space_clear(vehicles[i].x, vehicles[i].y, length)) {
                    vehicles[i].active = false;
                    continue;
                }
            }
            break;
        }
    }
}

void update_vehicles() {
    for (int i = 0; i < MAX_VEHICLES; i++) {
        if (vehicles[i].active) {
            if (vehicles[i].y == 1) {
                vehicles[i].x += 1;
            } else {
                vehicles[i].x -= 1;
            }

            if (vehicles[i].x < 0 || vehicles[i].x + vehicles[i].length > GRID_WIDTH) {
                vehicles[i].active = false;
            }
        }
    }
}

void render_vehicles() {
        for (int i = 0; i < MAX_VEHICLES; i++) {
                if (vehicles[i].active) {
                        int x = vehicles[i].x;
                        int y = vehicles[i].y;
                        char* representation = vehicles[i].y == 1 ? vehicles[i].representation : 
vehicles[i].reversed_representation;
                        for (int j = 0; j < vehicles[i].length; j++) {
```

```
                        if (x + j < GRID_WIDTH && y < GRID_HEIGHT) {
                            grid[y][x + j] = representation[j];
                        }
                    }
                }
            }
}

void reset_vehicles() {
        for (int i = 0; i < MAX_VEHICLES; i++) {
                vehicles[i].active = false;
        }
}

//Game State

void reset_game() {
        // Reset the frog position
        frogX = GRID_WIDTH / 2;
        frogY = GRID_HEIGHT - 1;
        previousFrogY = frogY;

        // Reset the score, timer, and frogs at home count
        score = 0;
        timer = 20;

        if (frogsAtHome == 3){
                frogsAtHome = 0;
        }

        reset_vehicles();

        // Clear the game grid
```

```
	for (int y = 0; y < GRID_HEIGHT; y++) {
		for (int x = 0; x < GRID_WIDTH; x++) {
			grid[y][x] = ' ';
		}
	}

	// Re-initialize home positions
	for (int x = 0; x < GRID_WIDTH; x++) {
		if (x == 7) {
			grid[0][x] = 'G';
			} else if (x == 9) {
			grid[0][x] = 'O';
			} else if (x == 11) {
			grid[0][x] = 'A';
			} else if (x == 13) {
			grid[0][x] = 'L';
			} else if (x % 2 == 0 || x % 3 == 0) {
			grid[0][x] = '?';
			} else {
			grid[0][x] = ' ';
		}
	}
}

void game_init() {
	reset_game();
	// Clear grid
	for (int y = 1; y < GRID_HEIGHT; y++) {
		for (int x = 0; x < GRID_WIDTH; x++) {
			grid[y][x] = ' ';
		}
	}
}
```

```
//Scoring

void display_live_score(int score) {
	lcd_goto_position(3, 0);
	lcd_write_int16(score);
	_delay_ms(100);
}

void display_final_score(int score) {
	lcd_clear_and_home();
	lcd_goto_position(1, 4);
	lcd_write_string(PSTR("Score: "));
	lcd_write_int16(score);
	_delay_ms(2000);
}

//Collisions

void handle_collision() {
	lives -= 1;
	lcd_clear_and_home();
	lcd_goto_position(1, 5);
	lcd_write_string(PSTR("SPLAT!"));
	play_tone(freq_E4);
	play_tone(freq_C4);
	_delay_ms(1000);
	game_init();
}

void check_collisions() {
	for (int i = 0; i < MAX_VEHICLES; i++) {
		if (vehicles[i].active) {
```

```
                        int x = vehicles[i].x;
                        int y = vehicles[i].y;
                        for (int j = 0; j < vehicles[i].length; j++) {
                                if (frogX == x + j && frogY == y) {
                                        handle_collision();
                                        return;
                                }
                        }
                }
        }

        // Check for victory
        if (frogY == 0 && grid[frogY][frogX] == ' ') {
                // Victory
                lcd_clear_and_home();
                lcd_goto_position(1, 6);
                lcd_write_string(PSTR("Victory!"));
                play_tone(freq_C4);
                play_tone(freq_C4);
                play_tone(freq_C4);
                play_tone(freq_E4);
                play_tone(freq_C4);
                game_init();  // Reset the game
                return;
        }
}

//*****************************************************************************************
//*                         Task 1 - Function 2 - Indefinite Frogger                      *
//*****************************************************************************************

void update_game_state(char input) {
  // Spawn vehicles and update their positions
```

```
if (rand() % 3 == 0) {
    spawn_vehicle();
}
update_vehicles();

// Clear grid
for (int y = 0; y < GRID_HEIGHT; y++) {
    for (int x = 0; x < GRID_WIDTH; x++) {
        grid[y][x] = ' ';
    }
}

// Re-initialize home positions
for (int x = 0; x < GRID_WIDTH; x++) {
    if (x % 2 == 0 || x % 3 == 0) {
        grid[0][x] = '?';
    }
}

// Update frog position based on input
int newFrogX = frogX;
int newFrogY = frogY;
if (input == '8' && frogY > 0) newFrogY--;  // Up
if (input == '0' && frogY < GRID_HEIGHT - 1) newFrogY++;  // Down
if (input == '*' && frogX > 0) newFrogX--; // Left
if (input == '#' && frogX < GRID_WIDTH - 1) newFrogX++;  // Right

if(keypressed()) play_tone(freq_F4);

// Check if the new position is a wall
if (grid[newFrogY][newFrogX] == '?') {
    handle_collision();  // Call the function if a wall is hit
    return;  // Exit early from update_game_state
```

```
    }

    // If not, update the frog's position
    frogX = newFrogX;
    frogY = newFrogY;

    // Check for collisions
    check_collisions();

    render_vehicles();

    // Update grid with new positions
    grid[frogY][frogX] = '&';
}

void render_game() {
    lcd_home();
    // Render home row
    lcd_goto_position(0, 0);  // Set the cursor position at the beginning of the first row
    for (int x = 0; x < GRID_WIDTH; x++) {
        if (x % 2 == 0 || x % 3 == 0) {
            lcd_write_data(0xFF);  // Write block character to LCD
        } else {
            lcd_write_data(' ');  // Write space to LCD
        }
    }
    // Render the rest of the grid
    for (int y = 1; y < GRID_HEIGHT; y++) {
        lcd_goto_position(y, 0);
        for (int x = 0; x < GRID_WIDTH; x++) {
            lcd_write_data(grid[y][x]);
        }
    }
```

```
        // Render the frog last, so it appears on top of other elements
    lcd_goto_position(frogY, frogX);
    lcd_write_data('&');
}

void play_frogger(void) {
    seed += 9;
    game_init();
    while(1) {
        char input = get_keypad_input();
        update_game_state(input);
        render_game();
        _delay_ms(100);
        if (input == '1') {  // '1' is the key to exit back to the menu
            seed -= 3;
            lives = 3;
            display_menu();
            return;
        }
    }
}

//*******************************************************************************************
//*                         Task 2 - Function 2 - Scored Frogger                            *
//*******************************************************************************************

// Spawn vehicles and update their positions
void update_game_state_scored(char input, int* score) {
    if (rand() % 2 == 0) {  // 10% chance to spawn a new vehicle each frame
        spawn_vehicle();
    }
    update_vehicles();
```

```
// Clear grid but preserve 'X' characters in the scoring row (row 0)
for (int y = 0; y < GRID_HEIGHT; y++) {
    for (int x = 0; x < GRID_WIDTH; x++) {
        if (y != 0) {
            grid[y][x] = ' ';
        } else {
            if (grid[y][x] != 'X') {
                grid[y][x] = ' ';
            }
        }
    }
}

// Re-initialize home positions
for (int x = 0; x < GRID_WIDTH; x++) {
    if (x % 2 == 0 || x % 3 == 0) {
        grid[0][x] = '?';
    } else {
        grid[3][x] = ' ';
    }
}

// Update frog position based on input and play sound
int newFrogX = frogX;
int newFrogY = frogY;
if (input == '8' && frogY > 0) newFrogY--;
if (input == '0' && frogY < GRID_HEIGHT - 1) newFrogY++;
if (input == '*' && frogX > 0) newFrogX--;
if (input == '#' && frogX < GRID_WIDTH - 1) newFrogX++;
if(keypressed()) play_tone(freq_F4);

// Check if the frog has entered a new row
if (newFrogY != previousFrogY) {
```

```
    if (newFrogY != 0) {
        *score += 20;
    }
    previousFrogY = newFrogY;
}

// Check if the new position is a wall or X on home row
if (grid[newFrogY][newFrogX] == '?' || (grid[newFrogY][newFrogX] == 'X' && newFrogY == 0)) {
    handle_collision();
    return;
}

frogX = newFrogX;
frogY = newFrogY;

// Check if the frog has reached a goal
if (frogY == 0 && grid[frogY][frogX] == ' ' ) {
    play_tone(freq_C4);
    play_tone(freq_E4);
    frogsAtHome += 1;
    lives -= 1;
    *score += 10 * (timer + 1);

                        //Update Frog Position if at home
    if(frogsAtHome > 0 && frogsAtHome < 2) {
        frogMarkY = frogY;
        frogMarkX = frogX;
    }

    if(frogsAtHome > 1) {
        frogMarkY2 = frogY;
        frogMarkX2 = frogX;
    }
```

```
        frogX = GRID_WIDTH / 2;
        frogY = GRID_HEIGHT - 1;
        timer = 20;

        if (frogsAtHome == 3) {
            *score += 500;
        }
    }

    check_collisions();
    render_vehicles();
    display_live_score(*score);
    grid[frogY][frogX] = '&';
}

void render_game_scored(int score) {
    lcd_home();
    lcd_goto_position(0, 0);
    for (int x = 0; x < GRID_WIDTH; x++) {
        if (x % 2 == 0 || x % 3 == 0) {
            lcd_write_data(0xFF);
        } else {
            lcd_write_data(' ');
        }
    }

    for (int y = 1; y < GRID_HEIGHT; y++) {
        lcd_goto_position(y, 0);
        for (int x = 0; x < GRID_WIDTH; x++) {
            lcd_write_data(grid[y][x]);
        }
    }
```

```
display_live_score(score);

        //if Frog at home remove Life/Frogs left to get home
if(frogsAtHome > 0 ) {
   grid[frogMarkY][frogMarkX] = 'X';
   lcd_goto_position(frogMarkY, frogMarkX);
   lcd_write_data('X');
}

if(frogsAtHome == 2) {
   grid[frogMarkY2][frogMarkX2] = 'X';
   lcd_goto_position(frogMarkY2, frogMarkX2);
   lcd_write_data('X');
}

if(lives > 2) {
   lcd_goto_position(0,19);
   lcd_write_data('X');
} else {
   lcd_goto_position(0,19);
   lcd_write_data(' ');
}

if(lives > 1) {
   lcd_goto_position(1,19);
   lcd_write_data('X');
} else {
   lcd_goto_position(1,19);
   lcd_write_data(' ');
}

if(lives > 0) {
```

```
        lcd_goto_position(2,19);
        lcd_write_data('X');
    } else {
        lcd_goto_position(2,19);
        lcd_write_data(' ');
    }

    lcd_goto_position(frogY, frogX);
    lcd_write_data('&');
}

int check_game_over() {
    if (timer <= 0 || lives <= 0) {
        return 1;
    }
    if (frogsAtHome == 3) {
        return 2;
    }
    return 0;
}

void play_scored_frogger(int score) {
    seed += 5;

    for (int i = 3; i > 0; i--) {
        lcd_clear_and_home();
        lcd_goto_position(1, 8);
        lcd_write_int16(i);
        play_tone(freq_C4);
        _delay_ms(500);
    }
    lcd_clear_and_home();
    lcd_goto_position(1, 8);
```

```
lcd_write_string(PSTR("Go!"));
play_tone(freq_C5);
_delay_ms(1000);

game_init();

while(1) {
    char input = get_keypad_input();
    update_game_state_scored(input, &score);
    render_game_scored(score);
    int gameStatus = check_game_over();

    if (gameStatus == 1 || input == '1') {
        seed -= 2;
        lcd_clear_and_home();
        lcd_goto_position(1, 6);
        lcd_write_string(PSTR("Game Over!"));
        play_tone(freq_E4);
        play_tone(freq_E4);
        play_tone(freq_D4);
        play_tone(freq_D4);
        play_tone(freq_C4);
        play_tone(freq_C4);
        display_final_score(score);
        lives = 3;
        frogsAtHome = 0;
                                display_menu();
        break;
    }
    if (gameStatus == 2) {
        seed += 10;
        lcd_clear_and_home();
        lcd_goto_position(1, 6);
```

```
            lcd_write_string(PSTR("You Win!"));
            play_tone(freq_E4);
            play_tone(freq_G4);
            play_tone(freq_A4);
            play_tone(freq_G4);
            play_tone(freq_F4);
            play_tone(freq_E4);
            play_tone(freq_D4);
            play_tone(freq_C4);
            display_final_score(score);
            lives = 3;
            frogsAtHome = 0;
                                display_menu();
            break;
        }
    }
}

//*************************************************************************************************
//*                                          Main                                                 *
//*************************************************************************************************

int main(void) {
    // initialise MCU, drivers and middleware
    atmel_start_init();

    // Initialise the LCD
    lcd_init();
    lcd_home();

    // Initialise keypad, and the SPST switch port
    keypad_init();
```

```c
    // Wait a moment, then initialise interrupts
    _delay_ms(1);
    init_interrupts();
    cli();  // Disable interrupts again initially
    display_menu();

    while (1) {
        char choice = get_keypad_input();  // Get user selection from keypad
        srand(seed);
        _delay_ms(500);  // Add a small delay to ensure any previous keypad input has been cleared

        switch (choice) {
            case '1':
                play_tones();
                break;
            case '2':
                play_frogger();
                break;
            case '3':
                play_scored_frogger(score);
                break;
            case '4':
                c_scale();
                break;
            default:
                // Invalid choice
                break;
        }
    }
    return 0;
}
```