Simon Game
PROJECT 4 WRITE-UP

Riley Karp
CS342 FALL 2018

Figure 1. Image of the original Hasbro Simon Game

Abstract

Do you miss vintage handheld games like Bop-It, Tamagotchi, and Simon? Obviously, yes you do. Well you're in luck! I'm bringing back the Simon game! In this fun game, a pattern of lights and corresponding sounds will be randomly generated one at a time from the 4 available options. The player must repeat back the pattern by pressing the button that corresponds to the proper lights. Each time the player gets the pattern right, the game will show the pattern again and add on a new light/note to the end. This continues until the player inputs the incorrect pattern, causing the game to end. Think this is too easy? Well you can increase the difficulty level by turning a dial (potentiometer) to increase the number of new notes that are added to the end of the pattern each time. The system receives analog voltages from the potentiometer, which then uses ADC (analog-to-digital conversion) to convert the analog signal into a digital code, which will be used to determine

how many new notes to add at the end of the pattern. The system's inputs are from the IR remote button presses and a potentiometer to determine the difficulty level, and the outputs are the the 4 LEDs and the sounds through the piezo element. The potentiometer must be calibrated to ensure an accurate mapping between output voltages and actual physical angle, which will be used to determine difficulty level. I chose to implement this project because I miss old games like the Simon game, which are way better than boring new iPhone games.

Resources

Parts list

- Any computer with the Arduino IDE installed (x1)
- IRLib2 Arduino Library
- Metro Mini development board (x1)
- USB-microUSB data cable (x1)
- insulated copper wire (~1 meter)
- full-sized breadboard (x1)
- resistor in the [500 Ω, 2.2 kΩ] range (x4)
- LED's (x4)
- TSOP382 IR receiver (x1)
- IR remote control (x1)
- Piezo device (x1)
- potentiometer (x1)

References

- Datasheets (in our Google Drive folder)
    - Metro_mini_schem.png
    - ATmega328_datasheet.pdf
    - ir-sensor.pdf
    - Tsop382.pdf

Procedure

Wiring the Circuit

The wiring is pretty straightforward and easy to follow by looking at the wiring diagram below. It is important to note that all resistors shown have 3 red bands indicating that they are all 2.2 kΩ resistors. This is fine since the resistors wired in series with the LEDs should be 500 Ω to 2.2 kΩ, but in my design, I used 560 Ω resistors. The LED's resistors should be within that range to ensure the LED does not get ruined by having too much current flow through it, but the resistance is also small enough that the light from the LED can still be seen. The IC shown in the diagram represents the IR receiver.

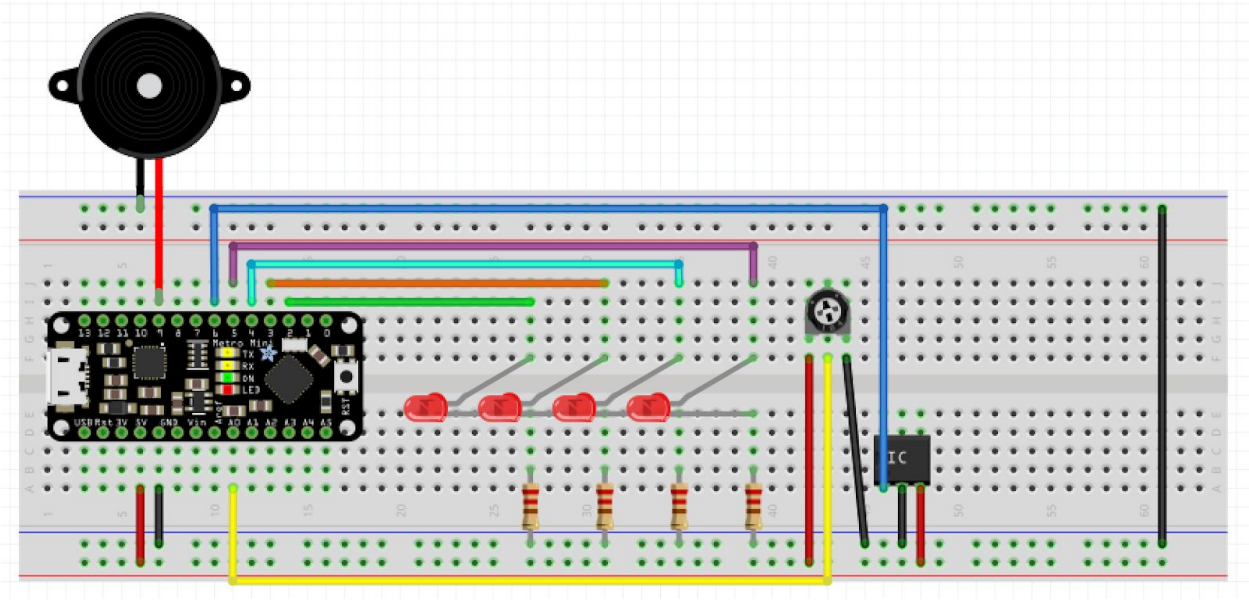

Figure 2. Wiring diagram showing how to construct the circuit

Writing the Code

The system is controlled by using Timer/Counter 0 to determine how long the LEDs should be on and the sound should be played, and to periodically take ADC readings from the potentiometer when ADC is enabled; Timer/Counter 1 is used to control the sound frequency to be played by the piezo; and the IR remote and receiver are used to control what the game does.

- Include the following IR communication libraries:
    - IRLibDecodeBase.h
    - IRLib_P01_NEC.h
    - IRLibRecv.h
- Define the button codes for each button (at least the buttons we'll be using, which are Button 0, Button 1, Button 2, Button 4, Button 5, and the enter button).
- Create 3 global variables to handle the IR remote inputs:

    - IRdecodeNEC nec: NEC decoder object
    - IRrecv rx(2): the TSOP38238 IR receiver is connected to pin 2
    - uint32_t valuePrevious: handles NEC repeat codes
- Create 11 global variables used for gameplay
    - const int maxLength = 20: maximum length of the pattern
    - int pattern[maxLength]: holds the randomly generated pattern of LEDs to be displayed
    - int idx = 0: current index in the pattern array
    - int numItems = 0: current number of items in the pattern
    - int ms = 0: for note duration
    - bool on = false: boolean to tell if any lights are on
    - int level = 1: level corresponding to # of new notes to add to the end of the pattern
    - bool repeat = false: boolean to tell if a button was repeated by being held down
    - bool won = false: boolean set to true if player wins
    - bool lost = false: boolean set to true if the player loses
    - int colorCount = 0: used to determine pattern of lights to display when the game ends
- Create 3 global variables to keep track of the locations and note frequencies for each color of LED. Index 0: red, 1: blue, 2: yellow, 3: green
    - int location[4] = { 0x40, 0x20, 0x10, 0x08 }: Array to hold LED locations in pad D
    - float freq[4] = { 880.00, 659.25, 554.37, 329.63 }: Array to hold note frequencies corresponding to each LED
    - int color = -1; // integer corresponding to the color of the LED that will light up when a button is pressed. initialized to -1 because no button is pressed
- Create 5 global variables for ADC reading
    - int nSamples = 0: number of ADC measurements taken
    - int N = 10: sliding window width
    - float omean = 0.0: online mean ADC value
    - bool complete = false: tells the main loop if ADC reading is complete
    - uint16_t sensedADC: most recent ADC measurement
- Create 4 functions:
    - void restart(): Restart Game: reinitialize global variables. Create new pattern & display the pattern
        - Set idx = 0
        - Set numItems = 0
        - Reinitialize won and lost booleans to false
        - Use a for loop and the newColor() function to add new colors to the pattern. The number of colors to be added is based on the level
        - Show the pattern using the showPattern() function
    - void newColor(): Generate random number to add to pattern of LEDs
        - insert a random number between 0 and 3 to the next empty spot in the pattern array

        - Increment numItems
    - void showColor(int color): Display a single LED & play the corresponding note
        - Initialize ms = 0
        - Set on to true
        - Set PIND to first turn off any LEDs that are on, and then turn on the correct LED based on the given color
        - Calculate and set the compare match A value of Timer/Counter 1 based on the note frequency corresponding to the given color
    - void showPattern(): Display the whole pattern of LEDs and sounds
        - Print the new pattern length to the serial monitor
        - Use a for loop to and showColor() to display each color in the pattern array
            - Call showColor()
            - Add a print statement to give the showColor() function time to set up everything it needs to do with the timer/counter 1
            - Put a while loop that continues until the LED & sound are turned off
        - Add a print statement indicating that the displaying is done
- In the main() function:
    - Configure GPIO pins
        - PD[6:3] should be outputs for the LEDs
        - PD[2] should be an input for the IR receiver
        - PB[1] should be an output for the piezo
        - PC[0] should be an input for the potentiometer
    - To set up Timer/Counter 1 for the piezo:
        - Set Timer/Counter 1 to CTC, toggle on compare match mode
        - Initialize to no prescale so the system starts off (we will eventually prescale by 1 in the TIMER0_COMPA ISR)
    - To set up Timer/Counter 0 for a 1ms compare match interrupt:
        - Set Timer/Counter 0 to CTC mode, toggle on compare match, and enable output compare match A
        - Prescale by 64
        - Set the OCR0A register to 250 so a compare match will occur every 1ms
        - Enable the ISR for Timer/Counter 0 Output Compare match A
    - To setup the ADC to periodically collect readings on PC[0]:
        - Set ADMUX so the ADC is right adjusted, on PC[0], and has reference voltage Vcc of 5V
        - Set ADCSRA to:
            - Automatic conversions
            - Prescale by 128
            - Don’t enable ADC or its ISR yet
        - Set ADCSRB so ADC conversions are auto-triggered by TIMER0_COMPA events

        - Set DIDR0 to disable digital inputs on the pins that aren't in use to save power
    - Enable global interrupts
    - Initialize the USART to enable printing to the serial monitor
    - Start the receiver using rx.enableIRIn()
    - Call the restart() function to start the game
    - Include an infinite loop that does the following:
        - Initialize repeat to false
        - Initialize color to -1
        - If a button is pressed
            - Decode the pulse train
            - If it's a repeat
                - Keep the previous value
                - Set repeat to true
            - Create a case-switch statement to determine what happens for each nec.value
                - For the select button
                    - Enable ADC measurements and its ISR
                    - Print that the select button was pressed
                - For button 0
                    - Call the restart function
                    - Print that button 0 was pressed
                - For button 1
                    - Use showColor() to display the green LED
                    - Set color to 3
                    - Print which button was pressed
                - For button 2
                    - Use showColor() to display the red LED
                    - Set color to 0
                    - Print which button was pressed
                - For button 4
                    - Use showColor() to display the yellow LED
                    - Set color to 2
                    - Print which button was pressed
                - For button 5
                    - Use showColor() to display the blue LED
                    - Set color to 1
                    - Print which button was pressed
            - If complete is true then
                - Calculate the angle based on the sensor characterization equation: float angle = (-0.27)*omean + 219
                - Calculate the level based on the angle: level = int(angle*5/180)

        - Make sure the level is between 1 and 5
        - Print the new level
        - Set complete to false
    - If color > -1 and repeat,won,lost are all false then
        - If the incorrect button was pressed (pattern[idx]!=color) then
            - Set lost to true
            - Print a message to tell the player they lost
        - Else if the correct button was pressed (pattern[idx]=color) then
            - Increment idx
        - If the end of the current pattern was reached (idx >= numItems) then
            - If there is no space left in the pattern array then
                - Set won to true
                - Print a message telling the player they won
            - Else (the pattern array still has space)
                - Add new item(s) based on the level
                - Check to make sure the array length is not exceeded when you add new items
            - Initialize idx to 0
            - Print a statement telling the player the pattern was correct
            - Show the new pattern
    - Set valuePrevious to nec.value
    - Start the receiver
- If lost is true and ms%500 is 0 then
    - Set on to false
    - If colorCounter is odd
        - Turn off all LEDs and sound
        - Print empty string to give the timer enough time to change its prescale before continuing
    - Else
        - Turn on all LEDs and sound
        - Print empty string to give the timer enough time to change its prescale before continuing
    - Increment colorCounter
- If won is true and ms%500 is 0 then
    - showColor(colorCount%4
    - Increment colorCounter
- Return 0

- In the ADC ISR (Read ADC value every 1ms when enabled):
    - Increment nSamples

    - Assign the low byte of the ADC register to sensedADC
    - Shift the low byte (sensedADC) into the high byte of the ADC register to obtain 10-bit precision (Something like this: sensedADC |= (ADCH & 0x03) << 8)
    - Set omean to the online filtered value:
        - omean = (omean * float(N-1)/float(N)) + (float(sensedADC)/float(N))
    - If nSamples > N then
        - Disable ADC
        - Set complete to true
        - Reinitialize nSamples to 0
- In the Timer/Counter 0 Compare Match A ISR (Interrupt triggered every 1ms. Increment counter that controls note duration. turn off sound and lights after 500ms)
    - Increment ms
    - If the game is on then
        - Set timer/counter 1's prescale to 1 so the sound turns on
    - If ms>=50 and the player hasn't lost then
        - Set on to false
        - Set timer/counter 1's prescale to 1 so the sound turns off
        - Set PORTD to turn off all 4 LEDs

The full code can be seen below in Appendix A.

Sensor Characterization

Sensor characterization is necessary to get an idea of the range of values that a specific sensor can produce, and its approximate accuracy. One sensor's values are likely different from another sensor's values or the values stated by the manufacturer. To characterize a sensor, position the sensor at known values (physical units) and record the resulting ADC code that the sensor produces at each physical value. For example, to characterize my potentiometer, I placed my sensor at known measured angles (in degrees), and recorded the corresponding ADC code (specifically, the mean ADC code of multiple measurements). I repeated these measurements from 0 degrees to 180 degrees, and then from 180 degrees to 0 degrees, to account for the effect of *hysteresis.* Hysteresis causes measurements to be a little lower than they actually are when you're increasing (i.e 0 degrees to 180 degrees), and a little higher than they actually are when you're decreasing (180 degrees to 0 degrees). This is because the sensor has a small amount of "electrical memory" or voltage left from the previous measurement, since voltage cannot change instantaneously. After collecting these measurements, plot physical distance units vs. ADC code and create a mapping function from ADC code to distance using the monotonic region of the plot. My mapping function for my potentiometer sensor can be seen in the results section below.

Results

Sensor Characterization

ADC Code vs. Potentiometer Angle was plotted for the min, max, and mean ADC codes at each angle. Since the three lines in the plot below are all fairly close and overlapping, that means that there is not much noise in the potentiometer measurements.

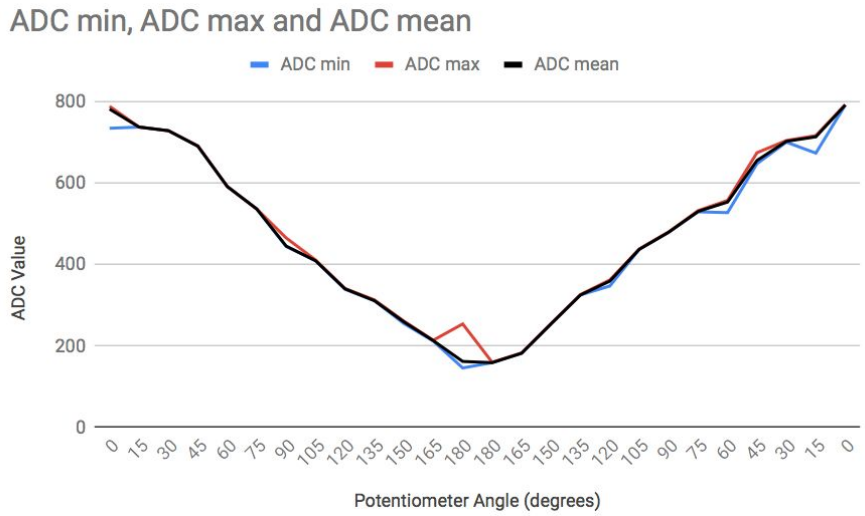

Figure 3. Plot showing the relationship between ADC code and potentiometer angle in degrees

The monotonic region of my angle vs ADC code plot can be seen below. A linear equation was fit to the data, which resulted in the following mapping from ADC code to angle in degrees:

Angle = -0.27*ADC + 219

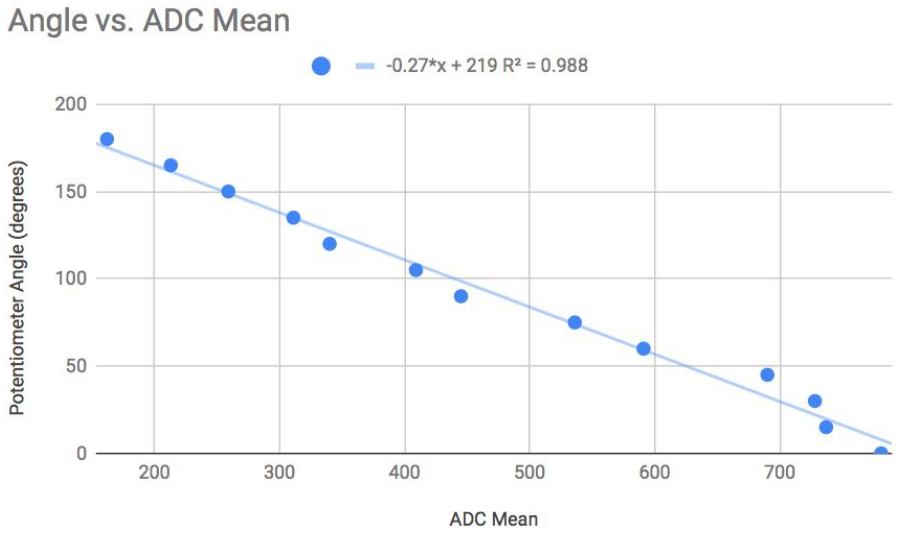

Figure 3. Scatterplot showing the relationship between angle in degrees and ADC code for my potentiometer sensor

System Performance

The Simon Game works just as expected, and the performance can be seen in the demo videos below. Video 1 shows that the game restarts when button 0 is pressed on the IR remote. A pattern of lights and sounds is displayed, and the game responds to button presses to play back the pattern in the correct order. The maxLength of the pattern is set to 5 for the demonstration, so when all 5 colors of the pattern are correctly played back, the “winning” pattern is displayed, until the button 0 is pressed to restart the game. In Video 2, the game is played again; however, the pattern that the user inputs is not correct, so the “losing” pattern is displayed, flashing all of the lights on and off every 500ms until button 0 is pressed to restart the game. Video 3 shows how to change the level. The potentiometer is turned and then the “enter/save” button is pressed on the IR remote to select the new level. The new level is printed out, and it determines how many new colors are added to the pattern after each correct answer. This is shown for all 5 levels.

Demo Videos

- Video 1: Playing & winning the game :)
- Video 2: Playing & losing the game :(
- Video 3: Changing levels

Discussion

This project could be improved by using a better filtering method to process ADC readings from the potentiometer, resulting in more consistency when changing levels. Future work on this project would involve making the game more realistic and more like the original version. Changes would include adjusting the shape, and also including the additional game modes that were present in the original Simon Game. A real world application of the IR remote aspect of this project is that similar IR remotes are used all the time to control things like radios and televisions.

Appendix A: Code

```
#include <avr/io.h>     // for register names
#include "USART.h" // USART (serial bus) communication library

// Infrared communication libraries
#include <IRLibDecodeBase.h>
#include <IRLib_P01_NEC.h>
#include <IRLibRecv.h>

// The Mini IR Remote's button codes
#define BUTTON_RIGHT  0xfd50af
#define BUTTON_LEFT   0xfd10ef
```

```
#define BUTTON_SELECT 0xfd906f
#define BUTTON_UP     0xfda05f
#define BUTTON_DOWN   0xfdb04f
#define BUTTON_0      0xfd30cf
#define BUTTON_1      0xfd08f7
#define BUTTON_2      0xfd8877
#define BUTTON_3      0xfd48b7
#define BUTTON_4      0xfd28d7
#define BUTTON_5      0xfda857
#define BUTTON_6      0xfd6897
#define BUTTON_7      0xfd18e7
#define BUTTON_8      0xfd9867
#define BUTTON_9      0xfd58a7

IRdecodeNEC nec;          // an NEC decoder object
IRrecv rx( 2 );           // the TSOP38238 IR receiver is connected
to pin 2
uint32_t valuePrevious;   // handles NEC repeat codes

/* Global variables for game */
const int maxLength = 20; // maximum length of the pattern
int pattern[maxLength]; // holds the randomly generated pattern of
LEDs to be displayed
int idx = 0; // current index in the pattern array
int numItems = 0; // current number of items in the pattern
int ms = 0; // for note duration
bool on = false; // boolean to tell if any lights are on
int level = 1; // level corresponding to # of new notes to add to the
end of the pattern
bool repeat = false; // boolean to tell if a button was repeated by
being held down
bool won = false; // boolean set to true if player wins
bool lost = false; // boolean set to true if the player loses
int colorCount = 0; // used to determine pattern of lights to display
when the game ends

/* Arrays to hold LED locations in pad D and corresponding note
frequencies for each color:
 * Index 0: red, 1: blue, 2: yellow, 3: green */
int location[4] = { 0x40, 0x20, 0x10, 0x08 };
float freq[4] = { 880.00, 659.25, 554.37, 329.63 };
```

```
int color = -1; // integer corresponding to the color of the LED that
will light up when a button is pressed. initialized to -1 because no
button is pressed

/* Global variables use for ADC reading */
int nSamples = 0; // number of ADC measurements taken
int N = 10; // sliding window width
float omean = 0.0; // online mean ADC value
bool complete = false; // tells the main loop if ADC reading is
complete
uint16_t sensedADC; // most recent ADC measurement

/* Restart Game: reinitialize global variables. Create new pattern &
display the pattern */
void restart() {
  idx = 0;
  numItems = 0;
  won = false;
  lost = false;
  for( int i = 0; i<level; i++ ) { // number of colors to add is
based on the level
    newColor();
  }
  showPattern();
}

/* Generate random number to add to pattern of LEDs */
void newColor() {
  pattern[numItems] = rand()%4; // random int between 0 and 3
  numItems += 1;
}

/* Display a single LED & play the corresponding note */
void showColor(int color) {
    ms = 0;
    on = true;
    PIND &= 0b10000111; // turn off all LEDs
    PIND |= location[color]; // turn on the correct LED
    // set correct sound to output through piezo
    int counts = ( (16*pow(10,6))/(2*freq[color]) ) - 1;
    OCR1AH = ( counts & 0xFF00 ) >> 8; // set high byte
    OCR1AL = ( counts & 0x00FF ) ; // set low byte
}
```

```
/* Display the whole pattern of LEDs and sounds */
void showPattern() {
  char t[16];
  sprintf(t, "New Pattern length: %d\n", numItems);
  printString(t);
  for( int i = 0; i<numItems; i++ ) {
    showColor( pattern[i] );
    sprintf(t, "Show Item: %d\n", i+1);
    printString(t);
    while(TCCR1B & 0x01){;}
  }
  printString("Done\n");
}

/* Read ADC value every 1ms when enabled */
ISR( ADC_vect ) {
  nSamples++;
  sensedADC = ADCL; // must read low byte first
  sensedADC |= (ADCH & 0x03) << 8; // 10-bit precision
  omean = ( omean * float(N-1)/float(N) ) + (
float(sensedADC)/float(N) ); // online mean ADC value

  if( nSamples >= N ) {
    ADCSRA &= 0x77;   // disable ADC
    complete = true; // tell the main loop that ADC reading is done
    nSamples = 0;
  }
}

/* Interrupt triggered every 1ms. Increment counter that controls
note duration.
turn off sound and lights after 500ms. */
ISR( TIMER0_COMPA_vect ) {
  ms++;
  if( on ) {
    TCCR1B |= 0b00000001; // set TC1's prescale to 1, turns on sound
  }
  if( (ms >= 500) & (!lost) ) {
    on = false;
    TCCR1B &= 0b11111000; // set TC1's prescale to 0, turns off sound
    PORTD &= 0b10000111; // turn off all LEDs
  }
```

```
}

/* Runs the Simon Game */
int main() {

  /* Configure GPIO:
    - PD2: input, IR sensor
    - PD3: output, green LED
    - PD4: output, yellow LED
    - PD5: output, blue LED
    - PD6: output, red LED
    - PB1: output, piezo
    - PC0: input, potentiometer */
   DDRD = 0b01111000;   //set PD[6:3] to outputs for LEDs
   DDRB = 0x02; // set PB1 to output for piezo

  /* Configure Timer/Counter 0 :  1 ms interrupts to control note
duration */
  TCCR0A = 0b00000010; // set mode to CTC
  TCCR0B = 0b00000011; // prescale by 64
  OCR0A = 0b11111001; // count up to 250
  TIMSK0 = 0b00000010; // enable output compare match A

  /* Configre TC1 for piezo */
  TCCR1A = 0b01000000; // toggle on compare match
  TCCR1B = 0b00001000; // set timer to CTC mode, no prescale

  /* ADC Configuration :  collect potentiometer readings on pad A0 */
  ADMUX  = 0x40;  // right-adjusted ADC on PC[0] w/ reference to 5V
(Vcc)
  ADCSRA = 0x77;  // ADC not yet enabled, but set up for automatic
conversions and prescale = 128
  ADCSRB = 0x03;  // when enabled, ADC conversions will be
auto-triggered by TIMER0_COMPA events
  DIDR0  = 0x3F;  // disable digital input on PC[5:0] to save power

  /* Global interrupt enable */
  SREG |= 0x80;

  initUSART();

  rx.enableIRIn();  // start the receiver
```

```
  // start game by generating and displaying the first note(s) of the
pattern
  restart();

  while( true ){
    repeat = false; // reinitialize repeated button boolean to false
    color = -1; // reinitialize color to -1 (not a real color value)
    if( rx.getResults() ) {              // wait for a button press
       printByte( nec.decode() );    // decode the pulse train
       printString( "\t0x" );
       printHexByte( (nec.value & 0xFF000000) >> 24 );
       printHexByte( (nec.value & 0x00FF0000) >> 16 );
       printHexByte( (nec.value & 0x0000FF00) >> 8 );
       printHexByte(  nec.value & 0x000000FF );
       printString( "\t" );
       if( nec.value == 0xFFFFFFFF )    // check to see if it's a
repeat code
       {
          nec.value = valuePrevious;    // if it's a repeat code,
keep the previous value
          repeat = true; // tell the main loop that the button is
being held down, so don't take any action
       }
       switch( nec.value ) { // respond to the button press: choose a
behavior based on the value!
          case BUTTON_LEFT:
            printString("Button L\n");
            break;

          case BUTTON_RIGHT:
            printString("Button R\n");
            break;

          case BUTTON_SELECT: // Turn on ADC reading to change level
            ADCSRA |= 0x88; // enable ADC measurements & the ADC's
ISR
            printString("Button S: Change Level\n");
            break;

          case BUTTON_UP:
            printString("Button U\n");
            break;
```

```
case BUTTON_DOWN:
  printString("Button D\n");
  break;

case BUTTON_0: // restart game
  restart();
  printString("Button 0: RESTART\n");
  break;

case BUTTON_1: // green LED, E4
  showColor(3);
  color = 3;
  printString("Button 1: GREEN\n");
  break;

case BUTTON_2: // red LED A5
  showColor(0);
  color = 0;
  printString("Button 2: RED\n");
  break;

case BUTTON_3:
  printString("Button 3\n");
  break;

case BUTTON_4: // yellow LED, C#5
  showColor(2);
  color = 2;
  printString("Button 4: YELLOW\n");
  break;

case BUTTON_5: // blue LED, E5
  showColor(1);
  color = 1;
  printString("Button 5: BLUE\n");
  break;

case BUTTON_6:
  printString("Button 6\n");
  break;

case BUTTON_7:
  printString("Button 7\n");
```

```
            break;

          case BUTTON_8:
            printString("Button 8\n");
            break;

          case BUTTON_9:
            printString("Button 9\n");
            break;

          default:
            printString("Button ?\n");
            break;
        }

        // Change level if ADC reading is done
        if(complete) {
          float angle = (-0.27)*omean + 219; // equation based on
sensor characterization
          level = int( angle*5/180 ); // set level to one of 5
possible levels based on angle
          if(level < 1) { // bounds checking
             level = 1;
          }
          else if(level > 5) { // bounds checking
             level = 5;
          }
          char t[16];
          sprintf(t,"New level: %d of 5\n", level );
          printString(t); // print level
          complete = false; // reinitialize complete boolean to false
        }

        // Handle a color being pressed during gameplay
        if( (color > -1) & (!repeat) & (!won) & (!lost) ) {
          if( pattern[idx] != color ) { // incorrect button was
pressed
             lost = true; // set lost to true
             printString("Oh no, wrong color! You lost :( Press button
0 to replay!\n");
          }
          else if ( pattern[idx] == color ) { // correct button was
pressed, but the end of the pattern hasn't been reached
```

```
            idx += 1; // go to next color in pattern
          }
          if( idx >= (numItems) ) { // the end of the pattern was
reached. (all items were correct)
            if( numItems >= maxLength ) { //pattern array is full
              won = true;
              printString("Congratulations! You Won! :) Press button
0 to replay!\n");
            }
            else { // pattern array is not full yet
              // add new items
              for( int i = 0; i < level; i++ ) {
                if( numItems < maxLength ) {
                  newColor();
                }
              }
              idx = 0; // reinitialize tracking index to start at the
beginning
              printString("Correct!\n");
              showPattern(); // display new pattern
            }
          }
        }
        valuePrevious = nec.value;
        rx.enableIRIn();
    }

    // Display losing pattern
    if( lost & ms%500==0) { // alternate all lights & sound on and
off every 500ms
      on = false; // gameplay is off. pattern plays until restart
(button 0) is pressed
      if( (colorCount%2) == 1 ) { // turn off all LEDs and sounds
every other half second
        PORTD &= 0b10000111; // turn off all LEDs
        TCCR1B &= 0b11111000; // set TC1's prescale to 0, turns off
sound
        printString(""); // need to print empty string to give the
timer enough time to change its prescale before continuing
      }
      else { // turn on all LEDs & sounds
        PORTD |= 0b01111000; // turn on all LEDs
```

```
        TCCR1B |= 0b00000001; // set TC1's prescale to 1, turns on
sound
        printString(""); // need to print empty string to give the
timer enough time to change its prescale before continuing
      }
      colorCount++; // increment counter
    }

    // Display winning pattern
    if( won & (ms%500==0) ) { // alternate color of LED &
corresponding sound every 500ms to play a song
      showColor(colorCount%4); // display one color
      colorCount++; // increment counter
    }
  }
  return 0;
}
```