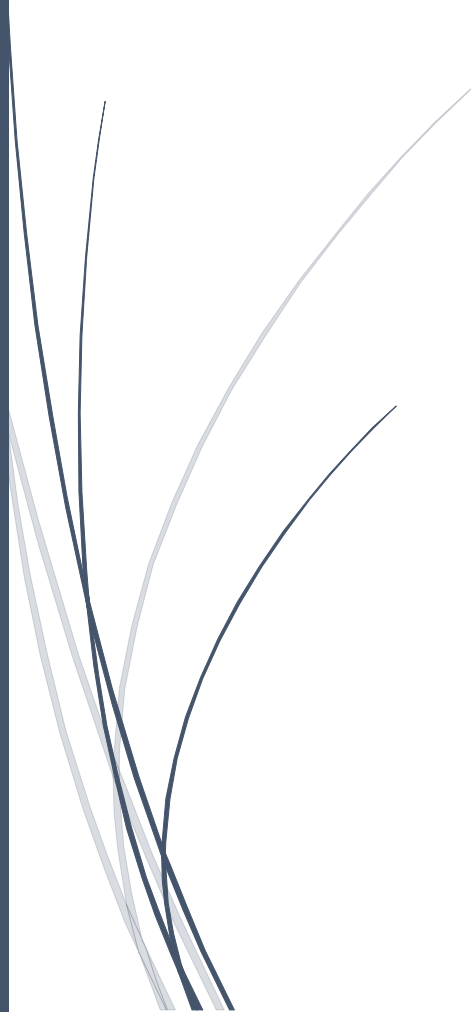
2/16/2018

Memory Box

An Arduino Project Documentation



ITX 1000

Table of Contents

1. Introduction

Arduino is an open-source platform used for building electronics projects. Arduino consists of both a physical programmable circuit board (often referred to as a microcontroller) and a piece of software, or IDE (Integrated Development Environment) that runs on your computer, used to write and upload computer code to the physical board.

The Arduino platform has become quite popular with people just starting out with electronics, and for good reason. Unlike most previous programmable circuit boards, the Arduino does not need a separate piece of hardware (called a programmer) in order to load new code onto the board – you can simply use a USB cable. Additionally, the **Arduino IDE** uses a simplified version of C++, making it easier to learn to program. Finally, Arduino provides a standard form factor that breaks out the functions of the micro-controller into a more accessible package.

2. Project Overview

The main idea behind this project was to create a simple memory game. In order to progress through the game, the user is required to observe the sequence in which different coloured LEDs blink, memorize the pattern and then repeat it, using buttons corresponding with given LED. The game lasts for several rounds (length can be configured) and with each passing round, the number of LED blinks increases by +1, starting with just one blink, up until the user makes a mistake or successfully completes all rounds. Each time the program starts, a different pattern of blinks is generated and then repeated throughout the single game.

To help a player memorize the pattern, the user is also given audio cues in form of simple sounds, each corresponding with one LED, lasting for as long as the LED flashes. During the game, different animations (sequence in which LEDs flash) are being displayed, indicating whether the game was won, lost or has just begun.

At the beginning of the game, the user can also choose one of three different difficulty levels, influencing the duration of LED blinks. The higher the difficulty level, the shorter the amount of time each LED flashes for.

The project was based on a popular TV show. It's not only fun to play, but also helps the user to develop better memory, stimulating the brain cells with much needed activity.

3. Physical Components

The project consists of several components put together to form one working digital organism. The central, most important unit - **Arduino board**, is responsible for number of functions – from processing the code, storing data to internal memory to outputting the signal. All the other components are connected to Arduino board via series of pins, either digital (1 - 13) or analogue (A0 – A5). Pins configured as output, as it is in case of our LEDs and the buzzer, can provide positive/negative current up to 40 mA of current to other devices/circuits. This is enough to brightly light up an LED or run many

sensors, but not enough to run most relays, solenoids or motors. The board also has 2x 5V pins, which were also utilized by us.

To create the Memory Box, we've used several components. The first one - **keypad membrane** - was connected to Arduino board through digital pins, numbered: 7, 8, 13 (for controlling the rows of buttons), and 2, 3, 4, 12 (for controlling the columns). The concept behind using these seemingly random pins - was to avoid employing PWM enabled pins (3, 5, 6, 9, 10, 11), which would provide additional functionality to other elements of the project.

We've used 4 different coloured **LED lights** (red, green, blue, yellow), which cannot be directly connected to a voltage source – implementing **resistors** (in this case: 220Ω) is required to limit or 'choke' the amount of current flowing through it. Unlike LEDs, resistors do not have positive or negative lead – they can be connected ether way. Too much current would cause the LED to burn out almost immediately. Each LED has two leads, one of them being positive (longer one) and the other one negative. The longer lead needs to be connected to the source of electricity (in our case: digital pins), while the negative one goes to the ground pin. The LEDs were connected to PWM pins, which allows us to create a dimming effect for certain circumstances (explained later), instead of just simple flashing.

Pulse Width Modulation, or PWM, is a technique for getting analog results with digital means. Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the portion of the time the signal spends on versus the time that the signal spends off. The duration of "on time" is called the pulse width. To get varying analog values, you change, or modulate, that pulse width. If you repeat this on-off pattern fast enough with an LED for example, the result is as if the signal is a steady voltage between 0 and 5v controlling the brightness of the LED.

The last major component used in the project is a **buzzer**. There are two types of buzzers: passive and active. The difference between the two is that an active buzzer has a built-in oscillating source, so it will generate a sound when electrified. A passive buzzer does not have such a source so it will not tweet if DC signals are used; instead, square waves of frequency between 2K and 5K need to be used. The buzzer used in our project is passive. We've used one 220Ω resistor to slightly muffle the otherwise loud buzzer.

One of the problems we've encountered during the prototyping process was the number of digital pins required to wire everything up. The keypad membrane alone uses 8 pins, for controlling each of the horizontal and diagonal rows of pins. Since our device utilizes only 8 buttons in total, we could afford having the last row disconnected, therefore making more room for other components.

All of the LEDs used in a project, the buzzer, and the resistors were soldered together with pieces of wire, making the connections permanent. Because of it, a breadboard was no longer needed in the project.

4. Explaining The Code

The source code is divided into a few smaller sections. At the very beginning, we've included one additional library (appropriately named "Keypad") enabling us to use various methods for controlling the keypad. Right below, we've defined number of pins used by LEDs, buzzer and keypad.

The next section of the code declares a few variables, necessary for the program to work properly. The constant variables, named: C, D, E, F, assign one of several integer numbers, each representing a different sound frequency used by the buzzer to play a different tune. Next, we declare the number of horizontal and vertical rows of the keypad. We also declare the names of each button in a form of two-dimensional array of characters, although this bit of code isn't really required, since it's not utilized in our project, using **ASCII character set** (representing the buttons) instead. Then, the rows of buttons are assigned to individual pins (e.g. rowPins[0] = pin 13, rowPins[1] = pin 8, etc), establishing the all-important physical connection between the keyboard and the Arduino board.

The very next line of code initializes an object of the class Keypad (called: customKeypad), which maps the keypad buttons using two-dimensional array of characters (hexaKeys[][]), keypad pins(rowPins, colPins), and number of rows/columns (ROWS, COLS). The only feature that this object is going to be used for is waitForKey() method, since we've remapped the keyboards (from ASCII standard) to use integers instead of characters anyway.

```
Keypad customKeypad = Keypad( makeKeymap(hexaKeys), rowPins, colPins, ROWS, COLS);
```

The next two lines of code are responsible for creating two arrays of integers – one (intSequence[20]) for storing random sequence of numbers (20 in total), which will impact the order of the LED's blinks. The other one (intAnswers[20]) will be used for storing the answers inputted by the user by pressing the buttons of the keypad.

The next variable, rounds, will quite importantly set the number of rounds each game is going to last for. The millisec variable, on the other hand, determines for how long each LED is going to blink. This will be controlled by the method difficultyLvl(lvl).

The method setup() is executed each time the reset button is pressed, as well as whenever the board resets for any reason, such as uploading a new sketch. Every Arduino sketch must have a setup() function, which only runs one time.

In our case, setup() method enables serial communication using band 9600, which allows us to use the serial monitor. It also declares the pins connected to LEDs and buzzer as output. The randomSeed() method initializes pseudo-random number generator, causing it to start at an arbitrary point in its random sequence. This sequence, while very long, and random, is always the same. we use analogRead(A0) as method's parameter (seed value), to make sure that generated values are different each time the method is invoked.

```
randomSeed(analogRead(0));
```

The method loop(), as opposed to setup(), runs repeatedly in loop for as long as the user continues to interact with Arduino. Every Arduino sketch requires this function.

The method beginGameAnimation() causes the LEDs to blink in a predefined sequence, specified in the method body. The for loop, for(int i = 0; i < 2; i++) ensures that code within the curly brackets runs twice, before progressing to the next statement. analogWrite(YELLOW, 255) sets the yellow LED (YELLOW == 11, which is a number of pin connected to that LED) to shine as bright as possible (255 is the maximum) for 100 milliseconds (delay(100)). This step is repeated with every LED, until all of them are powered on. From that point on, they start to turn off (analogWrite(RED, 0); 0 == no power), one by one in reversed sequence.

The next line of code, int lvl = customKeypad.waitForKey(); takes the input from a user (by pressing one of 3 keypad buttons responsible for choosing difficulty level, which is accomplished by

waitForKey() method of the object customKeypad), stores it in the lvl variable, and then passes it to the difficultyLvl(lvl) method as a parameter. Within the method body, the user input is being converted from ASCII values (e.g. button 4 == 52 ASCII) into the amount of time each LED is going to blink for, stored in the variable millisec.

The method populateRandomSequence() populates the previously created intSequence[] array with pseudo-random numbers ranging from 1 to 4. In order to generate the numbers, we utilize random(1, 5) method (1, 5 ensures that only values: 1, 2, 3 or 4 can be generated), powered by randomSeed() method, invoked in the setup section of the program.

Additionally, the populateRandomSequence() method ensures that the same LED cannot blink twice (or more) in the row. We've accomplished this by using for loop, which goes through the generated array and compares every two values stored next to each other (if statement). If the comparation proves to be *true* (Boolean value of *if* statement), it then adds +1 to the second of the two compared values. In case the two compared values are both 4, the method subtracts -2 from the second value, instead of adding +1 – otherwise we would end up with value 5, which would make the game unplayable.

The section **Main Game** starts off with a while loop, which controls the number of rounds each game is going to last for. It contains majority of the code responsible for progressing through the game. The rounds variable (declared previously) is being compared to count variable, which always starts at 1, and then gets incremented with each passing round. When count value (after number of incrementations) exceeds the value of rounds, the loop will no longer be executed.

```
while (count <= rounds)
```

The loop begins with invoking executeIntSequence(), which takes the values stored in previously generated intSequence[] and then executes ledFlasing(colour, frequency, millisec) method passing different sets of parameters, depending on it's value. For instance, if the first element of intSequence[] array equals 1, the parameters passed to method's body will be: RED (== pin number: 5) passed as colour argument, F (== sound frequency: 352), passed as frequency, and millisec (variable set by difficultyLvl() method, as explained previously). What's vitally important, this method is controlled by for loop, and the number of times ledFlashing() method is executed, depends on variable count – the very same variable used by us to control while loop. This means that, with each passing round, the program will execute one time more than in previous round. In the first round, only one LED will flash (and one corresponding tune will be played by buzzer), while in the second round, these numbers will increase by one (two flashes, two sounds), and so on.

The ledFlasing() method will use passed arguments in the following sequence:

1. delay(50) – waits for 50 milliseconds
2. analogWrite(colour, 255); - one of the LED's shining with full power (255 is the maximum value accepted by Pulse Width Modulation)
3. tone(buzzerPin, frequency, millisec); - passes 3 parameters to tone() method: number of pin connected to the buzzer, frequency of the tune played by the buzzer, the amount of time the tune it's going to last for
4. delay(millisec); - the amount of time analougeWrite() and tone() functions are going to last for (in other words: for how long the LED will blink, and the buzzer make a sound simultaneously)

The ledFlasing() method basically tells the user which buttons need to be pressed in order to progress through the game. The input from user is received through the keypad and is being stored in the intAnswers[] array, which is exactly what populateIntArray() does. It works similarly to populateIntSequence() method, as it also uses for loop controlled by count variable. The main difference is, (obviously) it takes the input from the user, instead of using generated numbers. Because the keypad uses ASCII character set, we've used the switch statement to "translate" them into 1-4 numbers and populate the intAnswers[] array with those numbers. For example, if user presses the yellow button (originally: letter "B" on the keypad), which inputs 65 (B = 65 in ASCII), that value will trigger case: 65 of the switch statement, which will assign number 4 to that particular element of intAnswers[j] array, controlled by variable j in for loop. After that, user input is **validated** (explained below, point #5) To summarize, the populateIntArray() method does the following:

1. start for loop: count variable, which controls the loop, depends on the while loop – if it's round #3, then the count == 3;
2. intAnswers[j] = customKeypad.waitForKey();
   - assigns input from the keypad to intAnswers[j] array
   - if the loop runs for the very first time, then j == 0, so the index element of that array also equals 0: intAnswers[0]
   - if the user has pressed red button (originally: 1), then 49 (1 == 49 in ASCII) is assigned to intAnswers[0]
3. delay(200); - waits 200 milliseconds before executing the next statement
4. switch(intAnswers[j]) {case 49: xxx, case 50: xxx and so on}
   - takes the ASCII value assigned to intAnswers[0] (index number [0] - if it's the first time for loop is running), transforms it to decimal value, and assigns it to intAnswers[0], replacing the previous number. E.g. 49 becomes 1
5. if (intSequence[j] != intAnswers[j])
   - checks user input stored in intAnswers[0] array, compares it against random-generated number, stored in intSequence[0] array. If the user has pressed the right coloured button (hinted by LED blink and the sound made by buzzer), then the validation ends there and the for loop can start over. If, however, the pressed button was incorrect, the Boolean value of the if statement is false (because intSequence[j] is different than (!=) intAnswers[j]), and the next two methods contained within if statement will be invoked. First one, loosingAnimation(); will make the LEDs blink in a certain sequence, indicating that the game is lost, while the next one, resetFunc() will make the program restart itself and start the execution from the very beginning
6. j variable will increment by +1, and the loop will start over. This time it's the second element of the arrays (intSequence[1], intAnswers[1]) that will be populated, validated etc. The loop will continue for as long as j <= count

The loosing animation is slightly different from the previous ones, as it uses **Pulse Width Modulation** to make the LEDs fade in and out. The animation will run twice, which is controlled by for loop. The while loop is responsible for the dimming effect itself. The starting value of j variable (controls while loop and brightness of the LEDs) is set to 0, and it will increase by +15 with every passing loop, shining for 30 milliseconds each time, which is enough to make the dimming visible. Once the value of j exceeds 255 (while j < 255), the loop terminates, and another while loop begins. It does exactly the same thing, but in reversed order. It starts off with j == 255, and then slowly decreases by -10 with each passing loop, making the LEDs shine less and less brightly. After each while loop is executed twice (again: controlled by for loop: for(int i = 0; i < 2; i++)), the animation ends.

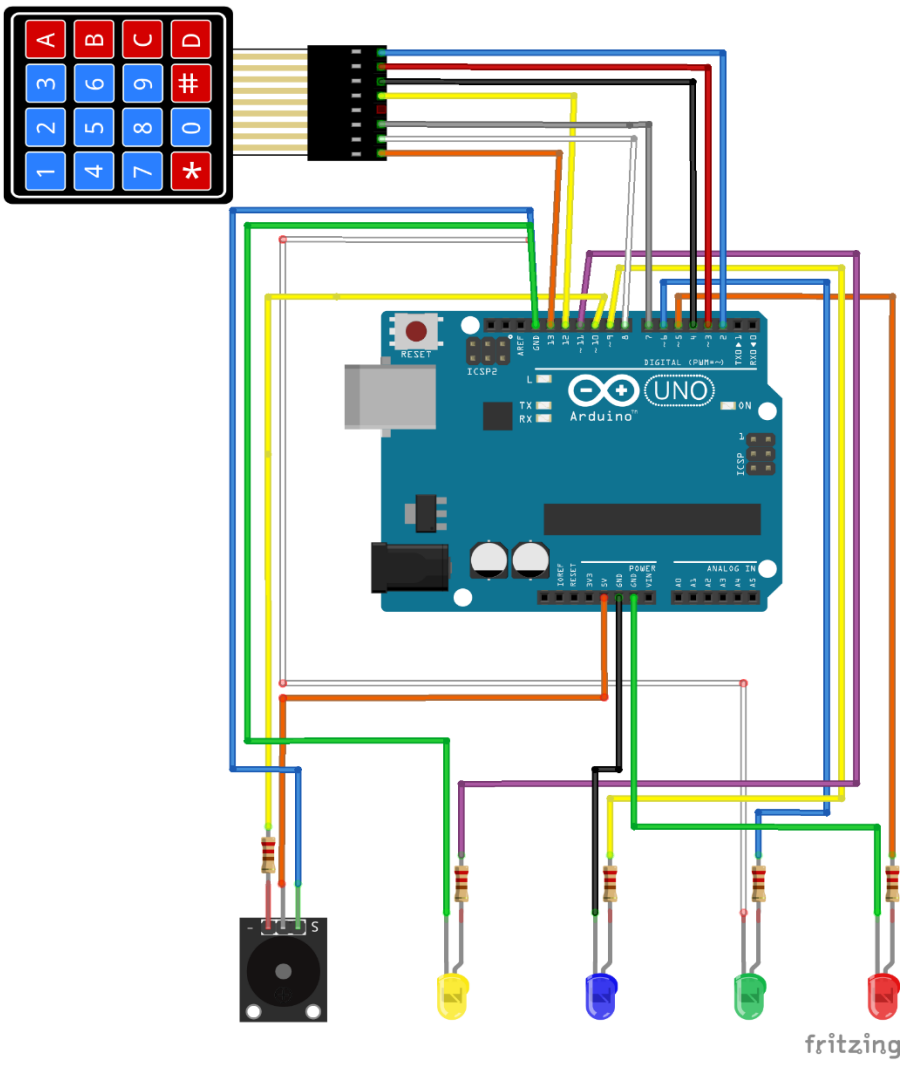
The very last method in for loop, resetFunction(), simply sets the count value to its initial value: 1 and starts the loop() method again.

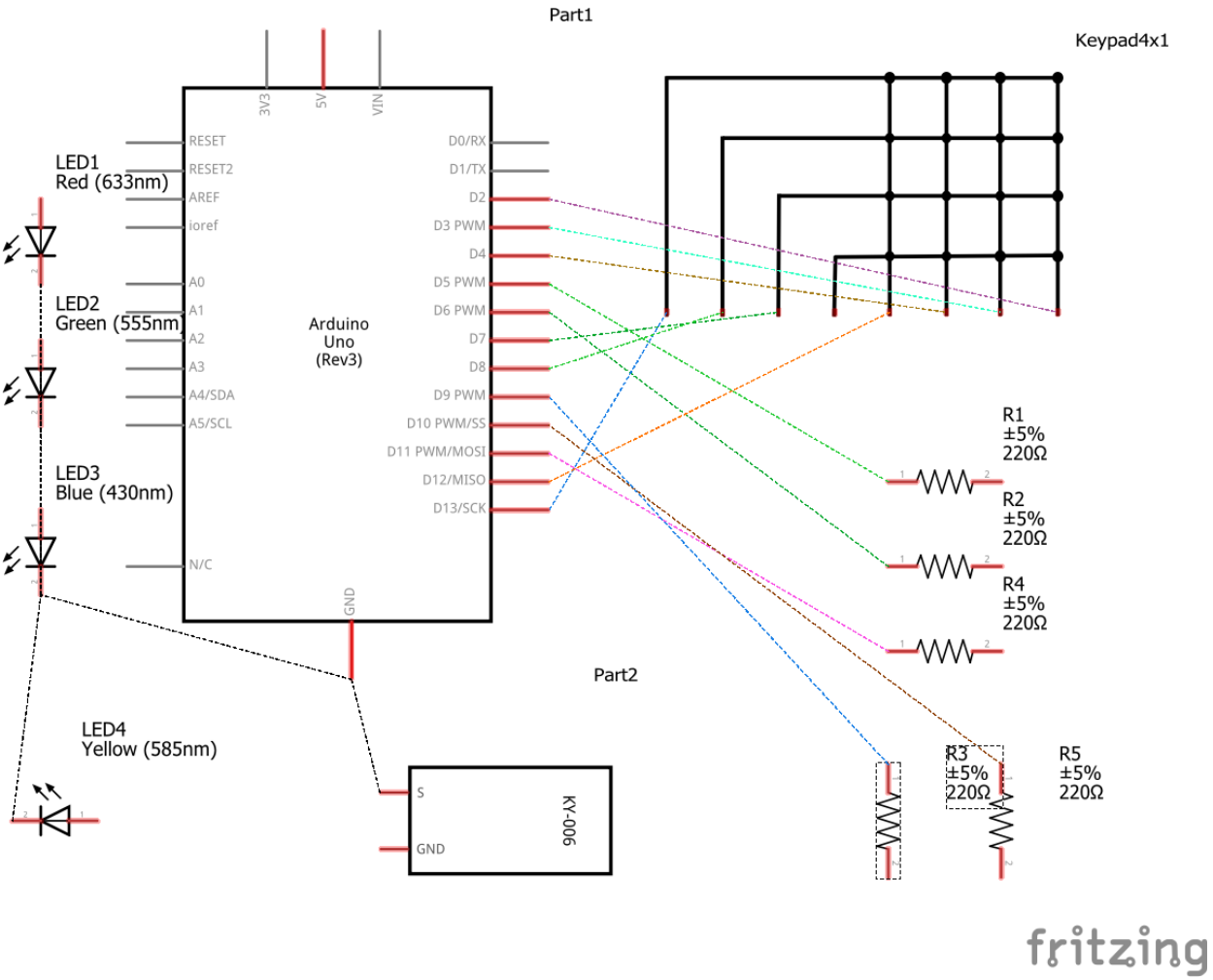
5. Conclusion

In conclusion, Arduino board is a great platform for developing variety of devices. With the wide array of available parts, it provides an inexpensive opportunity for converting your creative ideas into reality.

The Memory Box development process proved to be very intuitive and straight forward. The only major complains we have concerns the Arduino Integrated Development Environment, which we found to be quite limited in terms of debug capabilities. To address that problem, we've used a serial monitor to diagnose and fix any encountered bugs.

6. Schematics





7. The Source Code

```
// ----------------------------------------------------------------------------
// > > > > > > > > MEMORY BOX < < < < < < < <
// ----------------------------------------------------------------------------

// included libraries
#include <Keypad.h>

//============================================================================
//============================ DEFINING PINS ================================
//============================================================================

// define digital pins for using LEDs
#define RED 5
#define GREEN 6
#define BLUE 9
#define YELLOW 11

// define digital pin for controlling passive buzzer
```

```
#define buzzerPin 10

//================================================================================
//========================= D E C L A R I N G   V A R I A B L E S ================
//================================================================================

// buzzer - declare variables as sound frequencies
const int C = 264;  // Hz
const int D = 297;
const int E = 330;
const int F = 352;

//keypad - declare number of rows and columns
const byte ROWS = 4;    //four rows
const byte COLS = 4;      //four columns

//keypad - declare character names for each of the keypad's button , using two-dimensional array
//not really necessary, as the rest of my code doesn't utilize it, using ASCII standard instead
char hexaKeys[ROWS][COLS] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};
// each row is controlled by two pins simultaniously
byte rowPins[ROWS] = {13, 8, 7, };  // connect to the row pinouts of the keypad (array of pin
                                       numbers
                                       // the last row has no pin assigned to it (needed it for the buzzer)
byte colPins[COLS] = {12, 4, 3, 2}; // connect to the column pinouts of the keypad (array of pin
                                     numbers)

//initialize an instance of a class NewKeypad (library Keypad)
Keypad customKeypad = Keypad( makeKeymap(hexaKeys), rowPins, colPins, ROWS, COLS);

// create arrays of integers for storing random sequence of numbers (intSequence) and user's input
    (intAnswers)

int intSequence[20];
int intAnswers[20];

// number of rounds in single game
int rounds = 7;

int count = 1;

// LED flasing time
int millisec;

//================================================================================
//================================= S  E  T  U  P ================================
//================================================================================
```

```
void setup()
{

  Serial.begin(9600);

  pinMode(buzzerPin, OUTPUT); // declaring buzzerPin as output
  pinMode(RED, OUTPUT);       // declaring LED pin as output
  pinMode(GREEN, OUTPUT);     // declaring LED pin as output
  pinMode(BLUE, OUTPUT);      // declaring LED pin as output
  pinMode(YELLOW, OUTPUT);    // declaring LED pin as output
  randomSeed(analogRead(0)); // initializes pseudo-random number generator; an argument
                             analogRead(0) will make sure the generated values will be different
                             each time

}

//=============================================================================
//================================== L  O  O  P ===============================
//=============================================================================

void loop()
{
  // triggers 'an animation' indicating the device was just turned on or reseted, and the Arduino
  // awaits choosing difficulty level
  beginGameAnimation();

  // choosing difficulty level
  int lvl = customKeypad.waitForKey();  // the loop() is suspended until a key is pressed; the key value
                                           is stored in lvl variable

  difficultyLvl(lvl);           // lvl variable is passed to a difficultyLvl method body (4-6)

  // populates the intSequence array with pseudo-random numbers (range: 1-4)
  populateRandomSequence();


  ////////////////////////
  // M A I N   G A M E //
  ////////////////////////
while (count <= rounds)    // count always starts at 1, incremented by 1 each round; number of
                           rounds is specified in Declaring Variables section
  {
    // executes random sequence from intSequence to call a method ledFlashing with corresponding
    parameters (more detailed explanation in Methods section)
    executeIntSequence();

    // populates the array intAnswers with user input
    populateIntArray();

    // adds +1 to a counter, goes back to the beginning of while loop
```

```
   count++;

  } // while loop (main game) ends

//================================================================================
// this part of the code is reached when condition for while loop is met (becomes  true) [while (count
<= rounds)], meaning that the user succesfully finished all rounds

winningAnimation();   // LED lights will flash in a certain sequence, indicating that the user has won
                      // the game
resetFunction();      // program resets and the game starts over from the very beginning

} // (main) loop ends

//================================================================================
//============================== M  E  T  H  O  D  S ==============================
//================================================================================

// triggers a difficulty level "animation" (sequence of LEDs flashing) at the beginning of the game
void beginGameAnimation()
{
 for(int i = 0; i < 2; i++)    // i < 2 means the same sequence of LEDs flashing will be repeated twice
 {
   analogWrite(YELLOW, 255);   // YELLOW = 11, which is a number of the pin connected to yellow
                               LED; 255 - it will shine with full power...
   delay(100);                 // ...for 100 milliseconds
   analogWrite(BLUE, 255);     // same as above, only with blue LED
   delay(100);                 // and so on
   analogWrite(GREEN, 255);
   delay(100);
   analogWrite(RED, 255);
   delay(100);
   analogWrite(RED, 0);
   delay(100);
   analogWrite(GREEN, 0);
   delay(100);
   analogWrite(BLUE, 0);
   delay(100);
   analogWrite(YELLOW, 0);
   delay(100);
 }
}

// chooses difficulty level
void difficultyLvl(int level)   // parameter level is an integer number inputed by the user at the
                                beginning of each game
{
 switch(level)                 // if imputed number == [4-6], execute one of the following cases:
 {
   case 52: millisec = 200; break;   // 4 (hardest)  (52 ASCII == 4)   // millisec variable is set to 200
```

```
                                                    milliseconds - this value is then
                                                    pased to a proper method (see
                                                    above)
   case 53: millisec = 350; break;   // 5 (medium)   (53 ASCII == 5)   // and so on
   case 54: millisec = 750; break;   // 6 (easy)     (54 ASCII == 6)
   case 66: resetFunction(); break;// reset         (66 ASCII == B)   // if 'B' is pressed, the program
                                                    resets
  }
}

// populates the intSequence array with pseudo-random numbers (range: 1-4)
void populateRandomSequence()
{
   for (int i = 0; i < rounds; i++)    // for loop will be repeated for as many rounds were set up
   {
    intSequence[i] = random(1, 5);

    // ensures than no two values next to each other (in the array) are the same:
    for (int j = 0; j < rounds; j++)   // for loop will be repeated for as many rounds were set up
    {
      if(intSequence[j] == intSequence[j + 1])      // if index element j == j + 1 (e.g. intSequence[2] ==
                                                    intSequence[3]), then...
      {
       intSequence[j+1] = intSequence[j+1] + 1;     //... add +1 to element j + (e.g. 1intSequence[3])

       // if the two numbers next to each other are 4, then the latter one becomes 5 (because of if
       statements just above), which would break the game
       // when this happens -> subtract 2 from 5:
       if(intSequence[j+1] == 5)
       {
          intSequence[j+1] = intSequence[j+1] - 2;
       } // inner if
      } // outer if

    } // for ends

    Serial.println(intSequence[i]);  //debugging
   }
} // fillInRandomSequence ends

// executes random sequence from intSequence number to call a method ledFlashing with
corresponding parameters, resulting in LED flasing and buzzer playing sound
void executeIntSequence()
{
   for (int i = 0; i < count; i++)      // for loop repeated as many times as 'count' equals to (specified in
                                        while loop of the main game)
   {

     if (intSequence[i] == 1)               // if the element of random sequence (stored in intSequence[]
                                            array) == 1, then...
     {
```

```
        ledFlashing(RED, F, millisec);      // ... invoke ledFlashing method, passing the following
                                               arguments: RED (pin number connected to red LED), F
                                               (equals to 352, which is a frequency for note F), millisec
                                               (value of this variable is assigned to it when difficulty level
                                               method is invoked)
      }
      else if (intSequence[i] == 2)            // and so on
      {
        ledFlashing(GREEN, E, millisec);
      }
      else if (intSequence[i] == 3)
      {
        ledFlashing(BLUE, D, millisec);
      }
      else if (intSequence[i] == 4)
      {
        ledFlashing(YELLOW, C, millisec);
      }

    }
} // fillInSequence ends

// method for triggering appropriate LED (colour = pin number), sound frequency and LED
flash/buzzer sound time (depends on the level of difficulty)
// triggered by executeIntSequence() method (see below)
void ledFlashing(int colour, int frequency, int millisec)
{
  delay(50);
  analogWrite(colour, 255);   // causes one of the LEDs to shine with full power (analogWrite method,
                                 using PWM accepts values <= 0 (no power) and >= 255 (full power))
  tone(buzzerPin, frequency, millisec); // causes buzzer connected to digital pin 10 (buzzerPin) to play
                                           a sound of a specified frequency, for x amount of
                                           milliseconds (depends on difficulty level)
  delay(millisec);
  analogWrite(colour, 0); // after the delay (again: depends on the difficulty lvl method), turns off the
                             LED
} // ledFlasing ends

// populates the array intAnswers with user input
void populateIntArray()
{
    for (int j = 0; j < count; j++)     // as mentioned above, count starts at 1 and increments by +1 with
                                           each passing round (while loop), meaning the intAnswers array will
                                           accept one more array element each round (starting from 1 to
                                           [rounds] variable
    {
      Serial.println(intSequence[j]);   //debugging

      intAnswers[j] = customKeypad.waitForKey(); // the loop() is suspended until a key is pressed; the
                                                    key value is stored in the array intAnswers[j];
```

```
                                                // 'j' increments by +1 (j++) with each passing for
                                                loop
    delay(200);   // wait for 200 milliseconds

    // Serial.println(intAnswers[j]);  //debugging

    // assigns ASCII values to more convienient decimal numbers [e.g. 51 (ASCII) == 3 (decimal)]
    switch(intAnswers[j])
    {
      case 49: intAnswers[j] = 1; break;     // assings 1 to key button 49 (ASCII)
      case 50: intAnswers[j] = 2; break;
      case 51: intAnswers[j] = 3; break;
      case 65: intAnswers[j] = 4; break;
      case 66: resetFunction();      break;     // if 'B' is pressed, the game resets (calls for reset
                                                function)
    }

    Serial.println(intAnswers[j]);  //debugging

    // VALIDATES USER INPUT
    if (intSequence[j] != intAnswers[j])    // if user input doesn't match random sequence of flashing
                                            LEDs...
    {
      loosingAnimation();                   // ...loosing "animation" is displayed...
      resetFunction(); //call reset         // ...and the game resets
    }

   }
} // end of populateIntArray()

// if user input doesn't match random sequence of flashing LEDs, this method is invoked, causing the
LEDs to flash in order specified in this method's body:

// creates the pulsing effect (fade in/out), using Pulse Width Modulation technique
void loosingAnimation()
{
   for(int i = 0; i < 2; i++)    // i < 2 - will be repeated twice
   {
    int j = 0;
    while(j < 255)            // while loop will repeat until j is greater than 255
    {
       analogWrite(RED, j);          // red led will start with no power (int j = 0)
       analogWrite(GREEN, j);        // green led will start with no power (int j = 0)
       analogWrite(BLUE, j);         // blue led will start with no power (int j = 0)
       analogWrite(YELLOW, j);       // yellow led will start with no power (int j = 0)
       j = j + 15;                   // increments j by 15 with each while loop
       delay(30);                    // wait for 30 milliseconds to see the dimming effect
    }
    while(j > 0)              // does almost exactly the same thing as while loop above, only in opposite
                               direction (starts off LEDs shining full power, then decreases its power by 10
                               every 30 milliseconds until it reaches 0
```

```
    {
      analogWrite(RED, j);
      analogWrite(GREEN, j);
      analogWrite(BLUE, j);
      analogWrite(YELLOW, j);
      j = j - 10;
      delay(30);
    }
   }
} // loosingAnimation ends

// LED lights will flash in a sequence specified in the method's body, indicating that the user has won
the game
void winningAnimation()
{
   delay(500);              // waits for half a second after user has won the game, then starts the for
                            loop
   for(int i = 0; i < 8; i++)          // loop will be repeated 8 times
   {
      analogWrite(RED, 255);      // red LED, shining with full power...
      delay(100);                 // ... for 100 milliseconds...
      analogWrite(RED, 0);        // ... then turns off
      analogWrite(GREEN, 255);  // same as above, only with green LED
      delay(100);
      analogWrite(GREEN, 0);
      analogWrite(BLUE, 255);
      delay(100);
      analogWrite(BLUE, 0);   // and so on
      analogWrite(YELLOW, 255);
      delay(100);
      analogWrite(YELLOW, 0);
   }
} // winningAnimation ends

// resets the program
void resetFunction()
{
  count = 1;       // sets the starting value of count back to 1
  loop();          // goes back to the beginning of the loop() method
}
```

8. Changelog

- Ver 0.1 – 0.3
  - Early, unstable versions

- Ver 0.4
  - First working prototype

- Ver 0.5
  - Connected LEDs through analogue ports

- Ver 0.6
  - Implemented passive buzzer
  - Added resistor to lower the buzzer's volume

- Ver 0.7
  - Removed one of the membrane cables (pin 6)
  - Reshuffled membrane connectors to make room for LEDs (PWM pins)

- Ver 0.7.5
  - Reconnected LEDs from analogue pins to digital PWM ones (5, 6, 9, 11)

- Ver 0.8
  - Reorganized the code, created methods

- Ver 0.9
  - Added reset button
  - Added *begin game* animation
- Ver 0.9.5
  - Added more methods

- Ver 1.0
  - Prevented one LED from flashing twice in a row
  - Added dimming effect for LED *losing animation*
  - Fine-tuning the code
  - Refining documentation

- Ver 1.1
  - Further code improvements

- Ver 1.2
  - Rewritten reset function