Arduino Intermediate Activity

Authors: Chris Rennick, Fareed Rasheed, Hailey Doleweerd, Orion Bruckman, Nancy Nelson

Contents

1 Microcontrollers

Microcontrollers are small one-chip computers that are at the core of much of today's technology to automate and control tasks. Generally, they have a Central Processing Unit (CPU), some memory, and programmable input and output ports. This intermediate workshop builds on prior knowledge of microcontrollers and introduces polling loops, interrupts, and timers. With polling loops, our code manually checks the status of any input pins at a regular interval, and then we can respond. Interrupts are used for more critical events so that the code can respond more quickly.

These concepts will be used to create a simple, 2-player version of the game Pong.

1.1 Required Materials

- 1x Arduino UNO Rev. 4
- 1x OLED Display
- 1x Breadboard
- 3x Pushbuttons
- 12x Wires
- 1x USB-C Cable

2 Activity –Pong Overview

Pong is a simple game, however the logic used is fundamental to creating interactable devices that respond to the real world. This activity aims to help you grasp the principles of digital logic and interactable devices by introducing concepts such as button polling, timers and interrupts. These principles are the core of the systems we interact with every day ranging from handheld devices to complex game engines.

2.1 Build your Circuit

Follow Figure 1 below to connect the OLED display and the 3 pushbuttons to the Arduino. Ensure the positive rail on the breadboard that is connected to the OLED display is connected to the 5V pin on the Arduino, not the 3.3V. The pushbutton that will be the game pause button must remain connected to the Digital0, Digital1, Digital2, or Digital3 pins on the Arduino Uno as the pause function will be implemented using an interrupt. These pins are the only digital pins that can be used for external interrupts on the Arduino.

If your kit included a shield, attach the shield onto the Arduino and wire the buttons as shown. Otherwise, if your kit included an individual screen, wire the buttons and the screen as shown.

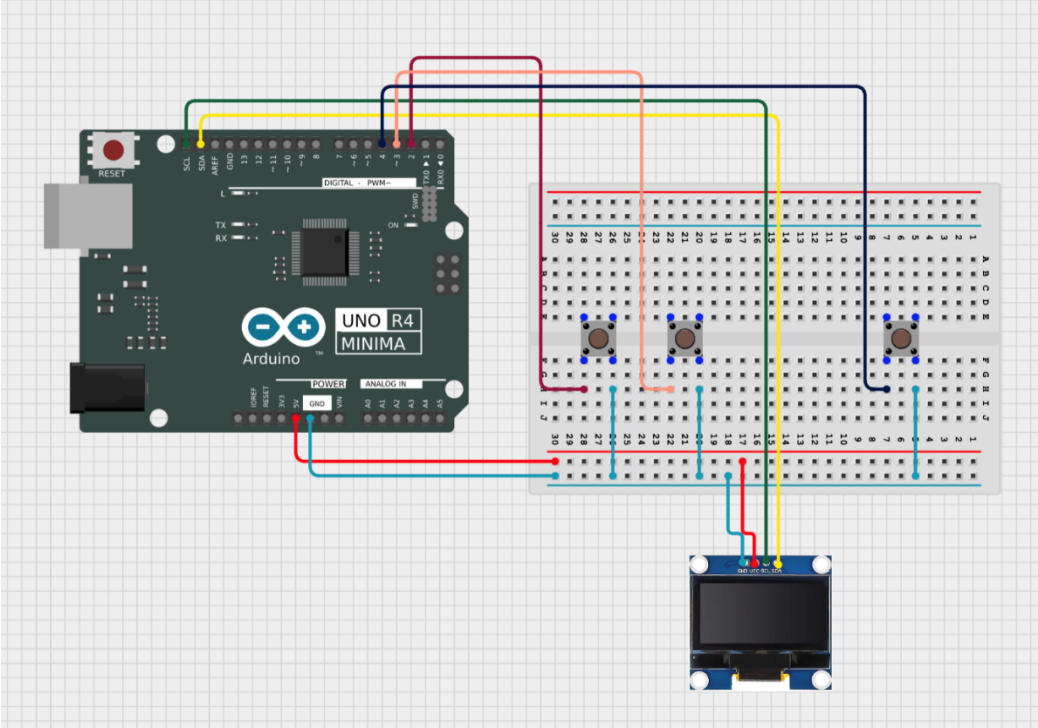

Figure 1 – Circuit diagram for Pong game

2.2 Skeleton Code Checklist

You'll build on some skeleton code that provides the framework for Pong. Most of the logic was pulled from the code and its up to you to fill it in using the tutorials and code blocks from this document.

You can find the skeleton code in the same LEARN folder as this document.

Here's a checklist to help you keep track of your progress through the skeleton. This is the bare minimum to create a functional game but there's lots more that can be added

- Declarations
  - Declare all the variables and libraries at the start of the program. This includes the variables in section 3.1-I2C and the button pin declarations according to your circuits
- setup()
  - Complete the screen configuration according to section 3.1-I2C
  - Declare the pinModes of the buttons
  - Attach the interrupt according to section 3.3
- pause()
  - Complete the logic to switch between the play and pause states
  - Find out how to pause the game clock
- loop()
  - Handle the paddle direction switches according to button presses according to section 3.2 Button polling
  - Handle the paddle constraints, restricting the paddle in the screen according to section 3.2
  - Ball Movement according to section 3.5

        - Ball collision with paddles according to section 3.5
        - Ball collision with screen according to section 3.5
        - Handle scoring, resetting rounds and game over conditions according to section 3.5
        - Draw the screen according to section 3.1 I2C
        - Game clock
        - Update the state of buttons on previous loop according to section 3.2
    - resetRound()
        - Fill in starting locations of balls and paddles according when resetting rounds after scoring

3 Coding the Game

Now you're ready to create the project (aka sketch) and write the code to control your system. Follow these steps:

1. Download the skeleton code from Learn
2. Start the Arduino Integrated Development Environment (IDE)
3. Go to **File** > **Open** to open the skeleton code. The skeleton code is missing most of its functionality. Use this document to fill it in.

3.1 I²C Communication

I²C is an important protocol for serial communication between devices. It is often used to connect peripherals like sensors or displays to a central controller system. In this case, I²C communication will be used to connect the Arduino microcontroller to the OLED display. I²C uses two wires to enable this synchronous transmission of data, the first being SCL which is the Serial Clock which keeps the two systems synchronized, and SDL which is Serial Data and is used for the serial transmission of data. The Arduino UNO has 2 dedicated I²C pins labelled SCL and SDA that can be used.

Here's the code to display the player paddles, scores, and ball for the Pong game. Copy and paste this into your project, then compile and upload the program. Sections in orange should already be included in the skeleton project.

You may need to install some libraries to interface with the screen. In the Arduino Library manager, search Adafruit GFX and Adafruit SSD1306. Make sure to also install all dependencies.

You'll know you're successful when you see the scores, paddles, and a ball on the screen.

This code can be pasted into a new sketch. Once you understand the process of writing to the screen, you can add this code to the corresponding sections of the skeletonto add display functionality.

```
//Defining screen
//Screen Variables
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
#define OLED_RESET -1
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

//Score tracking
int scoreP1 = 0;
int scoreP2 = 0;

const int PADDLE_WIDTH = 2;
const int PADDLE_HEIGHT = 12;

//Position tracking for player 1 paddle
int P1_X =0;
int P1_Y = 32;
//Position tracking for player 2 paddle
int P2_X = 128-PADDLE_WIDTH;
int P2_Y = 32;
//Position tracking for ball
int ballX = 64;
int ballY = 32;

void setup ()
{
//Copy screen setup code from section 3.1 - I2C Communication
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
  display.clearDisplay();
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
       }
void loop(){

//FILL IN with screen code from section 3.1 - I2C Communication
    //Display must first be cleared
    display.clearDisplay();

    // Sets up text for the player scores
    display.setTextSize(1);
    display.setCursor(40, 0);
    display.print("P1:");
    display.print(scoreP1);
    display.print("  P2:");
    display.print(scoreP2);
    display.setCursor(40,10);
    display.print(gameTime);

    //Set up paddles
    display.fillRect(P1_X, P1_Y, PADDLE_WIDTH, PADDLE_HEIGHT, SSD1306_WHITE);
    display.fillRect(P2_X, P2_Y, PADDLE_WIDTH, PADDLE_HEIGHT, SSD1306_WHITE);
    //Set up ball
    display.fillRect(ballX, ballY, 2, 2, SSD1306_WHITE);

    display.display();
    //Displays everything
    delay(15);
}
```

3.1.1 Challenge

Add the code necessary to display a game time in the format: “MM:SS” where MM are the number of minutes and SS are the number of seconds. For now, just display “00:00”. We will set up a timer to include the game time later in the workshop.

3.2 Button Polling

Button polling is the primary method of user input we will use for Pong. Suppose we want to determine when a button is pressed. Wire one end of the button to ground and the other to a digital pin as in Figure 1.

To test that the button is working, add the code below to your existing project. This code will turn the on-board LED on when the button is pressed.

This code can be pasted into a new sketch to demonstrate button polling. Once you understand the process, you can add polling functionality to the skeleton.

```
//This code will turn off the built in LED when the button is pressed
void setup(){
	pinMode(BTN_P1, INPUT_PULLUP);
	pinMode(LED_BUILTIN, OUTPUT);
	//Declares our button1Pin as an input
	// INPUT_PULLUP will by default set our pin to HIGH and will be set LOW when the button is pressed
}
void loop(){
// Previous code for OLED screen

	bool btnP1 = digitalRead(BTN_P1);
	//digitalRead() takes in the button input
	//LED_BUILT_IN is the LED built into every Arduino Board. It can also be accessed by digitalPin 13
	if(btnP1 == LOW && lastBtnP1 == HIGH){
		digitalWrite(LED_BUILTIN,HIGH);
	}else {
		digitalWrite(LED_BUILTIN,LOW);
}
lastBtnP1 = btnP1;
}
```

This code will compare the state of the button to the state on the previous loop. This is called edge detection and will detect change. The button is default high thanks to the pullup resistor and therefore will be read LOW when pressed.

3.2.1 Challenge

Add code to your project to read the state of button 2. Test that you can control the on-board LED with button 2 as well. After you complete this challenge, comment the code for writing to LED_BUILTIN based on the buttons.

3.3 Paddle Directions

In Pong, we will be using 3 buttons. Buttons 1 and 2 will be used to control the paddles of P1 and P2. When designing your game, read the button states to determine the direction of the paddles. Pressing button1 should flip the direction of player 1's paddles while pressing button2 should flip the direction of player 2's paddles.

The code below is what the logic for flipping the paddle direction might look like, but it is incomplete. Think about where the paddle direction flags should be declared.

Take a look at the skeleton code and paste the sections of this block you need into the skeleton.

```
//If the button states for either player change from Low to HIGH, the paddle
direction is flipped //p1 and p2Dir need to be declared as const int before
your setup. Assign them a value to determine their speed.


//If the state of the player1 button is high, the direction of the //paddle is
flipped. lastBtnP1 tracks the state of the button on the previous //loop of the
code. This is called edge detection and detects the change of the //button
state.
if (lastBtnP1 == HIGH && btnP1 == LOW){
      //If the paddle is not moving at the bottom, set dir +ve. Else if paddle
is not moving, set dir -ve.
      if (p1Dir == 0) p1Dir = (P1_Y <= 0) ? 1 : -1;
      //if direction is not 0, flip p1Dir
      else p1Dir *= -1;
    }
}
 // Paddle movement
P1_Y += p1Dir; // This is what actually updates the paddle location
//Above is the paddle movement for paddle1. Repeat the above code with the
corresponding variable names for paddle2.
```

How do we stop the paddles from extending past the screen borders? We need to compare the borders of the paddles with the size of the screen. If the paddles reach the edge, the directions must be set to zero(not moving).

This can be copied into the corresponding section of the skeleton code to add paddle constraint functionality.

```
//FILL IN paddle constraints from section 3.2 - Button Polling code block 3
if (P1_Y <= 0 || P1_Y >= SCREEN_HEIGHT - PADDLE_HEIGHT) {
      //constrain prevents the paddle from extending beond the screen
      P1_Y = constrain(P1_Y, 0, SCREEN_HEIGHT - PADDLE_HEIGHT);
      p1Dir = 0;
   }
```

3.3.1 Challenge

Recreate the above code for the second paddle.

3.4 Interrupts

Interrupts are methods of pausing code that is currently running to execute a short function before returning to the main code in the loop() section. If an interrupt is triggered, the original code will not resume until the interrupt is completed. Timer overflows are a type of interrupt. Interrupts constantly monitor the state of the pin they are attached to and will activate immediately when their condition is met.

Suppose we want to trigger an interrupt upon pressing a button. As soon as the pin the button is attached to changes, we want to trigger a function called `buttonInterrupt()`.

This code can be pasted into a new sketch to demonstrate how button interrupts work. After you understand how they work, you can use this method to add interrupt functionality to the skeleton.

```
//This code will wait for a button press to turn the built in LED off for a short duration

int BTN_PAUSE = 2;

void buttonInterrupt(){

//This is the code that will be executed after triggering the interrupt
	digitalWrite(LED_BUILTIN,LOW);

}

void setup(){

	pinMode(LED_BUILTIN,OUTPUT);
	pinMode(BTN_PAUSE, INPUT_PULLUP);

	//INPUT_PULLDOWN will by default set the pin low when the button is not pressed, and high when it is

	attachInterrupt(digitalPinToInterrupt(BTN_PAUSE),buttonInterrupt,CHANGE);

	/*attachInterrupt takes 3 values; digitalPinToInterrupt(pin) declares which pin to attach the interupt. The 2nd value is the interrupt function which must return void. The final value is the interrupt configuration. CHANGE will trigger the interrupt when the interruptPin changes state. FALLING will trigger when the pin changes from 1 to 0 – RISING will trigger when the state changes from 0 to 1.*/

}
void loop(){
	digitalWrite(LED_BUILTIN,HIGH);

	delay(1000);

}
```

After you've confirmed the interrupt works and turns off the LED temporarily, remove the `delay(1000)` statement from `loop()`.

When creating the function buttonInterrupt() or any interrupt function, do not use loops or delays. The interrupt must be short and quick so as to return to the main loop of the code as quickly as possible. In Pong, we will be using an interrupt to create a pause button. The pause button will interrupt the main game loop and enter the game into a pause state, where it will wait for a second button press to continue back to the main game.

3.5 Time Functions

The Arduino R4 has built in real time 32-bit clocks. These clocks are used to provide PWM functionality, precise time measurements and delay functionality. On a low level (meaning no abstraction), these clocks operate on 128Hz. After a certain amount of clock cycles, the timer will "overflow" meaning its maxed out its 32-bit capacity, and trigger an overflow event. This overflow event can be used to track

intervals of time or create time sensitive functionality. Manipulating the timer chips requires accessing the hardware control registers and bits.

The Arduino IDE thankfully abstracts all of this behind various built-in libraries and functions. In order to add our game clock, we will be using the function `millis()`. `millis()` will return the amount of milliseconds that have passed since the MCU was activated. On a low level, the timers are configured to overflow every millisecond, `millis()` will return the number of overflows that have occurred.

The Arduino IDE also includes many other time functions. `delay()` uses the hardware timers to precisely pause the CPU and `micros()` will return the microseconds since the MCU was activated. `analogWrite()` and PWM signal generation also uses the hardware timers but have abstracted them away behind functions and libraries.

```
gameTime = millis();
```

3.5.1 Challenge

1. The code above will return the time since the Arduino powered on, in milliseconds. We would like you to display the game time in the format MM:SS. You will need to write the code to calculate the minutes and seconds, and then modify your code from 3.3.1 to display the actual time.
2. Find a way to pause the game clock when the game is paused. Remember that the game is paused using an interrupt. This can be accomplished by subtracting the time spent in the paused state from the true time.

3.6 Game Logic

The game logic for Pong can be broken down into several components.

Before starting the game, wait for a player input. After receiving input wait for input from player 2. After both players are ready, the game can enter the playing state and the game begins.

Ball movement and positioning is broken down into X and Y components. By storing a separate variable for the speed of the ball in the X and Y, movement can be simulated by adding the velocity to the position every loop of the code. When the ball collides with the borders or paddles, the ball velocity is flipped.

Collision is determined by comparing the position of the ball with the position of the paddles and borders. Ball position overlaps, X or Y velocity correspondingly flips.

This code can be pasted into corresponding sections of the skeleton code to add collusion and movement functionality.

```
      //FILL IN as defined in section 3.6 code block 1
      ballX += ballVX;
      ballY += ballVY;
      //This is the movement for the ball. Similar to the paddles,
every loop of the code will update the position


  // Paddle collision
    //This will flip the direction of the ball on contact with a
paddle. It takes into account the width of the paddle when
creating hitboxes.
    //Remember, P1_Y and P2_Y represent the bottom of the paddles
and the collision zones must be "extended" by the paddle height.
    if (ballX <= P1_X + PADDLE_WIDTH &&
        ballY >= P1_Y && ballY <= P1_Y + PADDLE_HEIGHT) {
      ballVX *= -1;
      ballX = P1_X + PADDLE_WIDTH + 1;
    }
   if (ballX >= P2_X - 1 &&
        ballY >= P2_Y && ballY <= P2_Y + PADDLE_HEIGHT) {
      ballVX *= -1;
      ballX = P2_X - 2;
    }
//This is collision for the balls and paddles. Fill in collision
for the top and bottom of the screen. If the ball reaches the
```

3.6.1 Challenge

Currently, there is no logic for the ball colliding with the top and bottom walls of the game. Add logic to reverse the vertical direction of the ball when would go out of bounds vertically.

This code can be added to the corresponding section of the skeleton to add collision functionality with the tpo and bottom of the screen.

```
if (ballY <= 0 || ballY >= SCREEN_HEIGHT - 2) {

//This is the bounds of top and bottom the screen. What happens when the ball hits the bounds?

}
```

Scoring is achieved by checking if the ball has contacted either side of the border, without contacting the paddles. Recreate the below code for a player 2 score as well.

This code can be added to the corresponding section of the skeleton to add scoring functionality

```
if ballX < 0) {

      scoreP2++;

      resetRound();

}

if (ballX > SCREEN_WIDTH) {

      scoreP1++;

      resetRound();

}

//This is the scoring for one side of the screen. A similar code block must be written for the opposite side, change the variables!
```

3.6.2 Challenge

Complete the `resetRound` function so that it resets the position of the ball when it is called.

3.6.3 Challenge

Decide what value constitutes a winning score. Change your game so that after a player reaches that score, a “game over” is triggered. Include some way of resetting the game so the players can play again.

4 Challenge

4.1 Make It Your Own!

Using the tools you've learned, create a working game of Pong. Use button polling to read user inputs and control the direction and movement of paddles and interrupts to control pause functionality or any other functions. Try to implement a timer and game clock. Find the skeleton for the functions and code you need to get started on LEARN.

- Experiment with the game. Change the size of the ball or paddles
- Have ready screens for each player
- Change the speed of the ball or paddles
- Change the scale of the time