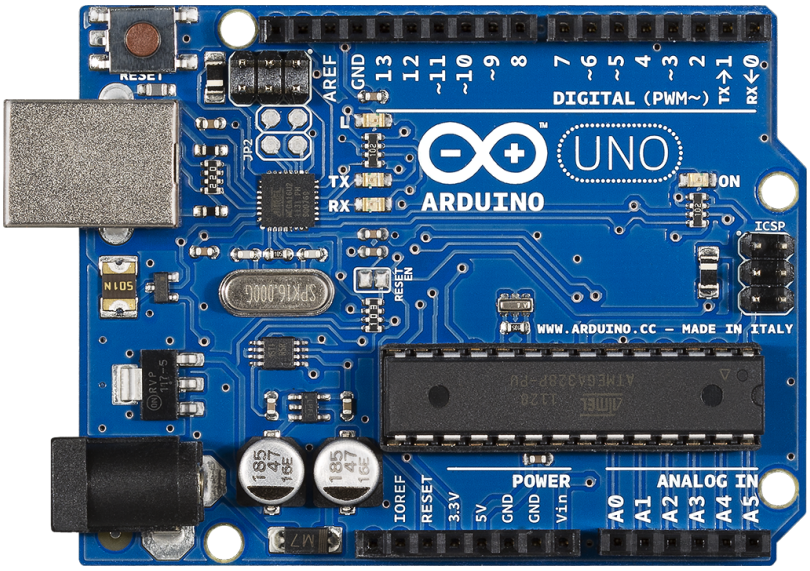
Introduction to Intelligent Mobile Systems Lab I



Lab Manual

Fall Semester

Instructors:
Kaustubh Pathak
Fangning Hu

JACOBS
UNIVERSITY

Contents

Chapter 1

Lab 1

1.1 Objective

In this lab, we are going to become familiar with how to program our Arduino step by step as well as how to use the Multimeter.

Note: After doing each task or sub-task, call the Instructor/TA to show them your result. You will be graded right away.

1.2 Getting Started on Arduino

1.2.1 Get Familiar with the Arduino Software

The Integrated Development Environment (IDE) of Arduino is free and open source. It is already installed on our lab computers. Please double click the Arduino icon on the desktop. A screenshot of the Arduino software is shown in Fig. 1.1.

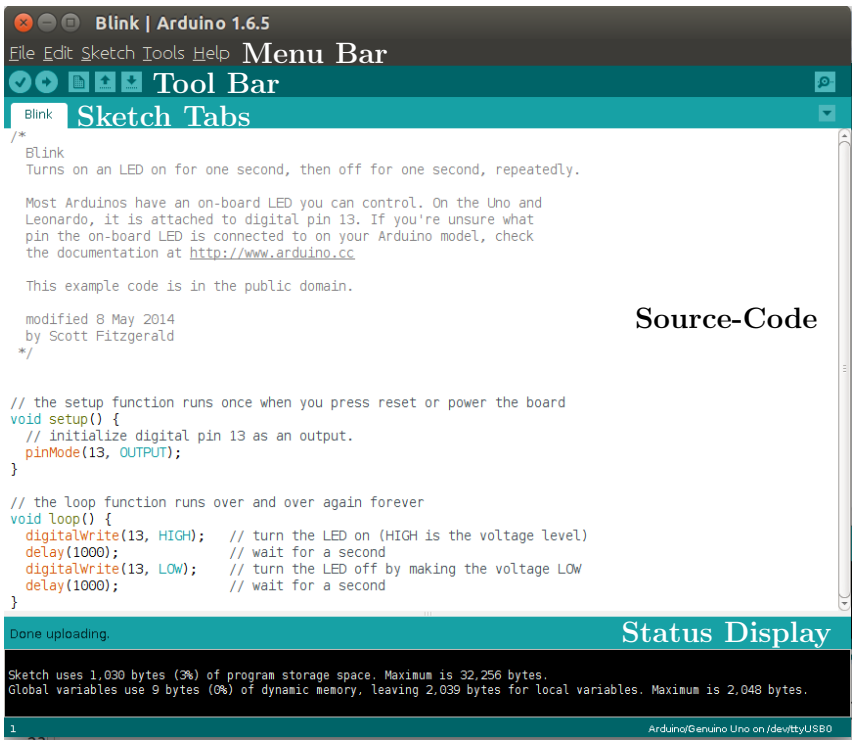


Figure 1.1: The Arduino IDE Window.

In the Tool Bar, you can find the most useful buttons. The button-descriptions can be found in the Table 1.1.

Button	Description
	Compile and check errors
	Upload codes to Arduino
	Create a new sketch
	Open an existing sketch
	Save the current sketch
	Open the serial monitor

Table 1.1: Description of the IDE Buttons

Try and become familiar with **New**, **Open**, **Save** in the Tool Bar. Open an existing file "Blink.ino" from the menu **File** $\longrightarrow$ **Examples** $\longrightarrow$ **01.Basic** $\longrightarrow$ **Blink**. If you don't find where to open it, you can simply type in the code in Listing 1.1 in the code space and save it.

```
void setup() { // Called once to initialize
  pinMode(13, OUTPUT);     // Initialize pin 13 for
                           // digital-write
}

void loop() { // Called repeatedly
  digitalWrite(13, HIGH); // Set pin 13 to 5V
  delay(1000);            // Wait 1 sec = 1000 millisec.
  digitalWrite(13, LOW);  // Set pin 13 to 0V
  delay(1000);
}
```

Listing 1.1: Blinking the Light Emitting Diode (LED).

Warning: If you cut and paste code from a listing (e.g. from Listing 1.1) in this manual to the Arduino IDE, some unnecessary spaces may be introduced: e.g. "2L" could possibly become "2 L" and as a result your code does not compile. Carefully check and correct such cut and paste errors.

1.2.2 Arduino code basics: digitalWrite

The Arduino uses a slightly simplified C/C++ programming language. There are two mandatory functions: the setup() function and the loop() function.

The setup() function executes only once initially, when Arduino first powers on or right after someone presses the reset button, causing the currently loaded program to restart from the beginning. Usually, in the setup() function we configure the pins of Arduino or start communication protocols. In the above example, the Pin13 on Arduino is configured to be an output pin by function pinMode(13, OUTPUT).

The loop() function runs repeatedly until Arduino is switched off. In the above example, digitalWrite(13, HIGH) function sets pin 13 to HIGH(5V) and digitalWrite(13, LOW) sets pin 13 to LOW(0V). The Pin13 will keep HIGH or LOW for 1000 milliseconds by function delay(1000).

Now you can compile the above sketch and check whether there are errors by clicking on the Compile Button in the Tool Bar. The error messages will be presented in red in the Status Display window. In order to run this code on Arduino, we need to upload this code onto Arduino board.

1.2.3 Upload the sketch to Arduino

Arduino can communicate to our computer simply by a USB cable. Now use the USB cable to connect your Arduino to the computer. Arduino will automatically power on after connection. The driver is already installed on our computer. If everything connects properly, a green LED light (power-on light) on Arduino will turn on.

Check the menu-item **Tools** $\longrightarrow$ **Serial Port** to configure the correct serial port on which the Arduino is connected to the computer. Usually, the Arduino is installed on serial port COM3 on Windows, but not always. On Linux, this is usually /dev/ttyUSB0. You can find out the correct port by checking the Windows **Control Panel - Device Manager**.

Now click the "Upload" Button, if everything was connected correctly, a yellow (or blue, depending on the board) LED light on the Arduino board will blink every 1 second. This on-board LED is connected to Pin13.

Task 1.1 *Show the working example to the instructor/TA.*

1.3 Build up a simple circuit

Now we can use the Arduino pins to control electronic devices. Before that, we will first build a circuit on a breadboard and then connect the Arduino pins to our circuit. The first experiment is to make a LED blink as before, but now we want to use our own LED on the breadboard rather than the on-board LED.

1.3.1 The Breadboard and Jumper Wires

A breadboard is very useful to create temporary prototypes of a circuit design and to do experiments on it since it does not require soldering. In Fig. 1.2, the breadboard used in our experiments is shown.

One can also connect two points by connecting a jumper wire between these two points.

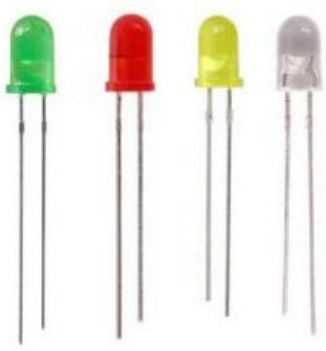
Warning: Never connect an Arduino OUTPUT pin directly to the GND.

To restrict current to $\leq 20mA$, always put a resistor $R \geq 250\Omega$ in between.

1.3.2 LED and Resistors: digitalWrite

An LED (Light Emitting Diode) is a two-lead semiconductor light source which emits light when a suitable voltage is applied to the leads. The current can only flow in one direction, i.e., from the long lead (+) to the short lead (-). BE CAREFUL of the positive and negative lead of the LED when you build a circuit.

Warning: Never connect an LED to Arduino pins directly without a resistor in between.



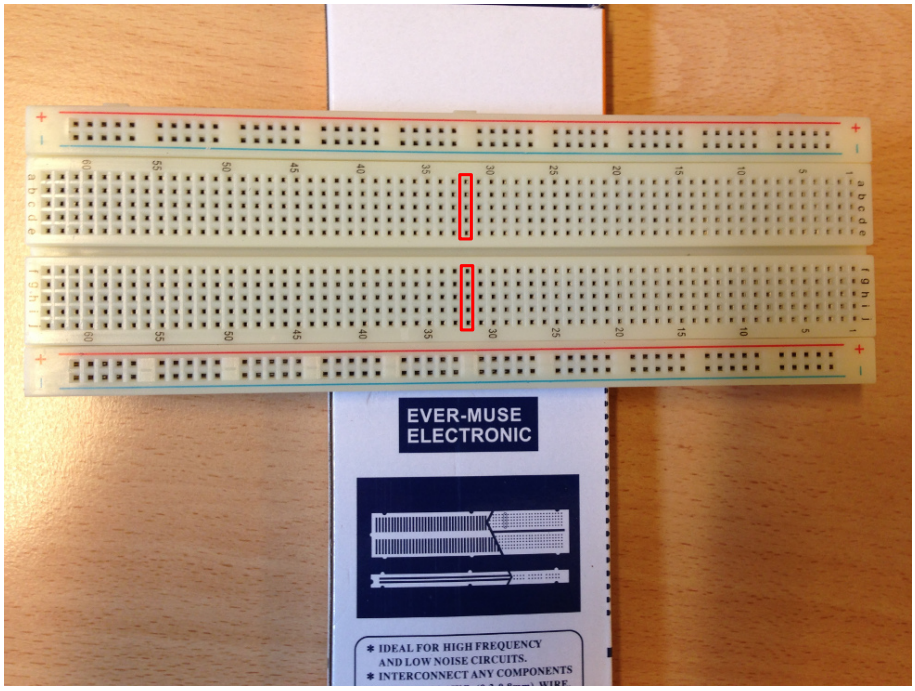


Figure 1.2: The breadboard used in the experiments. If your breadboard looks different, ask for the standard one. The top and bottom two rows are connected internally, as shown by the red and blue lines. In the middle, as shown by red rectangles, the top 5 vertical holes of each column are connected to each other. Similarly, the bottom 5 vertical holes of each column are connected to each other.

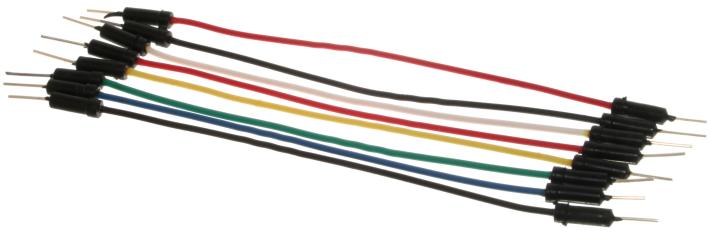


Figure 1.3: Jumper cables.

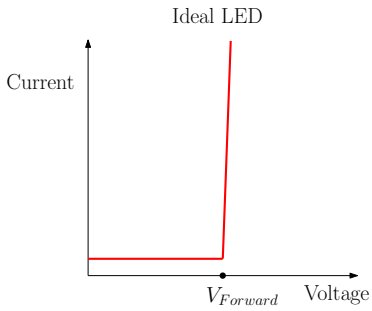


Figure 1.4: The ideal characteristics of an LED.

LED	White	Red	Yellow	Green	Blue
$V_{Forward}$	$3.3V$	$2.1V$	$2.2V$	$3.7V$	$3.1V$
Resistance	100Ω	200Ω	200Ω	100Ω	100Ω

Table 1.2: Recommended resistor values.

Different colors of LEDs have different forward-voltages (refer to Fig. 1.4). With a fixed voltage source (Arduino offers a $V_{source} = 5V$ voltage source), one can use different resistors to modify the voltage across the LED and hence keep the current flowing through it within safe limits: Please also refer to the presentation slides of this week.

Ohm's law states that the voltage (V) across a resistor is the product of the current and the resistance.

$$V_R = I \cdot R \tag{1.1}$$

By serially connecting a resistor and the LED (Figs. 1.6 and 1.5), the voltage across the resistor is V_R is:

$$V_R = V_{source} - V_{Forward} \cdot$$

$V_{Forward}$ is the forward voltage of the LED (see Fig. 1.4) and V_{source} from an Arduino output pin is typically 5 V.

To keep the current from the output pin of the Arduino as well as through the LED within safe limits, we should put a resistor R in series with the LED such that

$$I = \frac{V_R}{R} = \frac{5 - V_{Forward}}{R} \leq 20 \times 10^{-3} A. \tag{1.2}$$

Table 1.2 lists $V_{Forward}$ for LEDs with different colors as well as a corresponding resistance R which satisfies (1.2).

The schematic of the circuit is as in Fig. 1.6.

Task 1.2 *Proceed as follows:*

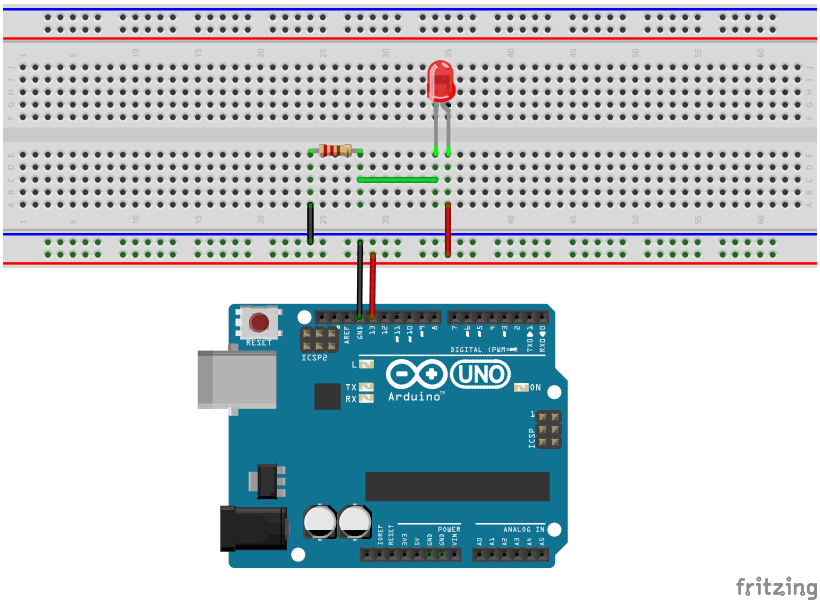


Figure 1.5: The Blink-LED Circuit with the Arduino Board.

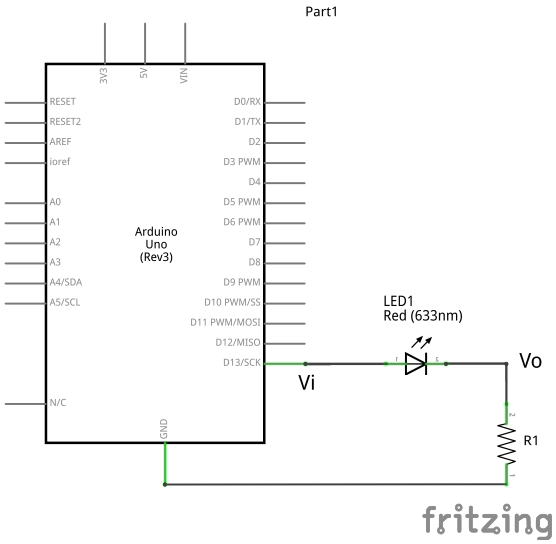


Figure 1.6: The Schematic of the Blink-LED Circuit

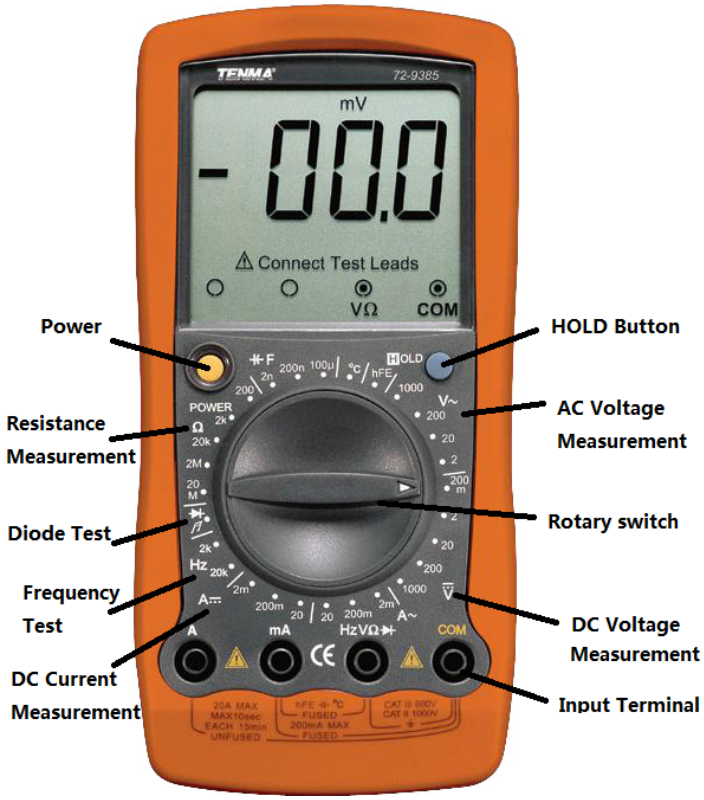


Figure 1.7: The Interface of the Multimeter.

1. *After uploading the code in Sec. 1.2.2, disconnect Arduino from the computer by pulling out the USB cable from the Arduino.*

2. *Pick an LED and the corresponding resistor from Table 1.2, serial connect them on the breadboard as shown in Figs. 1.5 and 1.6. To close the circuit, connect one end to the GND pin and another end to the Pin* 13 *of Arduino.*

3. *Get your circuit checked by the TAs or the instructors.*

4. *Now plug-in the USB cable to Arduinio. The LED should light-up every second.*

1.4 Using the Multimeter

The Multimeter is a useful instrument to measure the basic electrical properties and troubleshoot circuit problems. Figure 1.7 shows an overall display of our Multimeter.

Reminder: Please switch off the multimeter when not in use.

1.4.1 Resistance Measurement

Warning: To avoid damage to the Meter or to the devices under test, disconnect circuit power and discharge all the high-voltage capacitors before measuring resistance. The resistance ranges are: $200\Omega, 2k\Omega, 20k\Omega, 2M\Omega, 20M\Omega$.

To measure the resistance, follow the following general procedure:

- Insert the red test lead into the HzVΩ terminal of the multimeter and the black test lead into the COM input terminal.

- Set the rotary switch to an appropriate measurement position in Ω, i.e. in the sector marked "Resistance Measurement" in Fig. 1.7.
- The resistors for which you are measuring the resistance should not be connected to any circuit. You can connect the two ends of the resistor to the multimeter using alligator clips to achieve a stable connection.
- If the resistance is higher than the selected-range or in open circuit condition, the multimeter displays "1". You need to select a higher range in order to obtain a correct reading.

To obtain an accurate reading, note also the following points:

- The test leads can add 0.1Ω to 0.2Ω of error to the resistance measurement. To obtain accurate readings in low-resistance, short-circuit the input terminals beforehand and record the reading obtained (called this reading as X). (X) is the additional resistance from the test lead. Then use the equation: measured resistance value (Y) - (X) = accurate readings of resistance.
- If the input terminal short-circuit reading ≥ 0.5, check the test leads for any looseness or other cause.
- For high resistance ($> 1M\Omega$), it may require several seconds to obtain a stable reading.

Task 1.3 *Proceed as follows:*

1. *Measure the resistances of the resistors given to you and record them.*

1.4.2 Voltage Measurement

In this experiment, we will use the Multimeter as a Voltmeter to measure the voltage on the LED.

To measure the DC Voltage, we follow the following general procedure:

- Set the mode of the Multimeter to DC voltage mode by turning the rotary switch to "*DC Voltage Measurement*" in Fig. 1.7. The DC Voltage ranges are $200mV, 2V, 20V, 200V$ and $1000V$. If the value of voltage to be measured is unknown, use the maximum measurement position ($1000V$) and reduce the range step by step until a satisfactory reading is obtained.
- Insert the red test lead into the HzVΩ terminal and the black test lead into the COM input terminal.
- Connect the test leads across with the object to be measured. The red test lead on the higher voltage end and the black lead on the lower voltage end.
- The measured value shows on the display.
- If the LCD displays "1", it indicates that the existing selected range is overloaded: it is required to select a higher range in order to obtain a correct reading.

In each range, the Meter has an input resistance of approximately $10M\Omega$. This loading effect can cause measurement errors in high resistance (impedance) circuits. If the circuit impedance is less than or equal to $10k\Omega$, the error is negligible (0.1% or less). By connecting the Multimeter in parallel to the object for measuring the voltage across it, the overall resistance of the circuit will be slightly reduced, thus the voltage you measured will be a slightly lower than the actual voltage.

Task 1.4 *Proceed as follows:*

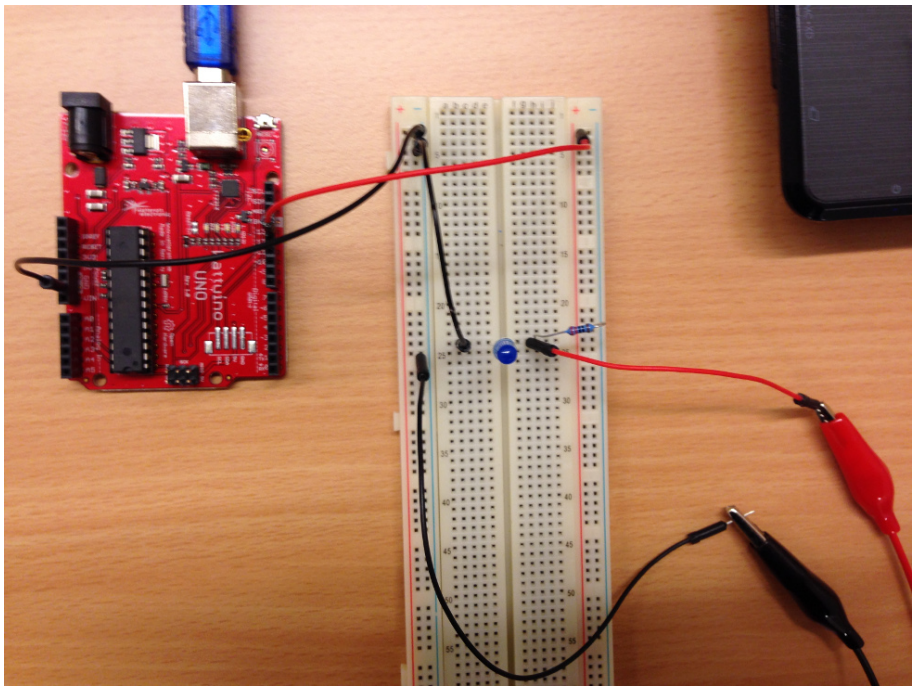


Figure 1.8: Jumpers connected to alligator clips (connected to the leads of the multimeter) for recording the voltage across the LED, which should be $V_{Forward}$.

1. *Unplug the Arduino (* ***Warning:*** *only touch the plastic parts of the USB cable and the Arduino board) and set up the circuit to measure $V_{Forward}$ as shown in Fig. 1.8. It is best to use the alligator clips and to connect them jumpers coming out of the appropriate notches of the breadboard as shown in Fig. 1.8. Do not directly connect the alligator clips to the resistor or the LED to avoid accidentally shorting them. The schematic of your circuit should look like Fig. 1.9. Call your instructor or TA to check that your circuit is correct.*

2. *Now plug in the Arduino to USB: It will start running the LED-blink program. Measure the voltage across the LED and record it. It will go to $V_{Forward}$ when the LED lights up.*

1.5 Buttons: digitalRead

Push-buttons or switches connect two points in a circuit when you press them. The push-button provided to you, shown in Fig. 1.10, has four pins. Two of them are connected to each other and form one-side of the connection; the other two are connected to each other and form the other side. When you press the button both sides are shorted.

Task 1.5 *Use your multimeter to find out which pins are connected to each other: turn the dial of the multimeter to the "Continuity Test" mode which has the icons of a diode and a musical note (marked as* ***Diode Test*** *in Fig. 1.7. Refer to page 20 of the TENMA Model 72-9385 Operating Manual). Now connect the two leads of the multimeter to the two pins of the button which you want to test for continuity - if they are shorted, you will hear a loud beep.*

The next example turns on the built-in LED on Pin13 when you press the button. The schematic of the circuit is shown in Fig. 1.11.

When the push-button is open (unpressed) there is no connection between the two legs of the push-button, so the pin-2 is connected to ground (through the pull-down resistor) and we

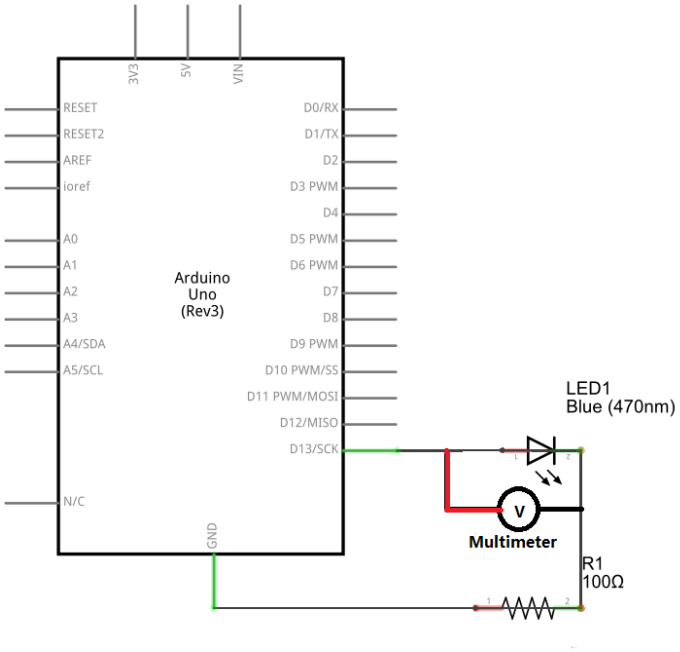


Figure 1.9: Use Multimeter as a Voltmeter.

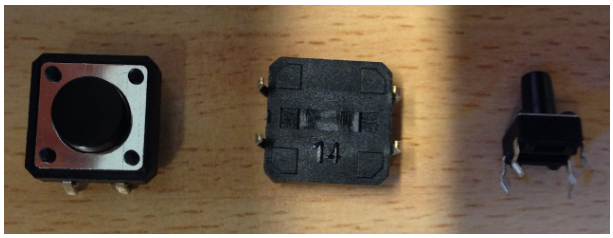


Figure 1.10: The bigger button is shown in top and bottom views. The smaller button is shown from the side. In the middle button, the shiny horizontal line just above the number "14" separates the two sides.

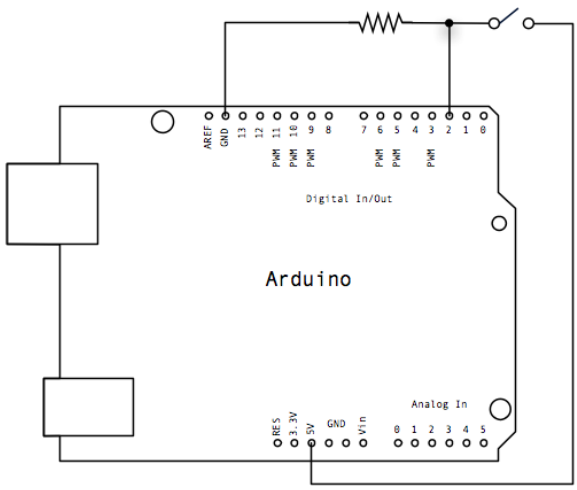


Figure 1.11: Using a push-button. Use a resistance of at least 330 Ω. We use the built-in LED connected to PIN13. Image credits: https://www.arduino.cc/en/Tutorial/Button

read a LOW from it. When the button is closed (pressed), it makes a connection between its two legs, connecting pin-2 to 5 volts, so that we read a HIGH.

You can run the following code on Arduino and check the results.

```
/*  Button
  Turns on and off a light emitting diode(LED) connected to digital
 pin 13, when pressing a pushbutton attached to pin 2. */

// constants won't change. They're used here to
// set pin numbers:
const int buttonPin = 2;     // the number of the pushbutton pin
const int ledPin =  13;      // the number of the LED pin

// variables will change:
int buttonState = 0;         // variable for reading the pushbutton status

void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
}

void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

sourcecodes/Button/Button.ino

Notice that in the code, one can set the pin to INPUT mode in setup(). In this example, Pin2 is set to INPUT mode in order to read in the voltage value from the circuit.

Task 1.6 *Perform the following subtasks:*

- *Setup the circuit as shown in Fig. 1.11. Use a resistor of at least* $330\,\Omega$. *Run the code on Arduino.*
- *Press the button – the built-in LED should light up.*

1.6 Potentiometer and Serial Monitor: analogRead

This example shows you how to read an analog input as an integer value from 0 to 1023, convert it into voltage and print it out to the serial monitor. The outer pins of the potentiometer connect

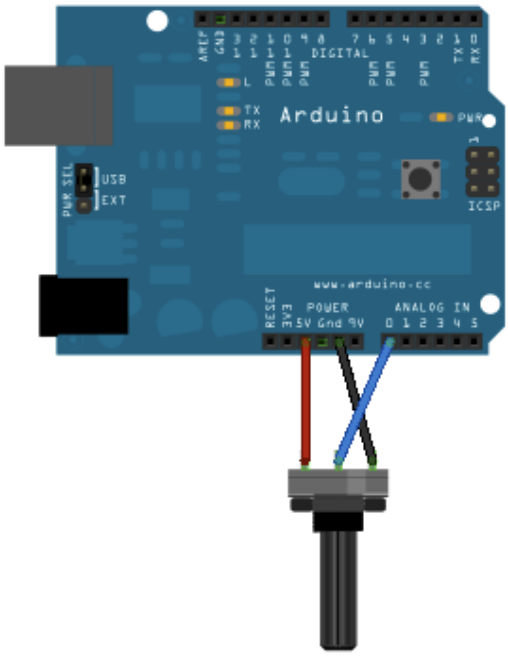


Figure 1.12: Using a potentiometer. Image credits: https://www.arduino.cc/en/Tutorial/ReadAnalogVoltage

to GND and $5V$ respectively and the middle pin to an analog input (pin $A0$). The potentiometer is a variable voltage divider - the middle pin divides the internal resistor in 2 series resistors depending on the knob position, therefore dividing the voltage applied to the outer terminals.

Task 1.7 *Measure the total resistance across the outer pins of the potentiometer given to you using a multimeter.*

The potentiometer pins cannot be inserted directly into the breadboard: use a female header set, or use female-to-female cables, as shown in Fig. 1.13.

The Arduino analog pin goes to the analog-to-digital converter (ADC) inside the microcontroller that measures the voltage 0 to $5V$ and converts it into a number sensorValue between 0 and 1023 by the following command:

```
int sensorValue= analogRead(A0);
```

Note that $1024 = 2^{10}$, so the ADC resolution is 10 bits. The value of 0 corresponds to 0V and the value of 1023 to 5V. To change the values from $0 - 1023$ to a range that corresponds to the

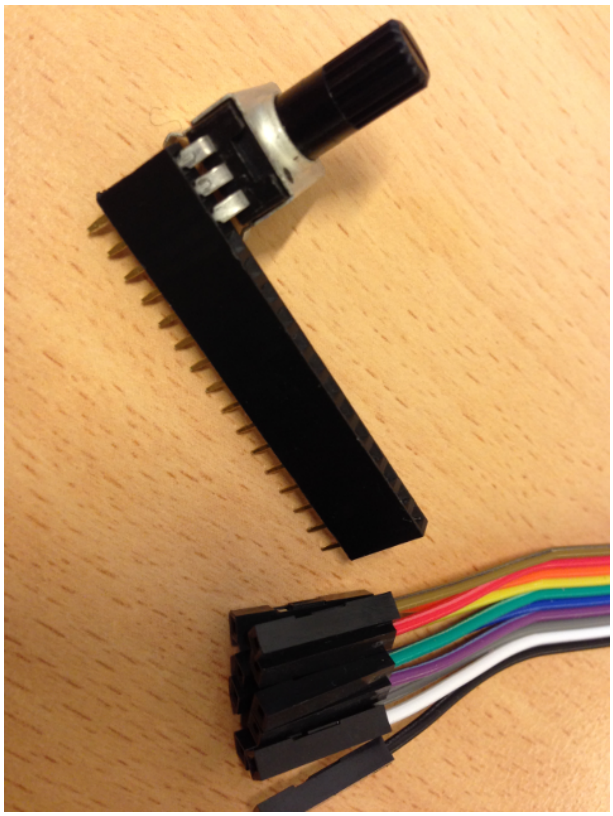


Figure 1.13: The potentiometer attached to a female header and female-to-female cables.

voltage the pin is reading, you'll need to create another variable, a float, and do a little math. To scale the numbers between 0.0 and 5.0, divide by 1023.0 and multiply that by sensorValue:

```
float voltage= sensorValue * (5.0 / 1023.0);
```

In order to print this value on our computer, we need to set up a serial communication protocol between the computer serial port and the Arduino by our USB cable. We can set up a serial communication at 9600 bits of data per second by the following command in the setup() function.

```
Serial.begin(9600);
```

Now we can print the voltage value on the serial monitor by the command

```
Serial.println(voltage);
```

Now, when you open your Serial Monitor in the Arduino development environment, you should see a steady stream of numbers ranging from $0.0 - 5.0$. As you turn the shaft of the Potentialmeter, the values will change, corresponding to the voltage coming into pin A0.

The complete codes is as follows:

```
/*  ReadAnalogVoltage
  Reads an analog input on pin 0, converts it to voltage,
  and prints the result to the serial monitor.
 */

// the setup routine runs once when you press reset:
void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
}

// the loop routine runs over and over again forever:
void loop() {
  // read the input on analog pin 0:
```

```
  int sensorValue = analogRead(A0);
  // Convert the analog reading (from 0 - 1023) to a voltage (0 - 5V):
  float voltage = sensorValue * (5.0 / 1023.0);
  // print out the value you read:
  Serial.println(voltage);
}
```

Listing 1.2: Using the potentiometer with analogRead

Task 1.8 *Refer to Fig. 1.12, and proceed as follows:*

1. *Setup the circuit and run the program on Arduino.*
2. *Change the amount of the resistance by turning the shaft of the potentiometer.*
3. *Open the serial monitor to observe the print-out.*

Chapter 2

Lab 2

In this lab, we are going to become familiar with the "Processing" programming language, its usage, and how we can make it communicate with Arduino.

2.1 Get Started with Processing

Processing is an open-source JAVA-like programming language mainly for visualizing data. Now please open the IDE of Processing and you will see a window similar in Fig. 2.1. It looks very similar to the Arduino IDE – this is because the Arduino IDE was originally inspired by Processing.

2.1.1 Your First Program

Note: If you want to know more details about a function, go to the online reference: https://processing.org/reference/

You're now running the Processing Development Environment (or PDE). There's not much to it; the large area is the Text Editor, and there's a row of buttons across the top; this is the toolbar. Below the editor is the Message Area, and below that is the Console. The Message Area is used for one line messages, and the Console is used for more technical details.

In the editor, type the following:

```
ellipse(50, 50, 80, 80);
```

This line of code means "draw an ellipse, with the center 50 pixels over from the left and 50 pixels down from the top, with a width and height of 80 pixels." Click the Run button, which looks like as shown in Fig. 2.2(a). If you've typed everything correctly, you'll see the display-window shown in Fig. 2.2(b).

If you didn't type it correctly, the message area will turn red and complain about an error. If this happens, make sure that you've copied the example code exactly: the numbers should be contained within parentheses and have commas between each of them, and the line should end with a semicolon. The code that you are typing is actually Java: if you know C/C++, it will not look too unfamiliar.

2.1.2 Draw

The computer screen (window) is composed of pixels. To create a new window, use the function *size(width, height)*. For example, `size(400,200)` creates a window with 400 pixels wide and 200 pixels high.

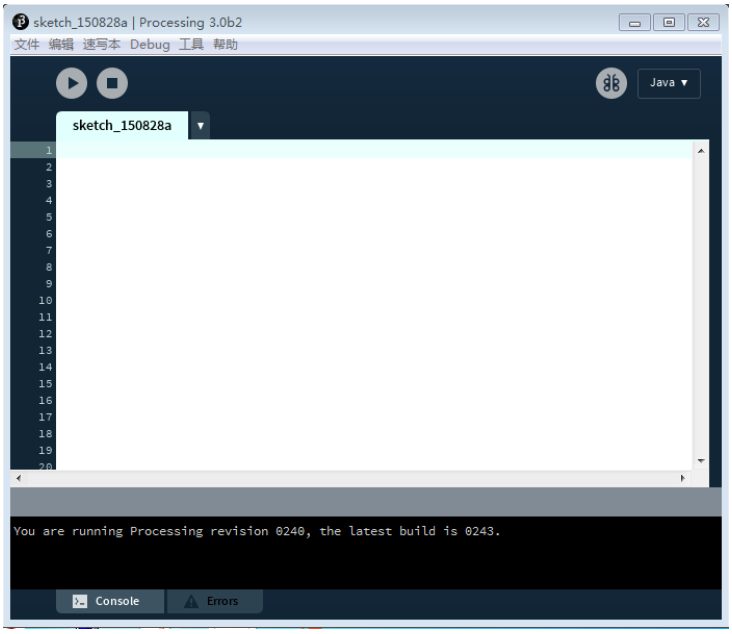
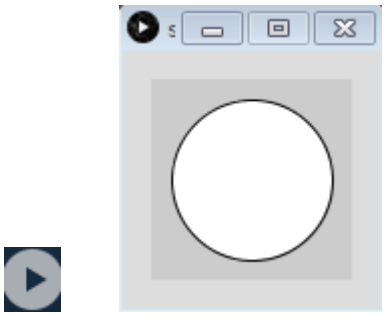


Figure 2.1: The Processing IDE Window.



(a) Run (b) A circle

Figure 2.2: Your first program.

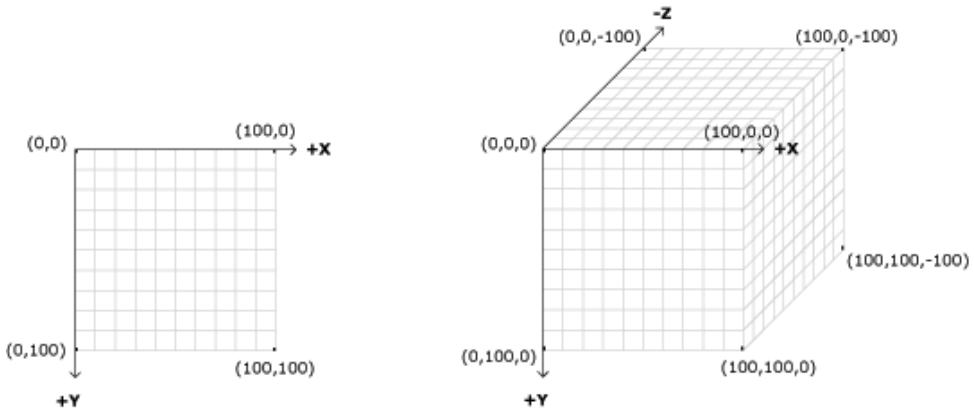


Figure 2.3: The Processing Coordinate-System.

Processing uses a Cartesian coordinate system with the origin in the upper-left corner. If your sketch is 320 pixels wide and 240 pixels high, coordinate $(0, 0)$ is the upper-left pixel and coordinate $(320, 240)$ is in the lower-right. The last visible pixel in the lower-right corner of the screen is at position $(319, 239)$ because pixels are drawn to the right and below the coordinate.

Draw a Point:

The following codes draw a point at coordinate $(250, 60)$. Try it your self and feel free to change the value inside the functions.

```
size(400,200); point(250,100);
```

Basic Shapes:

Processing includes a group of functions to draw different geometric graphs:

Syntax	Description
line$(x1, y1, x2, y2)$	$(x1, y1)$: start point; $(x2, y2)$: end point
triangle$(x1, y1, x2, y2, x3, y3)$	points of a triangle
quad$(x1, y1, x2, y2, x3, y3, x4, y4)$	points of a four sided polygon
rect$(x, y, width, height)$	(x, y) is the top-left point of a rectangle with the given sides.
ellipse$(x, y, width, height)$	(x, y) is the center of a circle
arc$(x, y, width, height, start, stop)$	*start* and *stop* are angles to start and stop an arc

Try to draw different shapes by yourself. For the angle parameters in the *arc* function, you can use *radians(180)* to denote the 180-degree angle.

The Order:

The computer runs the code line by line. If you want to put one object on top of others, you need to draw it after them.

Try the following codes and observe the difference.

Example: `size(400,200); ellipse(200,100,60,60); rect(50,50,200,50);`

Example: `size(400,200); rect(50,50,200,50); ellipse(200,100,60,60);`

Task 2.1 *Draw a 2D car.*

The Attributes of a Stroke:

Some most frequently used functions are listed below:

Syntax	Description
smooth()	smooth the line
strokeWeight(weight)	set the weight (in pixel) of a stroke
strokeJoin(join)	join ={MITER,BEVEL,ROUND}
strokeCap(cap)	cap ={SQUARE, PROJECT, ROUND}
rectMode(mode)	mode ={CORNER, CORNERS, CENTER, RADIUS}
ellipseMode(mode)	mode ={CORNER, CORNERS, CENTER, RADIUS}

Try the above functions to observe the different attributes of a stroke.

Example:

```
size(400,200); smooth(); strokeWeight(12); strokeJoin(ROUND); rect(50,50,150,150);
```

Color:

We can use *background(),fill()* and *stroke()* to change the color of a background, of an object, of a stroke.

For a black-white picture, only one value between $0 - 255$ is necessary. For example, 255 denotes white, 128 denotes Gray and 0 denotes black. For a color picture. One use three values to denote the combination of a RGB color.

Example:

```
size(400,200);
noStroke(); // no stroke
smooth(); //smooth
background(0,26,51); //set background to be dark blue
fill(255,0,0); // set color to be red
ellipse(132,50,150,150); // draw a circle
```

sourcecodes/Color/Color.pde

Task 2.2 *Improve your car from the last task by using different strokes and colors.*

2.2 Moving Object

In this section, we are going to learn how to use the mouse to interact with our drawing. In order to interact with the mouse, we need our code to run continuously. There is a function called *draw()* in Processing which runs repeatedly until you click the stop button. Every iteration of *draw()* constitutes one *frame*. The default frame rate is 60 frames per second.

Usually *draw()* is used together with an initialization function *setup()*. *setup()* only runs once in the beginning. We set up the frame rate inside *setup()* by function *frameRate(rate)*. Note how this is very similar to an Arduino sketch.

Run the following example and try to understand it.

```
float x = 60;
float y = 60;

int diameter = 60;

void setup() {
 size(480, 360);
 frameRate(30);
```

```
}

void draw()
{
 background(102);
 x = x+2.8;
 y = y+2.2;
 ellipse(x, y, diameter, diameter);
}
```

sourcecodes/Moving/Moving.pde

In this example, x, y are global variables and can be seen inside both functions *setup()* and *draw()*.

Task 2.3 *Run the example code and Move the function* `background()` *into* `setup()` *to observe the change. Find out why this change leads to different result.*

2.3 Interact with the Mouse

In order to interact with the mouse, we need variables to store the coordinate of the current mouse position. In Processing, *mouseX* and *mouseY* store the coordinate of the current mouse position.

Try the following example.

```
void setup() {
 size(480, 360);
 smooth();
 noStroke();
 fill(102);
}

void draw()
{
 ellipse(mouseX, mouseY, 9, 9);
}
```

sourcecodes/Follow/Follow.pde

Add the function *background(204)* inside *draw()* and observe the difference.

Next, we use another example to see how to detect mouse pressed by a boolean variable *mousePressed*.

```
void setup() {
  size(480, 120);
}

void draw() {
  if (mousePressed) {
    fill(0);
  } else {
    fill(255);
  }
  ellipse(mouseX, mouseY, 80, 80);
}
```

sourcecodes/Mouse/Mouse.pde

This program creates a window that is 480 pixels wide and 120 pixels high, and then starts drawing white circles at the position of the mouse. When a mouse button is pressed, the circle color changes to black.

The next example will show you how to detect which buttons of the mouse is pressed.

```
void setup() {
  size(480, 120);
}

void draw() {
  if (mousePressed) {
    fill(0);
    if (mouseButton == LEFT)
       println("Left mouse pressed.");
    else
       println("Right mouse pressed.");
  }
  else {
    fill(255);
  }
  ellipse(mouseX, mouseY, 80, 80);
}
```

sourcecodes/MouseDetect/MouseDetect.pde

Task 2.4 *Now, with all you have learned, you are already able to program a small animation. Modify Listing 2.2 so that ball bounces off the "walls". The bouncing off angle should be realistic. Hint: To compute the 2D vector direction of the bouncing, you can use the result discussed in GIMS-1, as shown in Fig. 2.4. You can resolve the incoming direction* $\mathbf{v}$ *into a component parallel to the wall unit-normal* $\hat{\mathbf{a}}$ *and perpendicular to it:*

$$\mathbf{v} = \mathbf{v}_{\parallel} + \mathbf{v}_{\perp}, \text{ where,} \tag{2.1}$$

$$\mathbf{v}_{\parallel} = (\mathbf{v} \cdot \hat{\mathbf{a}})\,\hat{\mathbf{a}} \tag{2.2}$$

$$\mathbf{v}_{\perp} = \mathbf{v} - \mathbf{v}_{\parallel}. \tag{2.3}$$

The direction in which the ball bounces is then

$$\mathbf{b} = -\mathbf{v}_{\parallel} + \mathbf{v}_{\perp} \tag{2.4}$$

To implement this simply, maintain two global variables float vx=2 and float vy=3 representing the components of the velocity vector, i.e. amount by which the x and y coordinates of the ball change, in each draw iteration. Now any time the ball hits one of the walls, you just negate either vx or vy while keeping the other component to be the same.

2.4 2D Transforms

2.4.1 Translation Transform

As shown in Fig. 2.5(c), in Processing, you typically, keep the coordinates of the shape static, but move the underlying imaginary graph-paper on which the shape is drawn.

```
void setup()
{
  size(200, 200);
  background(255);
  noStroke();

  // draw the original position in gray
  fill(192);
  rect(20, 20, 40, 40); // rectangle R1
```

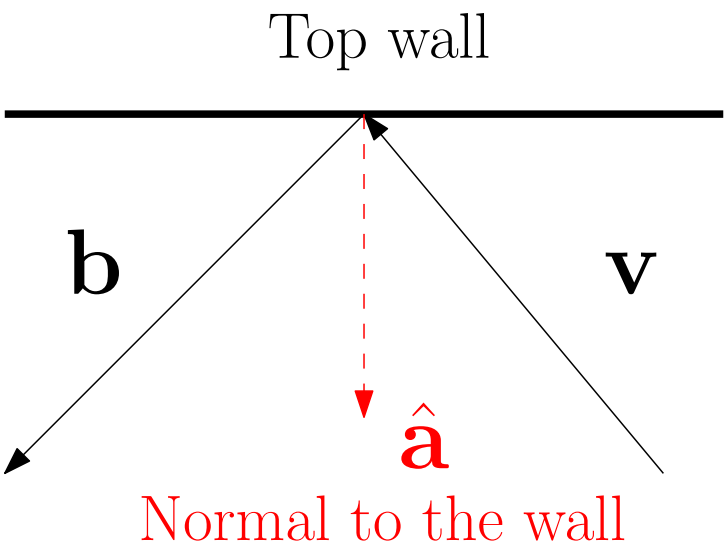
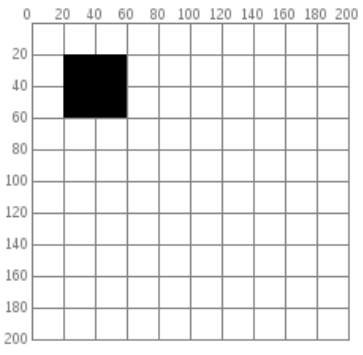
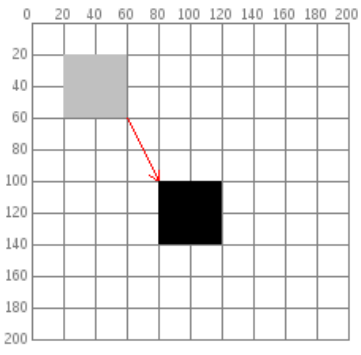


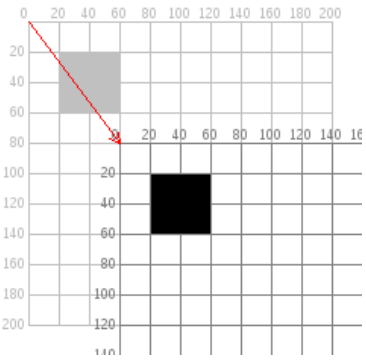
Figure 2.4: Computing the bouncing direction.



(a) `rect(20, 20, 40, 40)`

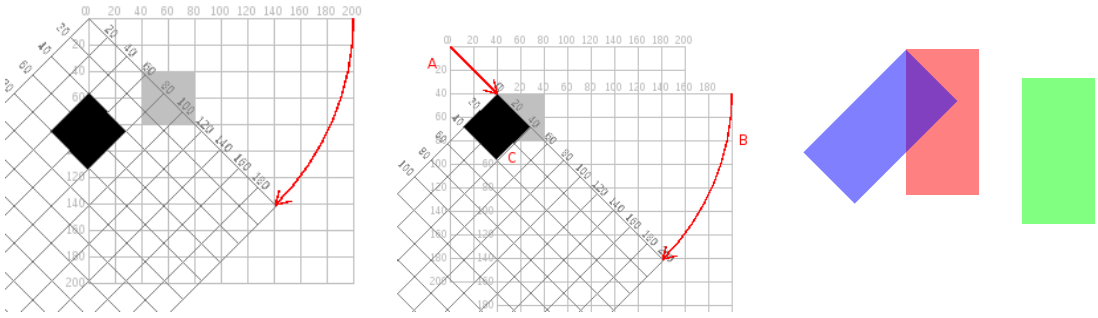


(b) `rect(20 + 60, 20 + 80, 40, 40)`



(c) move the graph paper (coordinate system) instead, and use the same drawing command as in Fig. 2.5(a).

Figure 2.5: Image credits: https://www.processing.org/tutorials/transform2d/



(a) Rotation about the origin of the CS.

(b) Rotation about the origin of the translated CS.

(c) The output

Figure 2.6: First two images are from: https://www.processing.org/tutorials/transform2d/

```
  // draw a translucent red rectangle by changing the coordinates
  fill(255, 0, 0, 128);
  rect(20 + 60, 20 + 80, 40, 40); // R2: the moved R1

  // draw a translucent blue rectangle by translating the grid
  fill(0, 0, 255, 128);
  pushMatrix(); // save the current position of the graph-paper (coordinate-
    system)
  translate(60, 80); // move the graph-paper (move the origin of the coordinate-
    system)
  rect(20, 20, 40, 40); // use the same command to draw a rectangle as for R1
  popMatrix(); // restore the previous position of the graph-paper
}

void draw()
{
  if (mousePressed==true){
     pushMatrix(); // save the current position of the graph-paper
     translate(mouseX, mouseY); // move the graph-paper
     rect(20, 20, 40, 40); // use the same command to draw a rectangle as for R1
     popMatrix(); // restore the previous position of the graph-paper
  }
}
```

sourcecodes/Transforms/move.pde

Task 2.5 *Why do you see a purple rectange after the setup is called but before you start interacting with the mouse?*

2.4.2 Rotation Transform

Let's now draw on a PDF file. Running the sketch below will create a PDF file: To view it, press Ctrl+k to open the sketch folder.

Warning: If you cut and paste code, make sure that there are no spurious blank spaces in the import command.

```
import processing.pdf.*;

int ox= 80;
int oy= 20;

void draw_rectangle(int rx, int ry){
  rect(rx, ry, 50, 100);
```

```
}

void setup(){
  size(250, 200, PDF, "trf_rot.pdf");
  background(255); // white background
  smooth();
  fill(255,0, 0, 127); // red, opacity= 127/255= about 0.5
  noStroke();
  draw_rectangle(ox, oy);

  pushMatrix();

  // move the origin to the pivot point
  translate(ox, oy);
  pushMatrix();

  // then pivot the grid
  rotate(radians(45));

  fill(0, 0, 255, 127); // blue, opacity= 127/255= about 0.5
  draw_rectangle(0, 0);

  popMatrix();

  fill(0, 255, 0, 127); // green, opacity= 127/255= about 0.5
  draw_rectangle(ox, oy);
  popMatrix();

}
```

sourcecodes/Transforms/rotation.pde

Task 2.6 *Why is the green rectange drawn where it is drawn?*

2.4.3 Scaling Transform

```
import processing.pdf.*;

int ox= 80;
int oy= 20;

void draw_rectangle(int rx, int ry){
  rect(rx, ry, 50, 60);
}

void setup(){
  size(300, 200, PDF, "trf_scaling.pdf");
  background(255); // white background
  smooth();
  fill(255, 0, 0, 200); // red, opacity= 200/255= about 0.78
  stroke(0, 0, 0, 250);

  draw_rectangle(ox, oy); // R1

  pushMatrix();
  scale(2, 2); // scale by factor 2
  fill(0, 0, 255, 127); // blue, opacity= 127/255= about 0.5
  draw_rectangle(ox, oy); // R2: But this appears shifted!!!
  popMatrix(); // undo the scaling

  pushMatrix();
  fill(0, 255, 0, 127); // green, opacity= 127/255= about 0.5
```

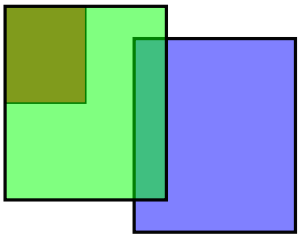


Figure 2.7: Output.

```
  translate(-ox, -oy); // Initial translation, so that R3 and R1 have the same
    top-left vertex
  scale(2, 2);
  draw_rectangle(ox, oy); // R3
  popMatrix();

}
```

sourcecodes/Transforms/scaling.pde

Task 2.7 *Why did the blue rectange R2 appear shifted w.r.t. the red-rectange, although they were drawn using the same command?*

2.4.4 Order of Transforms

The order in which you apply the translation, rotation, and scaling transforms is important: Changing the order will lead to very different results.

```
int ox = 80;
int oy = 20;

void draw_rectangle ( int rx , int ry ) {
  rect ( rx , ry , 50 , 60) ;
}

void setup () {
  size (300 , 200) ;
  background (255) ; // white background
  stroke (0 , 0 , 0 , 250) ;

  fill (0 , 0 , 255 , 174) ; // transparent blue
  draw_rectangle ( 0 , 0 ) ; // The base rectangle

  pushMatrix();
  translate ( ox , oy ) ;
  scale (2 , 2) ;
  rotate ( radians (45) ) ;
  fill (255 , 0 , 0 , 200) ; // transparent red
  draw_rectangle ( 0 , 0 ) ;
  popMatrix();
}
```

Listing 2.1: The importance of transform order

Task 2.8 *Shuffle the orders of the three transforms (translate, scale, rotate) in Listing 2.1 and explain the differences observed. Try at least one other permutation.*

2.5 3D Transforms

In Processing 2.0 and above, there are four render modes: the default renderer, P2D, P3D, and PDF which are specified within the `size()` command. We use the P3D renderer for drawing in 3D. There are also contributed libraries available in Processing which implement a variety of features: refer to https://processing.org/reference/libraries/#3d.

```
float rotx, roty;

void setup() {
  size(400,400,P3D);
  rotx= 0.0;
  roty= 0.0;
}

void draw() {
  background(255);
  pushMatrix();
  translate(width/2.0, height/2.0, -50);
  rotateX(rotx);
  rotateY(roty);
  draw_axes();
  popMatrix();
}

void draw_axes() {
  fill(100, 200, 100, 127);
  stroke(0);
  box(90, 60, 30);

  stroke(255, 0, 0);
  line(0, 0, 0, 100, 0, 0); // Red X-Axis
  stroke(0, 255, 0);
  line(0, 0, 0, 0, 100, 0); // Green Y-Axis
  stroke(0, 0, 255);
  line(0, 0, 0, 0, 0, 100); // Blue Z-Axis
}

/** This is an event-handler function.
*/
void mouseDragged() {
  float rate = 0.01;
  rotx += (pmouseY-mouseY) * rate;
  roty += (mouseX-pmouseX) * rate;
}
```

Listing 2.2: Rotation in 3D

Run the code in Listing 2.2. You will see that the XYZ coordinate-system (CS) in Processing is unfortunately a non-standard left-handed CS. The origin is at the top-left, X-axis points to the right of the screen, Y-axis points to the bottom of the screen, and the Z-axis is coming out of the screen. This is also depicted in Fig. 2.3.

2.5.1 Camera and Lights

Instead of using 3D transforms to see an object from different views, sometimes it is more convenient to use a 3D camera looking at a static object, as shown in Listing 2.3.

```
void setup() {
  size(400,400,P3D);
}
```

```
void draw() {
  background(255);
  camera(
         mouseX - width/2, mouseY - height/2, 100, // Eye (camera) position
         0, 0, 0, // scene-center (where the camera is looking)
         0, 1, 0 // "up" direction
        );

  ambientLight(255,150,150);

  spotLight(
             0, 255, 0, // color of the light: green
             0, height/2, 300, // 3D location of the light source
             0, 0, -1, // the direction in which the light source is pointing
             PI/4, // The spotlight cone-angle
             10 // The concentration: how biased is the light towards
               //   the center of the spotlight cone
  );

  draw_axes();
}

void draw_axes() {
  fill(100, 200, 100);
  stroke(0);
  box(90, 60, 30);

  stroke(255, 0, 0);
  line(0, 0, 0, 100, 0, 0); // Red X-Axis

  stroke(0, 255, 0);
  line(0, 0, 0, 0, 100, 0); // Green Y-Axis

  stroke(0, 0, 255);
  line(0, 0, 0, 0, 0, 100); // Blue Z-Axis
}
```

Listing 2.3: Camera and Lights

Task 2.9 *Modify Listing 2.3 to add the following functionality:*

- *Create the central cuboid such that each face has a different color. Refer to https://processing.org/examples/rgbcube.html.*
- *Right now, in the code line:*
  `mouseX - width/2, mouseY - height/2, 100, // Eye (camera) position` *as the x and y coordinates of the mouse change, the distance of the camera from the object changes. Why?*
- *To avoid this effect and hence to keep the camera a constant distance from the object, let us map the mouse-movements to shpherical coordinates with radius $R = 200$. We want the camera coordinates x_c, y_c, z_c to change as follows:*

$$x_c = R \cos\phi \cos\theta \tag{2.5}$$
$$y_c = R \sin\phi \cos\theta \tag{2.6}$$
$$z_c = R \sin\theta \tag{2.7}$$

  where, the angles $\phi \in (-\pi, \pi]$ and $\theta \in [-\pi/2, \pi/2]$ are mapped to the mouse offsets `mouseY - height/2` *and* `mouseX - width/2` *respectively. Modify the code and see the result.*
- *Use keyboard events to be able to change the radius R of the spherical coordinates. Refer to https://www.processing.org/tutorials/interactivity/.*

2.6 Communication from Processing to Arduino

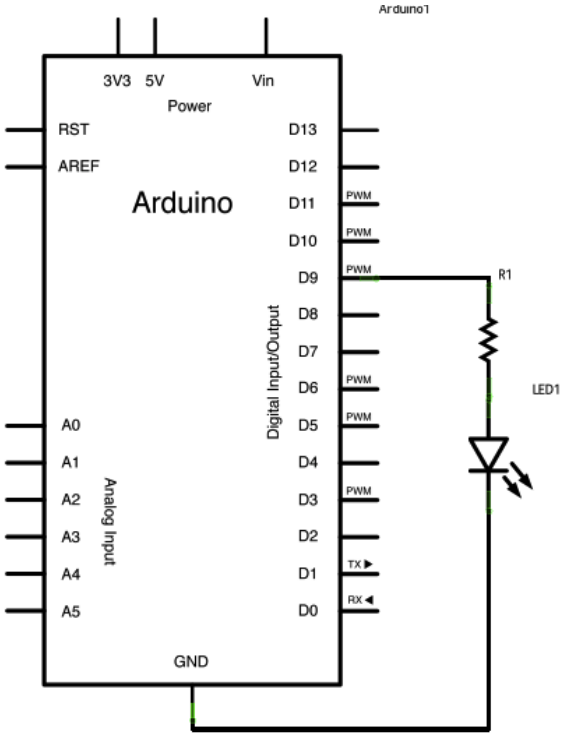
In this lab, we are going to connect Processing to Arduino and control an LED from Processing.

2.6.1 Using the Serial Communication: analogWrite

In the first lab session, we already know that data can be transferred by a sketch running on Arduino via USB cable to the computer using serial communication. Arduino has a serial monitor to print out the value of the data. Processing also offers library to deal with data via serial communication. In order to use serial communication in Processing, we need to import a library in *setup()*: `import processing.serial.*;`

In this example, we are going to learn how to use the X coordinate of the mouse to control the brightness of an LED. In Processing display window, we can visualize the brightness of the LED depend on the X coordinate of the mouse.

Please set up the circuit as the diagram shows:



Arduino reads data from the serial port and stores the data into a variable called *brightness* (ranging from 0 to 255) and uses them to set the brightness of the LED.

```
const int ledPin = 9;      // the pin that the LED is attached to

void setup()
{
  // initialize the serial communication:
  Serial.begin(9600);
  // initialize the ledPin as an output:
  pinMode(ledPin, OUTPUT);
}

void loop() {
  byte brightness;

  // check if data has been sent from the computer:
  if (Serial.available()) {
    // read the most recent byte (which will be from 0 to 255):
```

```
    brightness = Serial.read();
    // set the brightness of the LED:
    analogWrite(ledPin, brightness);
  }
}
```

sourcecodes/Dimmer/Dimmer.ino

The data is the X coordinate of the mouse sent by Processing.

```
import processing.serial.*;

Serial port;                              // Create object from Serial class
int val;                                  // Data received from the serial port

void setup() {
  size(200, 200);
// Open the port that the board is connected to and use the same speed (9600 bps
    )
  port = new Serial(this, "COM3", 9600);
}

 void draw() {
 // draw a gradient from black to white
 for (int i = 0; i < 256; i++) {
 stroke(i);
 line(i, 0, i, 150);
 }

 // write the current X-position of the mouse to the serial port as
 // a single byte
 port.write(mouseX);
 }
```

sourcecodes/Dimmer/dimmer/dimmer.pde

Task 2.10 *Understand the code and successfully control the brightness of an LED by the mouse.*

2.6.2 Using the Firmata Library

The Firmata library allows you to control an Arduino board from a Processing sketch in a very generic way. On the Arduino side you run a standard Firmata sketch from the examples. On the Processing side, you can now fully configure the Arduino board directly in your Processing sketch!

Arduino Side

Prepare the circuit as shown in Fig. 2.8. An RGB LED consists of three LEDs of red, green, and blue colors together sharing a common anode. Hence, there are 4 pins in all. The common anode is the second pin which is the longest. On one side of it is the red-LED's cathode, and on the other side of it are the cathodes of the green and blue LEDs, in that order.

You can use a resistance of 200 Ω for all three LEDs, although, for better results, you can choose them specifically for color based on the experiment in Lab 1 (Table 1.2).

In the Arduino IDE open the example sketch: File ⟶ Examples ⟶ Firmata ⟶ StandardFirmata, compile it and transfer to Arduino as usual. Nothing will happen yet. Proceed further as described below.

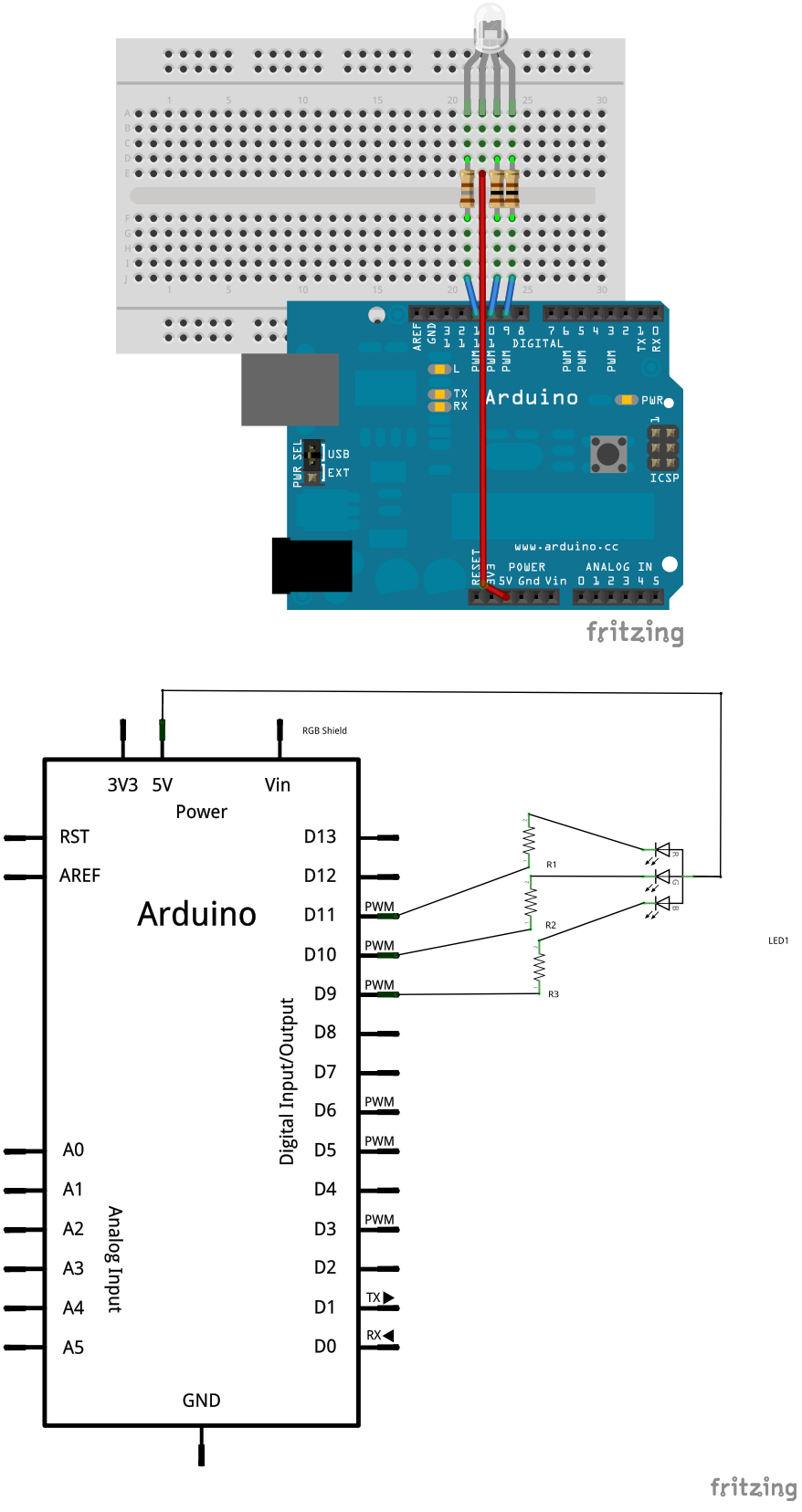


Figure 2.8: The circuit for a common-anode RGB LED.

Processing Side

In Processing, follow the following steps:

- Goto Sketch $\longrightarrow$ Import Library $\longrightarrow$ Add Library, and select "Arduino (Firmata)". It will be placed in a sub-folder called "libraries" in your sketchbook. You do not necessarily have to double check it, but if you are curious about the location of your sketchbook folder, select the "Preferences" option from the File menu (or from the "Processing" menu on the Mac) and look for the "Sketchbook location." In the "libraries" folder, you should now see a sub-folder there called "arduino". If you are running Linux, you need to go to the sub-older sketchbook/libraries/arduino/library and change the name of the file Arduino.jar to arduino.jar and restart Processing.
- Put a colorful image (called `color-wheel.jpg` in the example in the subfolder "data" of your sketch folder (press **Ctrl+k** to open it). Create this subfolder if it does not exist. Make sure that your image does not have a double extension like "`color-wheel.jpg.jpg`": the last extension may be hidden on Windows Explorer.

```
import processing.serial.*;
import cc.arduino.*;

// Arduino related
Arduino arduino;
int red_pin = 11;
int green_pin = 10;
int blue_pin = 9;

// Declaring a variable of type PImage
PImage img;

void setup(){
  //println(Arduino.list());
  arduino = new Arduino(this, Arduino.list()[0], 57600);
  // If the LED color does not change, the Arduino port is wrong.
  // Replace Arduino.list()[0] to the right port-name (such as "COM1").

  arduino.pinMode(red_pin, Arduino.OUTPUT);
  arduino.pinMode(green_pin, Arduino.OUTPUT);
  arduino.pinMode(blue_pin, Arduino.OUTPUT);

   size(500 , 500) ;
  // The image file should exist in the data subfolder of the sketch folder
  img= loadImage("color-wheel.jpg");
  background(0);
  // Draw the image to the screen at coordinate (0,0)
  image(img, 0, 0);
  loadPixels(); // tell Processing, we want to access pixel values
}

void draw(){
  if (mousePressed == true) {
    // Get the color at mouseX, mouseY
    // The pixels are stored in row-major format
    int loc= mouseY*width + mouseX;
    int r = int( red(pixels[loc]) ); // convert float to int
    int g = int( green(pixels[loc]) );
    int b = int( blue(pixels[loc]) );
    setColor(r, g, b);
  }
}
```

```
void setColor(int r, int g, int b){
  // We are using a common-ANODE RGB LED.
  arduino.analogWrite(red_pin, 255-r);
  arduino.analogWrite(green_pin, 255-g);
  arduino.analogWrite(blue_pin, 255-b);
}
```

Listing 2.4: Controlling an RGB LED's color by Processing.

Task 2.11 *Proceed as follows:*

1. *Set up the circuit and run the sketch in Listing 2.4. There may be a few seconds delay before the image shows up. The color of the LED should change based on the color of the pixel you clicked on.*

2. *Explain why the sketch works, in particular, why we subtracted the RGB values from 255 in the* setColor *function before sending them to Arduino? Hint: Think about the voltages being applied across the individual red, green, and blue LEDs.*

Chapter 3

Lab 3

In this lab, we are going to connect several different sensors to the Arduino and see their output. The list of sensors and their data-sheets can be accessed here.

3.1 Light Dependent Resistor (LDR)

3.1.1 Measuring Resistance Range Manually

An LDR or photo-cell (refer to Fig. 3.1) changes its resistance based on the light intensity. Bright light leads to a lowering of resistance while covering the LDR to block light will lead to a substantial increase of resistance.

Task 3.1 *We would like to measure the range of resistance values exhibited by the LDR provided to you under various light conditions.*

1. *Follow the instructions for measuring resistances on your multimeter. For the TENMA 72-9385, these are on page 17 of the Operating Manual.*
2. *Cover the LDR by a thick piece of paper and note down the resistance measured. You may have to wait for a few seconds for the value to stabilize.*
3. *Subject the LDR to bright light and note down the resistance measured.*

3.1.2 LDR in a Voltage-Divider

This experiment is reminiscent of the one with a potentiometer in Sec. 1.6. Look at the circuit in Fig 3.2. The objective is to turn the LED on as soon as the LDR detects that it has become

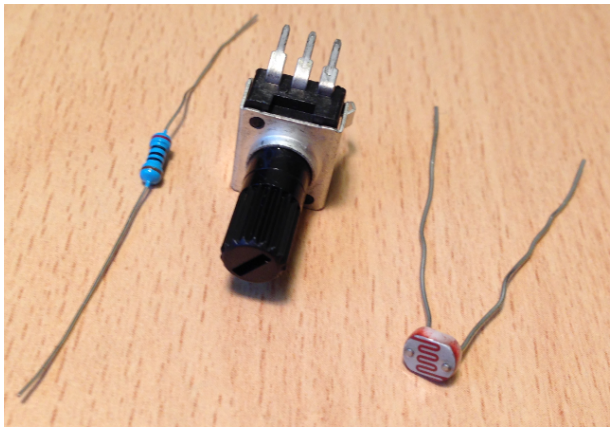


Figure 3.1: From left to right: A normal resistor, a potentiometer, and an LDR.

dark. The LDR is read by analog-input.

```
int input_pin= A0;
int LED = 10;
int sensor_sample = 0;

void setup(){
 Serial.begin(9600);
 pinMode (LED, OUTPUT);
}

void loop(){
  sensor_sample = analogRead(input_pin);
  Serial.print("sensor value = " );
  Serial.println(sensor_sample);

  if (sensor_sample < 650 ){
    digitalWrite(LED, HIGH);
  }else{
     digitalWrite(LED, LOW);
  }
  delay (50);
}
```

Listing 3.1: Using the potentiometer with analogRead

Task 3.2 *Set up the circuit as in Fig. 3.2 and upload the code Listing 3.1. Using the voltage divider rule, the voltage-measured by the analog-input A0 is* $5V \times R_2/(R_1 + R_2)$. *Recall that the ADC returns a value between* 0 *and* 1023 *for the input voltage range of* 0 *to* $5\,V$.

1. *Based on the values you measured in Task 3.1, is it better to use* $R_2 = 1\,K\Omega$ *or* $R_2 = 10\,K\Omega$*?*
2. *Depending on which value you selected for* R_2*, you may have to change the threshold* 650 *in Listing 3.1. Tweak its value such that the LED turns on when the LDR is covered.*

3.2 The Piezo-Buzzer Plays a Melody

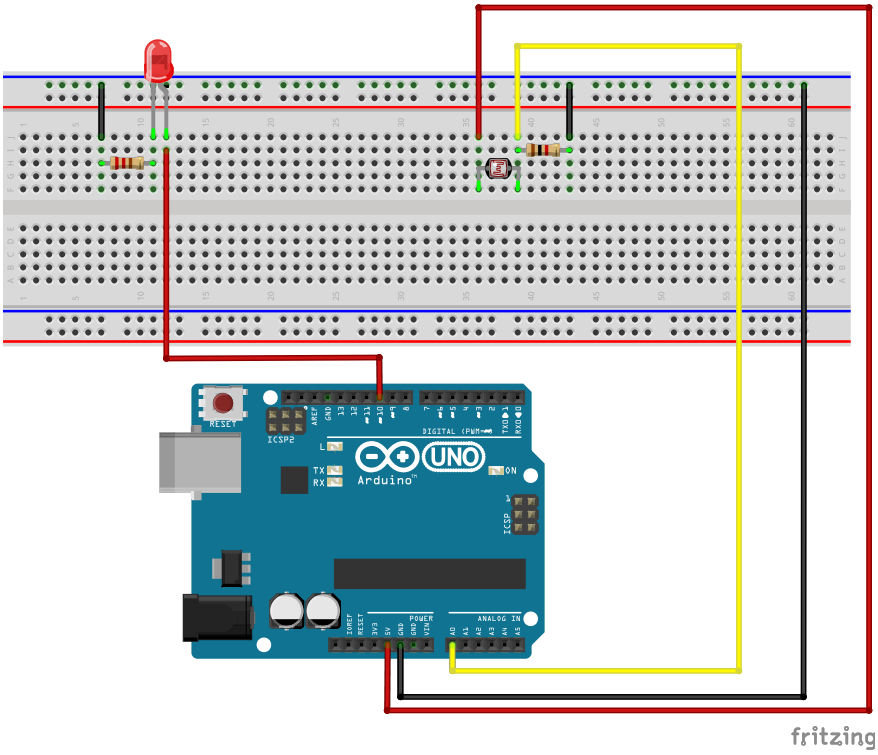
In this experiment, we play a melody on a piezo-buzzer (Fig. 3.3) using the tone() function, and change the tempo of the melody using a potentiometer. This experiment is an extended version of the basic one at this link.

Warning: The two pins are of unequal size, the positive being longer. If the buzzer doesn't fit into the breadboard easily, try rotating it slightly to fit into diagonal holes.

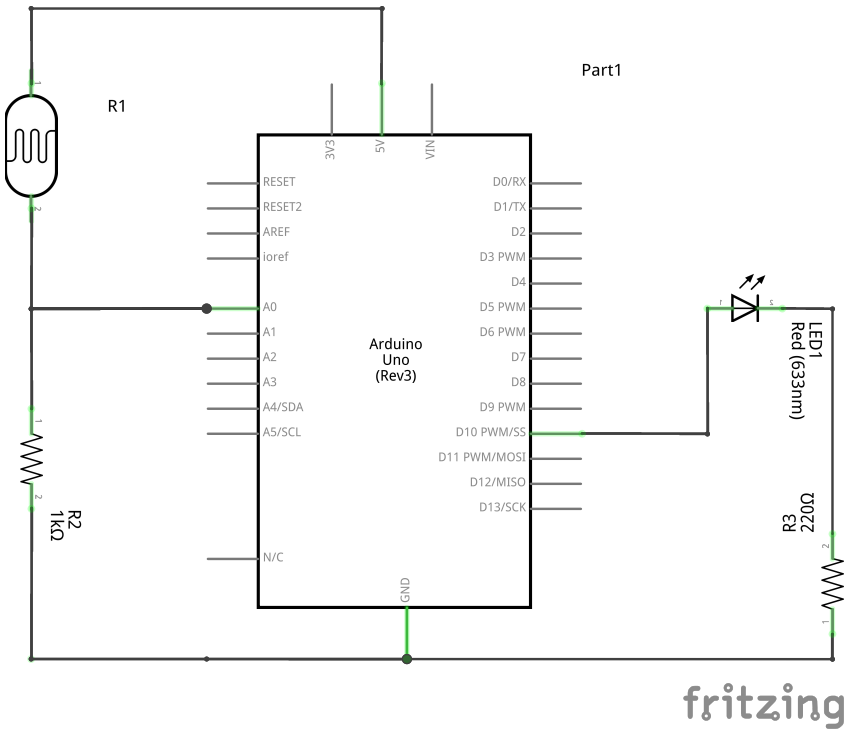
The Arduino has a built-in command called tone() which clicks the buzzer at a certain frequency for a given length of time. We hard-code the frequencies of some common notes. A more complete list can be found here.

Task 3.3 *Proceed as follows:*

1. *Measure the total resistance across the pins of the piezo-speaker given to you using a multimeter: mind the polarity.*
2. *Set up the circuit as shown in Fig. 3.4.*
3. *You should be able to control the tempo of the melody by turning the dial of the potentiometer. Why does the melody play twice every time you turn the dial?*



(a) Breadboard connections.



(b) Schematic.

Figure 3.2: LDR in a voltage divider

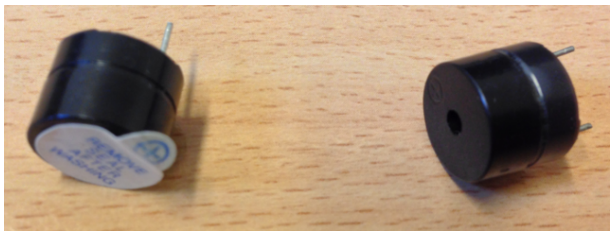
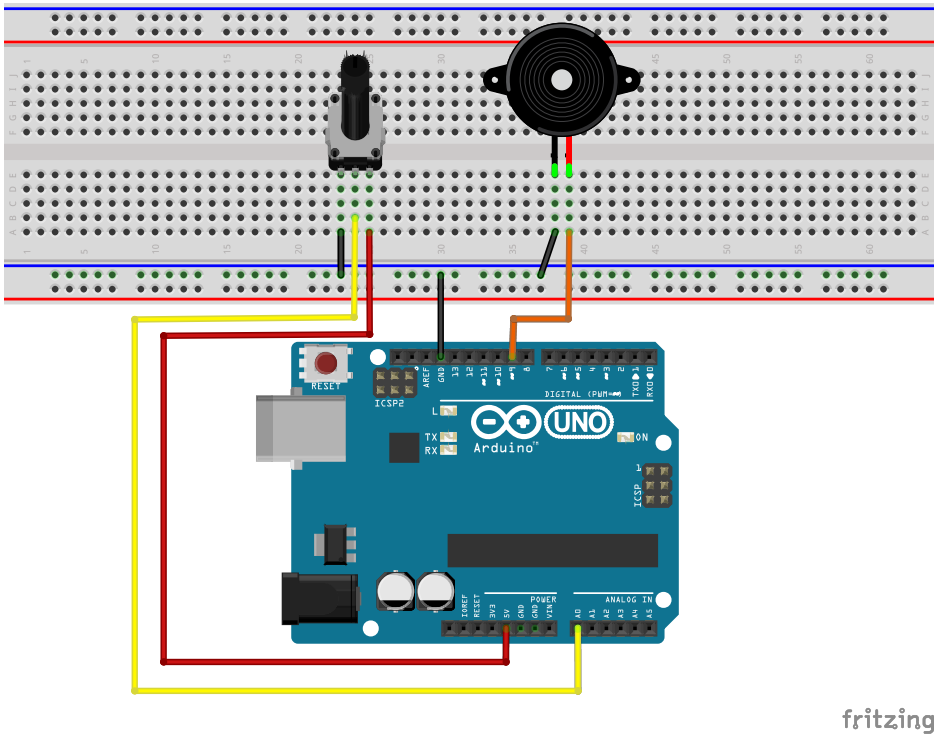
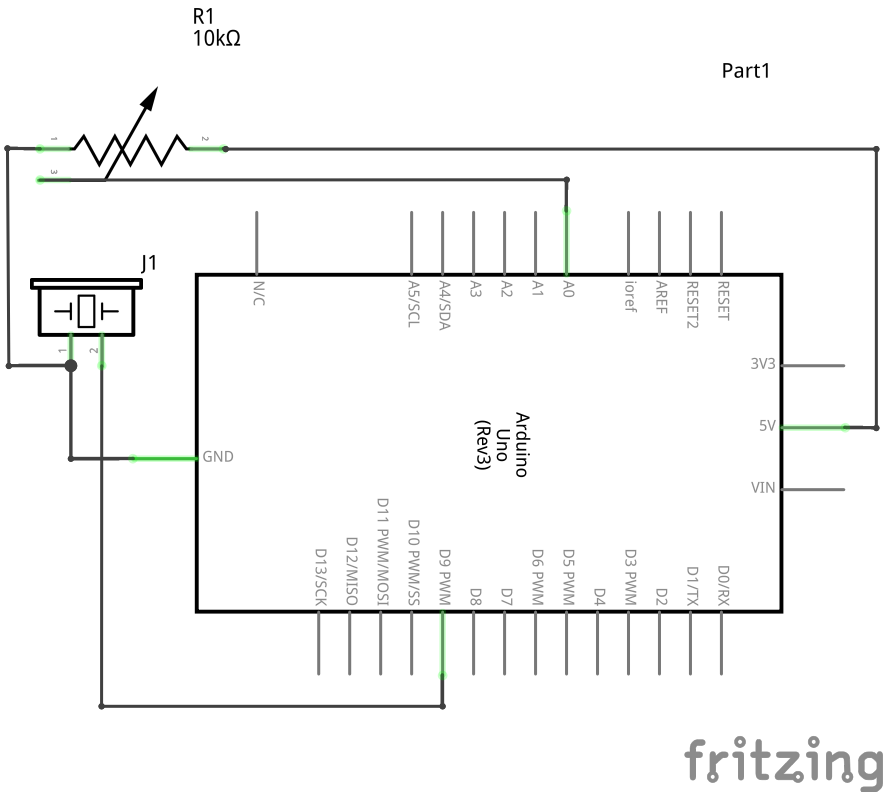


Figure 3.3: The Piezo-Buzzer. If your buzzer has a sticker on top, like on the left buzzer above, do not remove it – otherwise, the buzzer will be too loud.

```
const int buzzerPin = 9; // Use a PWM pin

// We'll set up an array with the notes we want to play
// change these values to make different songs!
// Length must equal the total number of notes and spaces
const int songLength = 18;

// Notes is an array of text characters corresponding to the notes
// in your song. A space represents a rest (no tone)
char notes[] = "cdfda ag cdfdg gf "; // a space represents a rest

// Beats is an array of values for each note and rest.
// A "1" represents a quarter-note, 2 a half-note, etc.
// Don't forget that the rests (spaces) need a length as well.
int beats[] = {1,1,1,1,1,1,4,4,2,1,1,1,1,1,1,4,4,2};

// The tempo is how fast to play the song.
// To make the song play faster, decrease this value.
// It is modified by turning the dial of the potentiometer

#define DEFAULT_TEMPO 150
int tempo= DEFAULT_TEMPO;

int potentiometer= 0; // Current reading from the potentiometer
int last_potentiometer= 0; // Reading from the potentiometer in the last
    iteration of the loop.

void setup()
{
  pinMode(buzzerPin, OUTPUT);
  Serial.begin (9600) ;
}

void loop() {
  potentiometer= analogRead(A0); // value between 0-1023
  Serial.println(potentiometer);
  bool dial_changed= abs(last_potentiometer - potentiometer) > 2;
  if (dial_changed)
  {
    last_potentiometer= potentiometer;
    float tempo_factor= 1.0; // varies from 0.5 to 2
    if (potentiometer >= 512)
    { // Varies from 1 to 2
      tempo_factor= 1.0 + float(potentiometer - 512)/float(1023 - 512);
    }else
    { // Varies from 0.5 to 1
      tempo_factor= 0.5 + float(potentiometer)/float(1023 - 512);
    }
    tempo= int(tempo_factor*DEFAULT_TEMPO);
    play_tune();
```



(a)



(b)

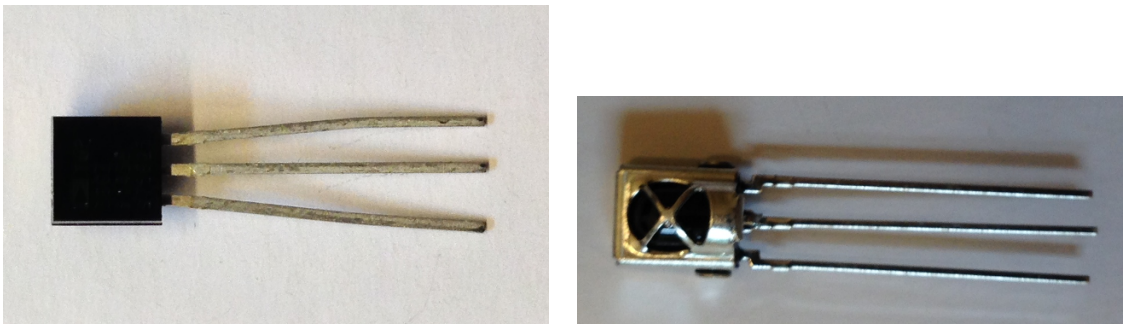
Figure 3.4: The circuit for the piezo-buzzer experiment.

```
  }
}

void play_tune()
{
  int i, duration;
  for (i = 0; i < songLength; i++) // step through the song arrays
  {
    duration = beats[i] * tempo;  // length of note/rest in ms
    if (notes[i] == ' ')          // is this a rest?
    {
      delay(duration);            // then pause for a moment
    }
    else                          // otherwise, play the note
    {
      tone(buzzerPin, frequency(notes[i]), duration);
      delay(duration);            // wait for tone to finish
    }
    delay(tempo/10);              // brief pause between notes
  }
}

int frequency(char note)
{
  // This function takes a note character (a-g), and returns the
  // corresponding frequency in Hz for the tone() function.
  int i;
  const int numNotes = 8;  // number of notes we're storing

  // The following arrays hold the note characters and their
  // corresponding frequencies. The last "C" note is uppercase
  // to separate it from the first lowercase "c". If you want to
  // add more notes, you'll need to use unique characters.

  // For the "char" (character) type, we put single characters
  // in single quotes.

  char names[] = { 'c', 'd', 'e', 'f', 'g', 'a', 'b', 'C' };
  int frequencies[] = {262, 294, 330, 349, 392, 440, 494, 523};

  // Now we'll search through the letters in the array, and if
  // we find it, we'll return the frequency for that note.

  for (i = 0; i < numNotes; i++)  // Step through the notes
  {
    if (names[i] == note)         // Is this the one?
    {
      return(frequencies[i]);     // Yes! Return the frequency
    }
  }
  return(0);  // We looked through everything and didn't find it,
              // but we still need to return a value, so return 0.
}
```

Listing 3.2: Playing a melody on the piezo-buzzer and changing the tempo.

3.3 Temperature Sensor

Warning: The temperature-sensor (Fig. 3.5(a)) has three pins: it is important to keep the polarity in mind, else the sensor will burn-out. The polarity can be found as follows: View the sensor from its flat side (with some text visible), with its pins facing down. Then, the left pin



(a) Temperature Sensor (b) Infra-Red Sensor

Figure 3.5: The temperature sensor and the IR sensor.

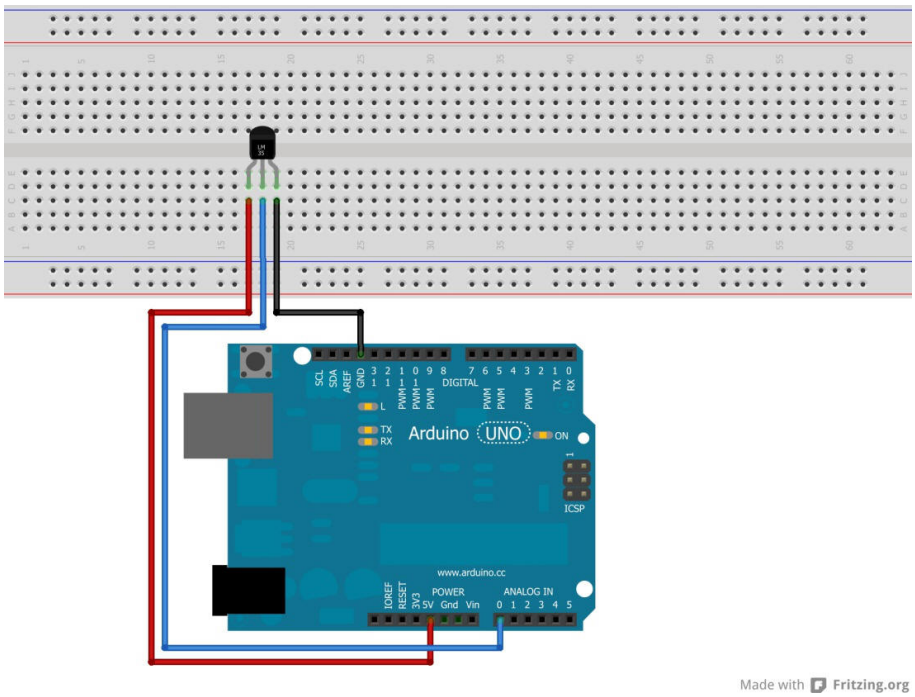


Figure 3.6: The circuit for measuring the temperature.

should be connected to 5V, the right pin to GND, and the middle pin has the temperature-signal and hence should be connected to A0. The performance of the sensor will improve, if the Arduino is connected to an external power-supply (this is not the case in the labs) instead of taking power from the USB connection.

The middle-pin gives out a voltage between $0 - 2$ V. A value of 0 V corresponds to a temperature of $-50°$ C, and a value of 2 V corresponds to a temperature of 150° C. According to the manufacturer, the sensor is accurate to $\pm 2°$ C between $-40°$ to $+125°$ C.

Task 3.4 *Set up the circuit shown in Fig. 3.6 and use the code in Listing 3.3. Open the serial-monitor: the average-temperature (on a moving window of 10 samples) and its standard deviation is printed. Touch the sensor from the top (without touching its pins) and you will see the temperature change.*

Task 3.5 *Why do we use the value* 410 *in the call to the* map *function in Listing 3.3?*

```
int TMP36= A0; // The middle lead is connected to analog-in 0.
int temperature= 0;
```

```
int wait_ms= 20; // wait time betweem measurements in millisec.

#define NR_SAMPLES 10
int samples[NR_SAMPLES]; // array of samples

void setup(){
  Serial.begin(9600);
}

void loop(){
  float sum= 0.0;
  for (int i=0; i< NR_SAMPLES; ++i){
    // map values from range [0, 410] to [-50, 150]
    samples[i]= map(analogRead(TMP36), 0, 410, -50, 150);
    sum += samples[i];
    delay(wait_ms);
  }
  float mean= sum/NR_SAMPLES;
  float sum_square_deviation= 0.0;
  for (int i=0; i< NR_SAMPLES; ++i){
    sum_square_deviation += (samples[i] - mean)*(samples[i] - mean);
  }
  float standard_deviation= sqrt(sum_square_deviation/NR_SAMPLES);
  Serial.print("mean: ");
  Serial.print(mean, 3);
  Serial.print(" C, \t std: ");
  Serial.println(standard_deviation);

}
```

Listing 3.3: Printing the Average Temperature and its Standard Deviation

3.4 Passive Infra-Red (PIR) Sensor for Motion Detection

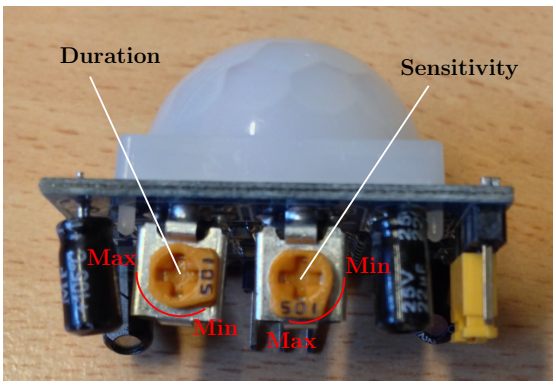
The Passive Infra-Red (PIR) sensor (Fig. 3.7) detects IR light radiating from objects in its field-of-view (FOV) and uses this information to detect motion of humans, animals, and other objects. It is called passiv because it does not emit any energy of its own for detection. When motion is detected, the output pin is set to 5V, which can be easily read by the microcontroller.

By shifting the jumper shown in Fig. 3.7(c), the sensor can be put into two modes:

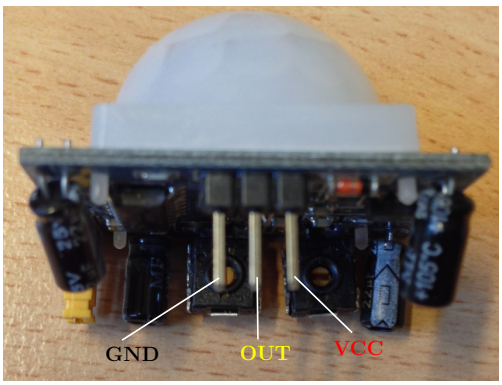
1. If the jumper is as shown in Fig. 3.7(c), i.e. covering the inner two pins, the output signal remains active as long as a motion is detected. This mode is recommended for Arduino projects.

2. If the jumper is covering the outer two pins, the output signal is kept activated for a certain period and then deactivated – even if motion is still being detected. After a certain period of deactivation, if the motion is still being detected, the output is activated again.

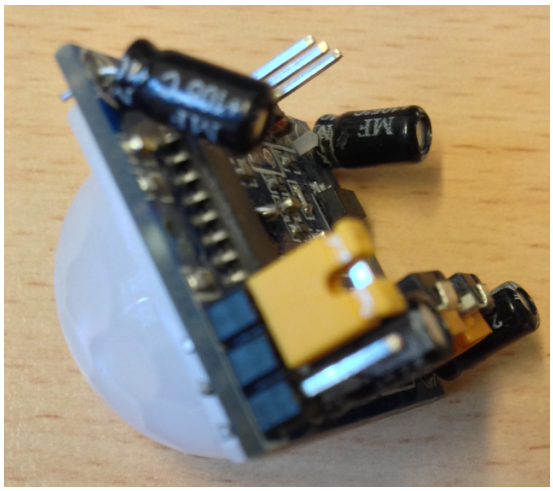
```
int piezo= 5;
int pir= 7;
int motion_status= 0;

void setup(){
  pinMode(piezo, OUTPUT);
  pinMode(pir, INPUT);
}
```



(a) The knobs for tweaking the sensitivity (range) and duration of the output signal.



(b) You can also take off the plastic lens to see the markings for the pins.



(c) The orange jumper covering the inner two pins.

Figure 3.7: The PIR sensor HC-SR501.

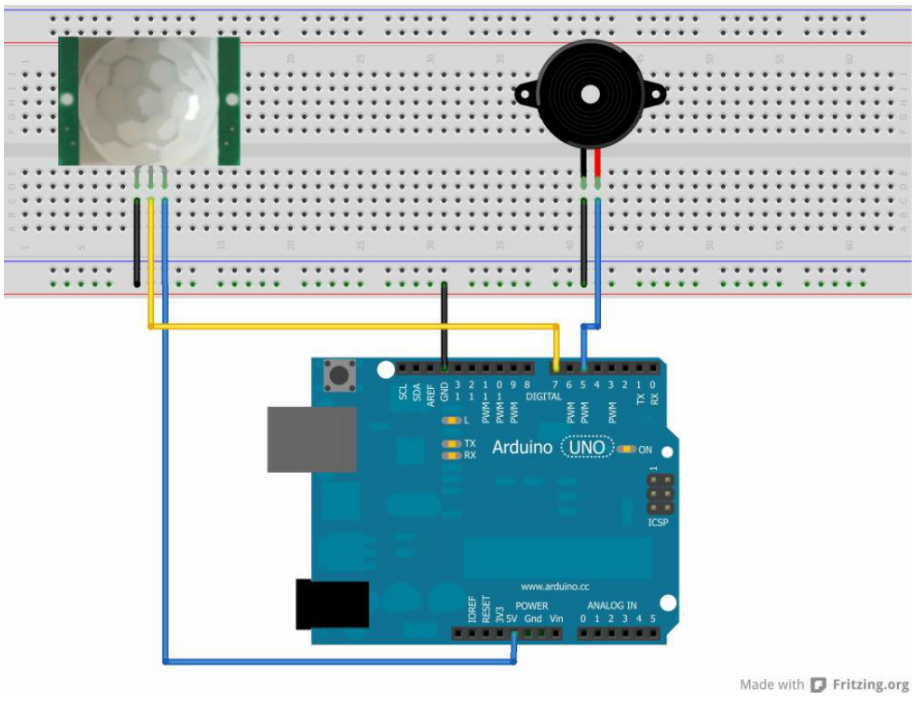


Figure 3.8: The piezo-speaker sounds when a motion is detected.

```
void loop(){
  motion_status= digitalRead(pir);
  if (motion_status == HIGH){
    digitalWrite(piezo, HIGH);
    delay(500);
    digitalWrite(piezo, LOW);
  }else{
    digitalWrite(piezo, LOW);
  }
}
```

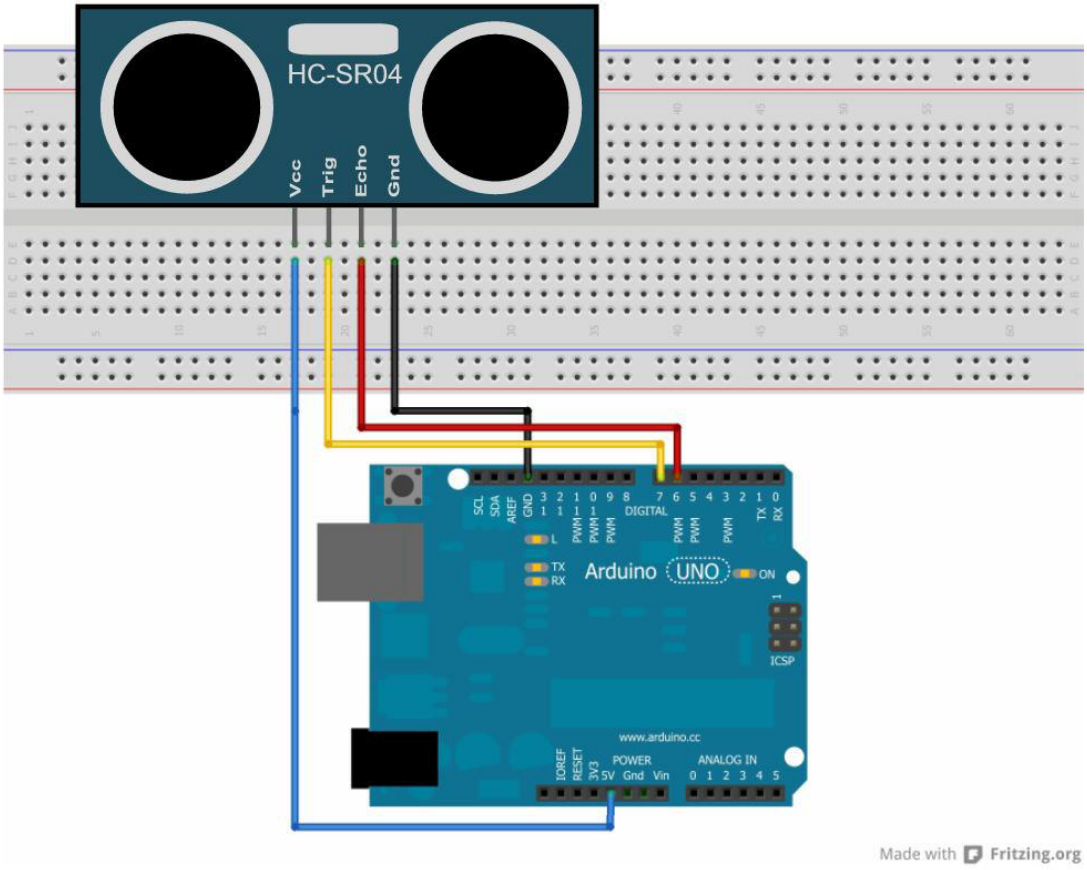
Listing 3.4: The code for the PIR experiment.

Task 3.6 *Set up the circuit as shown in Fig. 3.8 and upload the code shown in Listing 3.4. The piezo-speaker will sound as soon as it detects a motion. Play around to estimate its FOV. Try not to annoy your neighbors.*

3.5 Ultrasonic distance sensor

Learn how to use the basic HC-SR04 distance sensor. The module has 2 ultrasonic transducers (speaker and microphone) and a controller that makes the measurement. It works by sending a series of sound waves (40kHz) and recording the time it takes for them to return after bouncing off an object. The controller then outputs this time interval encoded as the width of a voltage pulse (that can be read by arduino and interpreted as distance).

Now assemble the circuit:



Use the sample code:

```
const int trig = 7;
const int echo = 6;

void setup()
{
  pinMode(trig, OUTPUT);
  pinMode(echo, INPUT);

  digitalWrite(trig, LOW);

  Serial.begin(9600);
}

void loop()
{
  //send an impulse to trigger the sensor start the measurement
  digitalWrite(trig, HIGH);
  delayMicroseconds(15);  //minimum impulse width required by HC-SR4 sensor
  digitalWrite(trig, LOW);

  long duration = pulseIn(echo, HIGH);  //this function waits until the sensor
    outputs the result
  //the result is encoded as the pulse width in microseconds

  //'duration' is the time it takes sound from the transmitter back to the
    receiver after it bounces off an obstacle
  const long vsound = 340;  // [m/s]
  long dist = (duration / 2L) * vsound / 10000L;  // 10000 is just the scaling
    factor to get the result in [cm]

  //REMEMBER: when doing operations with 'long' variables, always put 'L' after
    constants otherwise you will have bugs!

  if (dist > 500L || dist < 2L)
  {
    Serial.println("Invalid range!");
  }
  else
  {
    Serial.print(dist);
    Serial.println(" cm");
  }
  delay(1000);
}
```

sourcecodes/Ultrasound/ultrasound.ino

Task 3.7 *Create a car parking sensor by combining your existing circuit with the circuit from experiment 2 (piezo speaker). Your code must do the following: If the distance is under 100cm it should beep periodically, at a faster rate as the distance gets smaller. If the distance is under 20cm it should beep continuously without interruption.*

3.6 InfraRed Remote Control

IR technology is best suited for line-of-sight, low-range, low-cost, low-power remote control and is widely used as such for TV, Audio Player, Air-Conditioning and many other home devices. The remote has an infrared LED that emits a series of light pulses (with timings given by specific codes) which are then received by a Photodiode (semiconductor that converts light into current) and converted back to digital codes. In this lab we will not deal with the low-level generation of

IR pulses and their decoding, but instead use a general purpose remote and an Arduino library that does the bulk work.

Assemble the receiver circuit as shown in Fig. 3.9, paying attention to the difference between similar looking sensors: Fig. 3.5(b).

3.6.1 Install the Library

1. Find out your sketchbook folder in File ⟶ Preferences ⟶ Sketchbook Location. Now go to this folder and then to its sub-folder called "libraries". Do you see a folder called `IRremote`? If yes, you're done: Go to "Run the Example". If no, follow the remaining steps below.

2. Download the file IRremote.zip to a folder of your choice.

3. In the Arduino IDE click Sketch ⟶ Include Library ⟶ Add .ZIP library, navigate to the folder where you downloaded `IRremote.zip` and select it.

3.6.2 Run the Example

Use the sample code to see the different codes of various buttons:

```
#include <IRremote.h>

const int RX_PIN = 11;
IRrecv irrecv(RX_PIN);  //declare the IR device and specify the receiving pin
decode_results results;  //decode_results is a class type (similar to struct
    typedef)

void setup()
{
  Serial.begin(9600);
  irrecv.enableIRIn();  //start receiving
}

void loop()
{
  if (irrecv.decode(&results))  //returns 1 if decoding is successful and 0
    otherwise
  {
      Serial.println(results.value, HEX);  //print the decoded value on the
    monitor
      irrecv.resume();  //receive the next value
  }

  delay(100);  //you can do something else meanwhile, fast polling is not needed
}
```

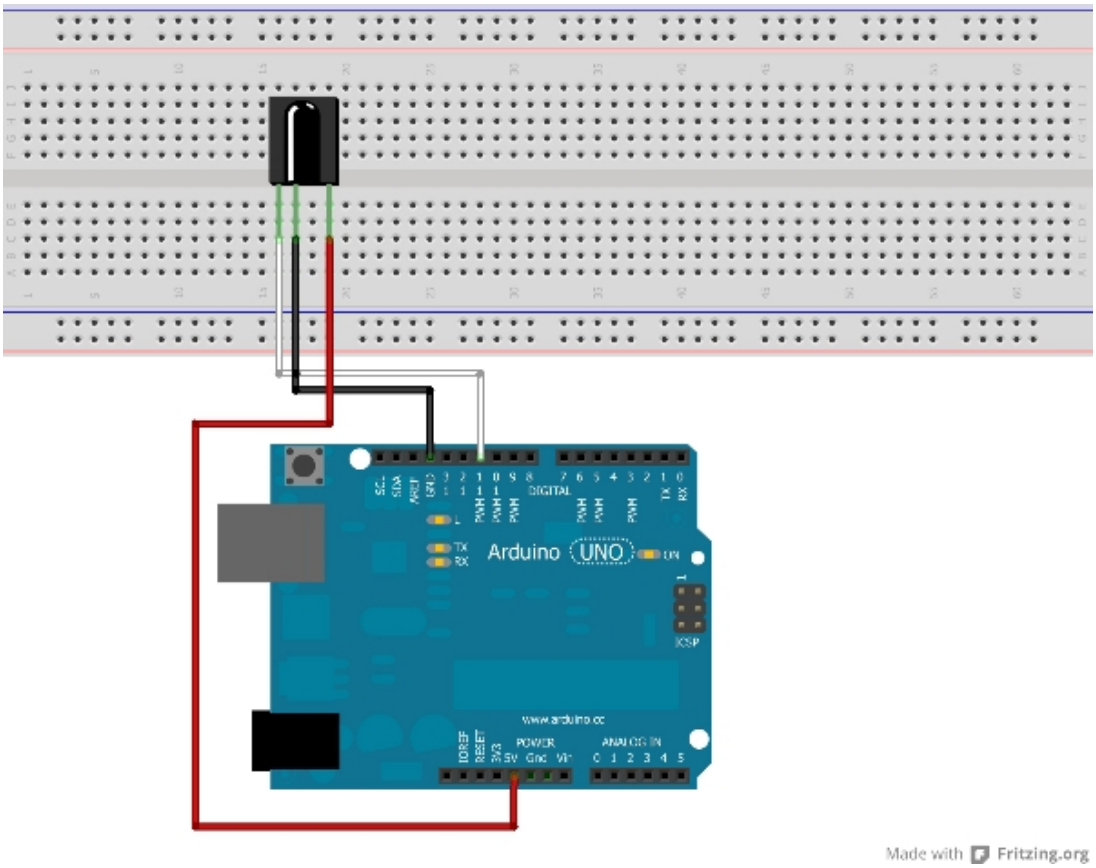
sourcecodes/IRremote/infrared.ino

Task 3.8 *Record the codes of the following 4 buttons:* $CH-$, $CH+$, CH, $PREV|<<$

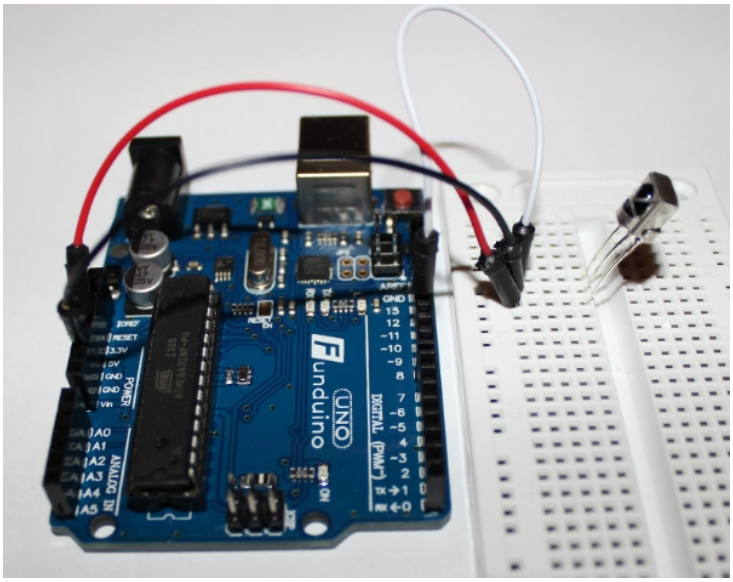
3.7 LCD display (Optional)

Note: This an optional experiment: You can skip it if you run out of time.

LCD technology consists of a liquid crystal placed between 2 polarization filters that can let light pass through or block it depending on the CHtage applied to it, therefore making the pixel appear white or black. To get a color LCD, an additional RGB filter is applied so that every



(a)



(b) Note the order of the pins.

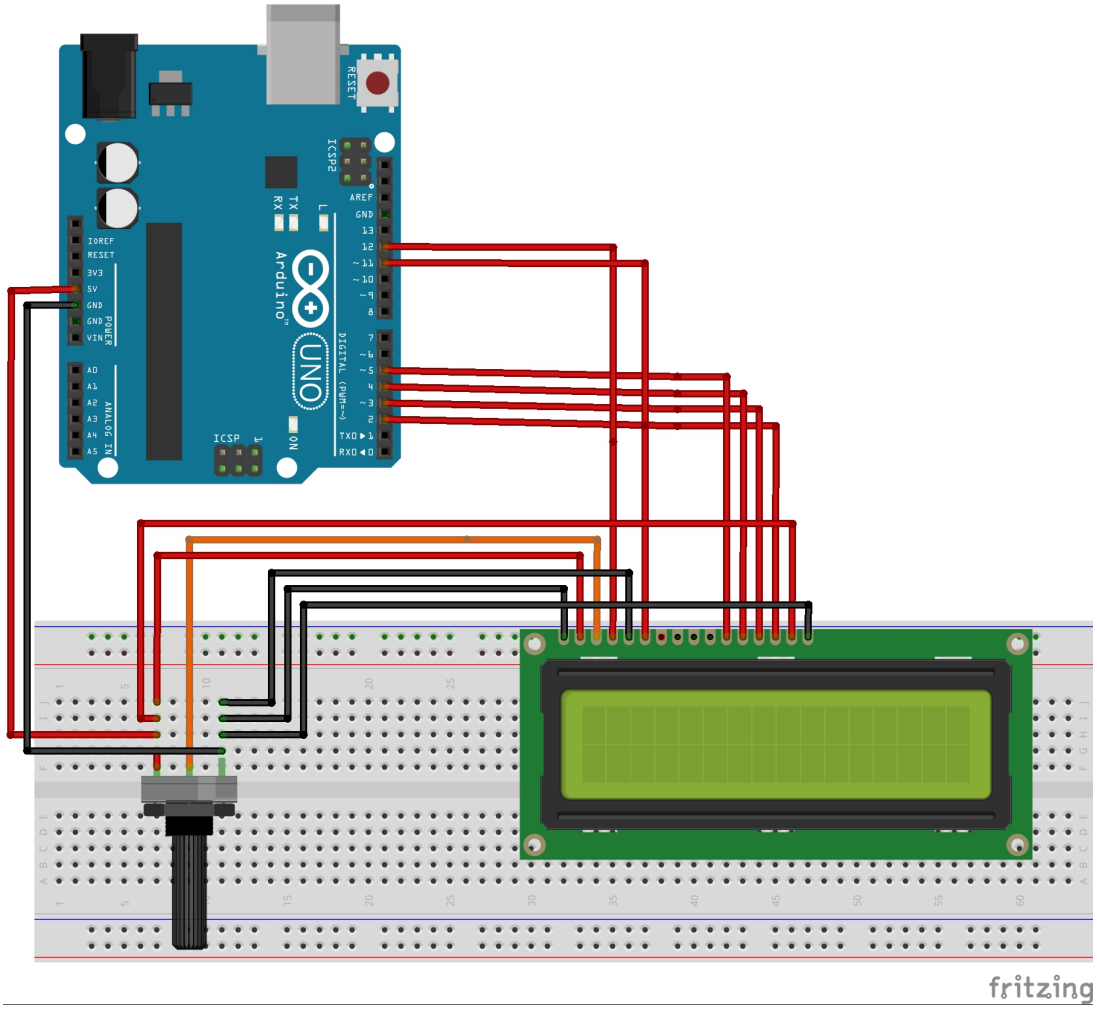
Figure 3.9: IR sensor circuit

3 pixels become 1 RGB color pixel. Note that unlike LED displays (OLED, AMOLED, etc.), LCD's need either ambient light or a backlight (typically LED panel) to be visible.

Since every pixel that needs to be addressed must have its own electrical connection, even a low resolution display would require a huge number of external connections (tens or hundreds). Therefore displays typically have an Integrated Circuit directly on the glass that converts the incoming serial data (through a small number of external connections) to parallel data for all the individual pixels.

In this lab we are going to use a $16 \times 2 = 32$ character display, capable of displaying any ASCII char in each of the 32 addressable areas. This is somewhat simpler than a full-matrix display which has every pixel addressable and is capable of custom graphics.

Assemble the circuit:



Get familiar with the LCD commands:

```
#include<stdio.h>
#include<LiquidCrystal.h>

#define ROW_LEN 16

LiquidCrystal lcd (12 , 11 , 5 , 4 , 3 , 2) ; // specify the arduino pins that
    connect to the LCD
// Note: You can also use analog-in pins for LCD, so e.g.
// you can replace 2 by A2 above (and change the corresponding wiring).
```

```
char string1[ROW_LEN + 1]; // The last char is for null-termination with \0
char string2[ROW_LEN + 1]; // The last char is for null-termination with \0

void printLCD(void){
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print(string1);
  lcd.setCursor(0, 1);
  lcd.print(string2);
  delay(4000);
}

void setup ()
{
// initialize and specify the lcd type ( LiquidCrystal is a generic library for
    various LCD    s  )
  lcd.begin(ROW_LEN , 2) ; // the one we use has 16 characters x 2 rows
}

void loop ()
{
// display coordinates start from top left and increase towards bottom right
// before printing we can set the cursor where we want
  lcd.setCursor (0 , 0) ; // first character on the first row
  lcd.print ("JacobsUniversity") ;
// should fit exactly on the first row

  delay (1000) ;
  lcd.clear () ;
// clear everything from the LCD

// Set the cursor to column 4 and row 1
// we can also write without setting the cursor first
// the cursor will automatically increment by 1 ( column )
  lcd.setCursor (4 , 1) ;
  lcd.print ( 'I') ;
  delay (250) ;
  lcd.print ( 'M') ;
  delay (250) ;
  lcd.print ( 'S') ;
  delay (250) ;
  lcd.print ( ' ') ;
  delay (250) ;
  lcd.print ( 'L') ;
  delay (250) ;
  lcd.print ( 'a') ;
  delay (250) ;
  lcd.print ( 'b') ;

  delay (500) ;
  lcd.clear () ;

// Let us now print different kinds of numbers (ints, longs, floats),
// which we may have received as samples from sensors etc.

  int i= -20703;
  unsigned int ui = 65535;
  snprintf(string1, ROW_LEN + 1, "A: %7d", i);
  snprintf(string2, ROW_LEN + 1, "B: %7u", ui);
  printLCD();

  unsigned long ul= 2147383648L;
  snprintf(string1, ROW_LEN + 1, "C: %lu", ul);
```

```
  snprintf(string2, ROW_LEN + 1, "D: %lx", ul); // print it in hexadecimal (base
     16)
  printLCD();

  float f= 3.1415926;
  // In Arduino, you cannot sprintf a float
  // snprintf(string1, ROW_LEN + 1, "E: %5.3f", f); // will not work
  // snprintf(string2, ROW_LEN + 1, "F: %e", f);    // will not work

  // Instead convert it to an int and then display
  long int lf= (long int)(f*100000);
  snprintf(string1, ROW_LEN + 1, "E: %ld", lf);
  snprintf(string2, ROW_LEN + 1, "F: %lx in HEX", lf); // print it in
    hexadecimal (base 16)
  printLCD();

  lcd.clear();

}
```

sourcecodes/LCD/lcd_sprintf_ino.ino

Task 3.9 *Turn the potentiometer and explain its purpose.*

Task 3.10 *Read the documentation of snprintf, and how the formating-string is specified. What does the format* `"%7u"` *mean? Why did we use* ***`snprintf`*** *and not the usual* ***`sprintf`****?*

Chapter 4

Lab 4

In this lab, we are going to connect several different actuators and motion sensors to the Arduino. The list of actuators and their data-sheets can be accessed here.

Info: In the last two experiments involving the accelerometer and the IMU, half of the groups will start with the accelerometers and the rest with the IMUs — afterwards the groups will exchange the sensors.

4.1 Servomotor

A servomotor is an essential device for automated action of various mechanisms. Small servomotors are widely used for remote controlled airplanes, cars, robots and other toys while bigger servomotors are used for power steering in automobiles, control of valves in plants and many other systems.

Servomotors usually have a shaft that can precisely rotate at a specific angle, but linear servos also exist that have an output rod sliding back and forth. They are compact objects integrating an electric motor and a (fixed) gearbox, a sensing potentiometer on the output shaft and power/control electronics that adjust the motor power and direction such that the output shaft reaches the desired angle.

In this lab we are going to use a micro servo as shown in Fig. 4.1. In the actual servo given to you, the colors of the leads are slightly different. The brown lead is for the GND, red for 5V as usual, and orange for the control-signal (shown as yellow in Fig. 4.1). Extend the leads using the jumpers.

Warning: Do not restrict the rotation of the motor-shaft in any way: This may lead to high currents. Run the example code in Listing 4.1.

```
#include <Servo.h>
Servo s;  //create servo object;

void setup()
{
  s.attach(8);  //control signal on pin 8
}

void loop()
{
  //now we can control the servo with write(angle)
  //angle of the servo 0-180 degrees
  //90 degrees = servo is centerd

  s.write(0);
  delay(3000);

  s.write(45);
```

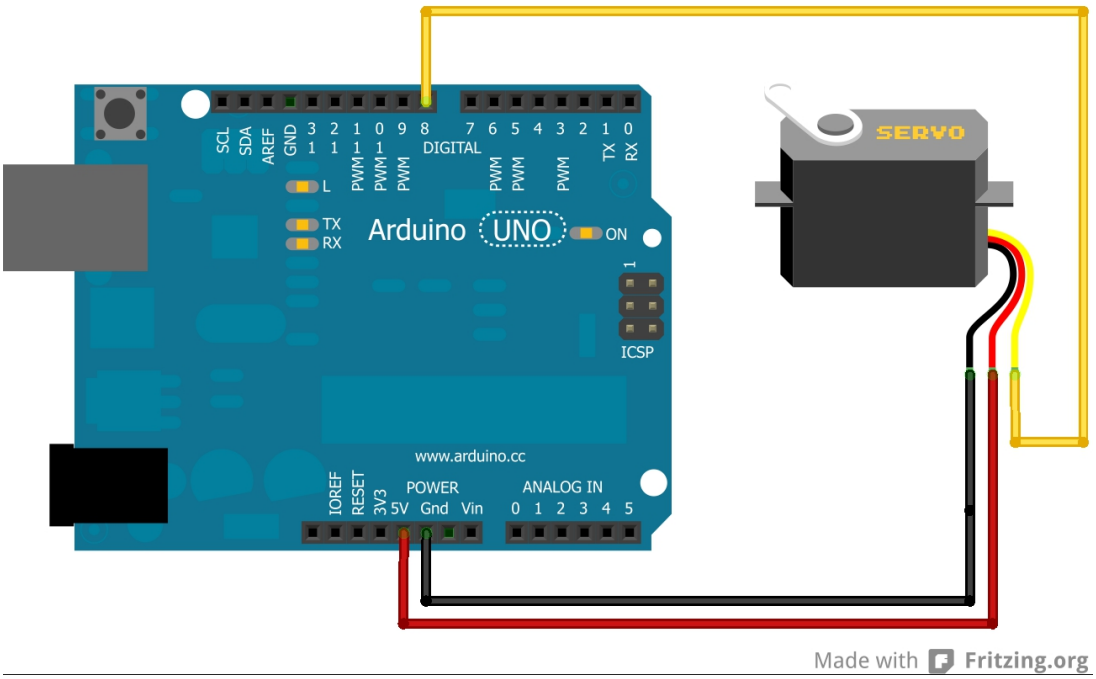


Figure 4.1: A micro-servo.

```
  delay(3000);

  s.write(90);
  delay(3000);

  s.write(135);
  delay(3000);

  s.write(180);
  delay(3000);
}
```

sourcecodes/Servo/servo.ino

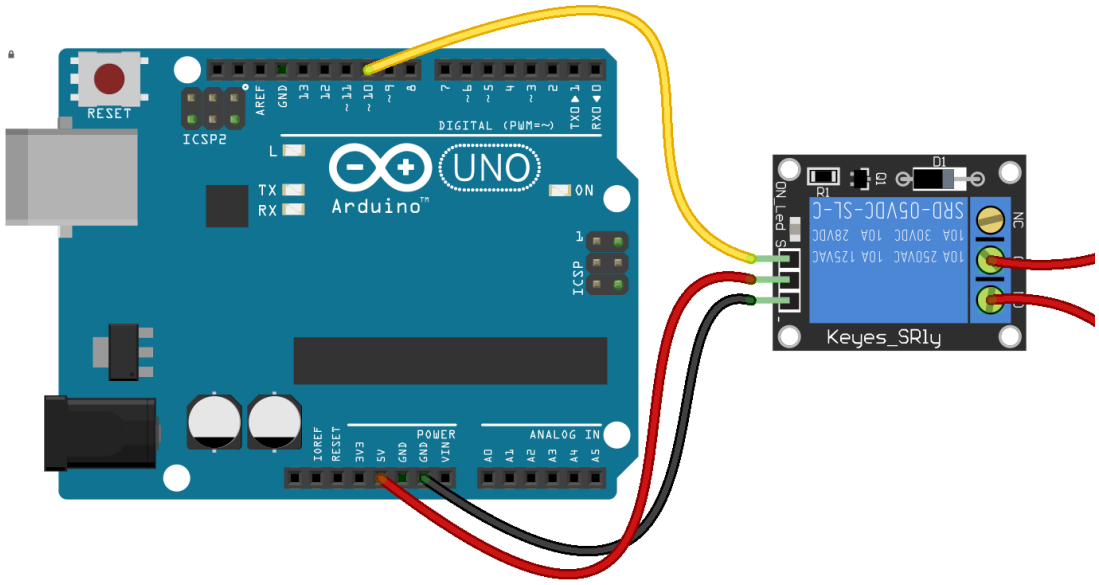
Task 4.1 *Which two PWM pins cannot be used on the Uno if you're using the servo library? Refer to the library documentation.*

Task 4.2 *Change the code to stop the servo at* 0°. *Remove the power from the motor without disturbing its shaft. Now, to observe the servo rotation more clearly, attach one of the unsymmetrical white plastic flanges (also shown attached in Fig. 4.1) to the servo shaft. There is no need to screw it in. Connect the power to the motor again.*

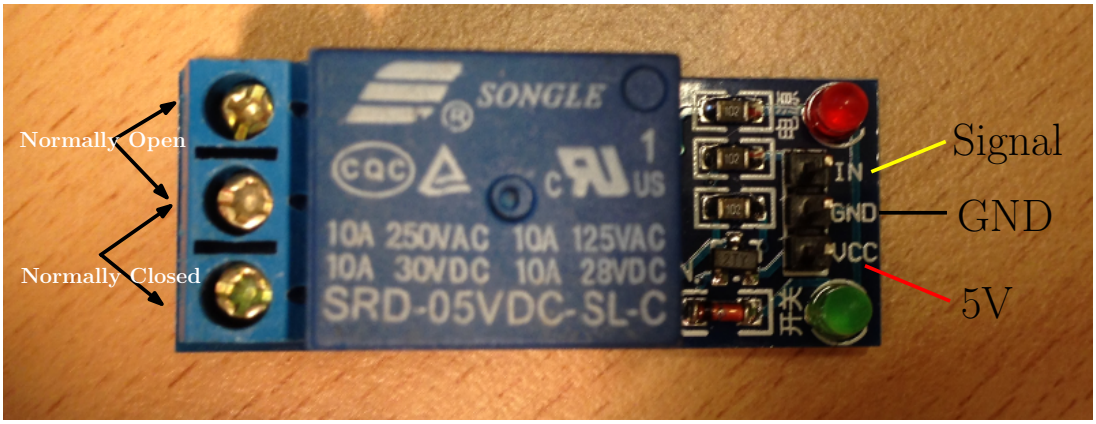
Task 4.3 *Make the servo transition smoothly from* 0 *to* 180° *and back in a continuous cycle. Adjust the period of the cycle to* 6 *seconds.*

4.2 Power Switching Relay

A relay is an electrically controlled mechanical switch and is used to turn relatively large loads (currents) on/off, e.g. a large motor or a lamp. The power to these large loads is supplied externally (not from Arduino). The Arduino board only switches the load on or off through the relay: otherwise, the two circuits are independent. Unlike transistors, relays can NOT be



(a)



(b)

used for high frequency switching (PWM) due to the slow response time owing to inertia and mechanical wear.

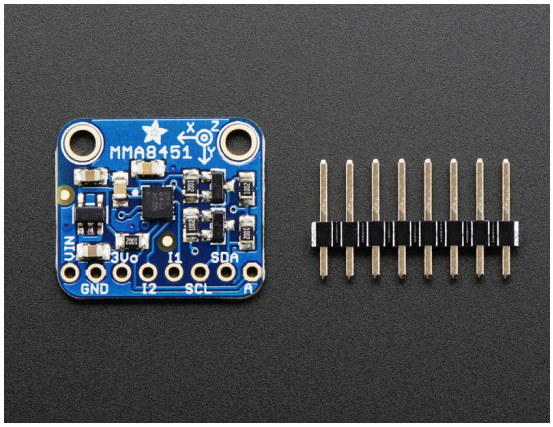
Look at Fig. 4.2(b): usually the two terminals marked "Normally Closed" are connected, and the ones connected "Normally Open" are open.

Task 4.4 *Test using a multimeter in "Continuity Check" mode to see if the configuration on your relay is as shown in Fig. 4.2(b). The relay need not be connected to Arduino.*

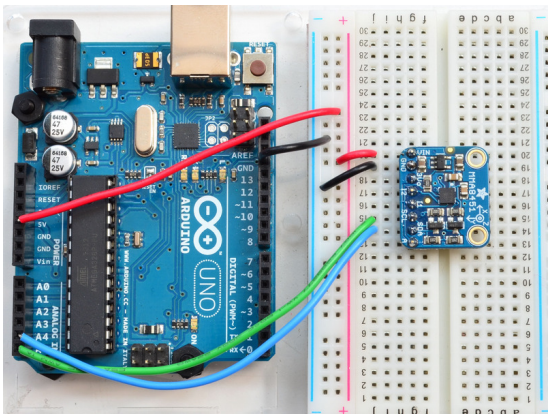
Task 4.5 *Proceed as follows:*

1. *Write a sketch which sets the pin 10 (setup in digital-output mode) HIGH for 5 seconds and LOW for 5 seconds. Upload it to Arduino, and disconnect it from the USB.*

2. *Now make the circuit: connect the pin marked "Signal" and shown with a yellow line to the Arduino pin 10 in digital output mode. The other two pins should be connected to Arduino 5V and GND as shown. Connect the Arduino to the USB.*



(c) The break-out board for MMA8451 and the supplied headers. Source: Link



(d) The circuit for running the demo code. Source: Link

Figure 4.2: The Adafruit MMA8451 Accelerometer Breakout. Note that the XYZ axes directions are clearly marked.

3. *When pin 10 is HIGH, the relay switches on, you will hear a click and the green LED switches on. The 'Normally Closed" terminals are now open and the "Normally Open" closed: Test this using your multimeter in "Continuity Check" mode.*

4.3 Accelerometer for finding Orientation

4.3.1 The Accelerometer and the IMU

Initially half of the groups will be given the blue accelerometer (Fig. 4.2(c)) and the other half the green IMU (Fig. 4.5). Do not confuse between the two.

STOP: The groups which received the green IMU initially can skip ahead to Sec. 4.4 and come back to this section later. After finishing your first experiment, swap your IMU with an accelerometer from another group (or vice versa).

4.3.2 Introduction

An accelerometer measures the g-force acceleration, i.e. acceleration in relation to a state of free-fall. It is the acceleration which is felt by the body as weight, as opposed to a state of free-fall which is felt as "weightlessness." At rest on a table, the g-force on an accelerometer is caused by the normal force exerted on it by the surface of the table which keeps it from falling freely — hence, the g-force shown by the accelerometer is pointed in the opposite direction of gravity (up instead of down). If there is acceleration in direction orthogonal to the gravity direction, e.g. when a car or roller-coaster accelerates fast and you feel pressed against the seat, it is also an example of a g-force. In the example, it is pointed in the direction of the acceleration of the car.

Most modern accelerometers are micro-electro-mechanical systems (MEMS). The MMA8451 is a miniature accelerometer from Freescale with a 14-bit ADC resolution. You can set the acceleration range to ±2g, ±4g, or ±8g. We are using a break-out board for this sensor (Fig. 4.2(c)) which facilitates its use in Arduino projects. To use the break-out board with a bread-board the supplied headers are soldered on the board – this has been done for you.

4.3.3 Download the Libraries

In the Arduino IDE, check under the menu-item Sketch ⟶ Include Library, under the category "Recommended Libraries". If this category is entirely absent or if you do not see "Adafruit MMA8451" and "Adafruit Unified Sensor" listed there, proceed as follows:

- Download the file `Adafruit_MMA8451_Library-master.zip` to a folder of your choice. In the Arduino IDE, go to the menu item Sketch ⟶ Include Library ⟶ Add .ZIP Library, and in the file-dialog, select the zip file.
- Download the file `Adafruit_Sensor-master.zip` to a folder of your choice. In the Arduino IDE, go to the menu item Sketch ⟶ Include Library ⟶ Add .ZIP Library, and in the file-dialog, select the zip file.
- Restart the Arduino IDE. Now you should see "Adafruit MMA8451" and "Adafruit Unified Sensor" listed under "Recommended Libraries".

4.3.4 Make the Circuit

Now, insert the headers on which the sensor is mounted to the breadboard. Hold the sensor break-out board from the sides, align to the holes properly and press from the headers side:
Warning: Do not use excessive force.
The break-out board has the following marked pins, which you should connect as instructed below.
Hint: Use longer jumpers so that you will be able to tilt the breadboard easily later.

Vin. The chip uses 3 VDC, hence, the breakout has a regulator on board which takes in 3-5VDC and safely converts it to 3 VDC. Connect this pin to the 5V pin of your Arduino.

GND. Connect it to the GND of the Arduino.

3Vo. This is the 3.3V output from the regulator. We will not use it.

I2, I1. These are for interrupts: We will not need them.

SCL. I2C clock pin, connect to the I2C clock line of the Arduino. On Uno, this is **A5**. On some variants, a separate notch marked "SCL" is also provided.

SDA. I2C data pin, connect to the I2C data line of the Arduino. On Uno, this is **A4**. On some variants, a separate notch marked "SDA" is also provided.

A. This can be used to change the I2C address from 0x1C to 0x1D if you connect it to the 3Vo pin. We will not need it.

4.3.5 Run the Demo

In the Arduino IDE, open File ⟶ Examples ⟶ Adafruit MMA8451 Library ⟶ MMA8451 demo, compile, and upload to Arduino. Now open the serial monitor, you should see an output similar to that shown in Fig. 4.3.

There's three lines of output from the sensor in every sample (this description is from here):

- `X: -24 Y: 392 Z: 4014`
  This is the "raw count" data from the sensor, its a number from -8192 to 8191 (14 bits) that measures over the set range. The range can be set to 2g, 4g or 8g.

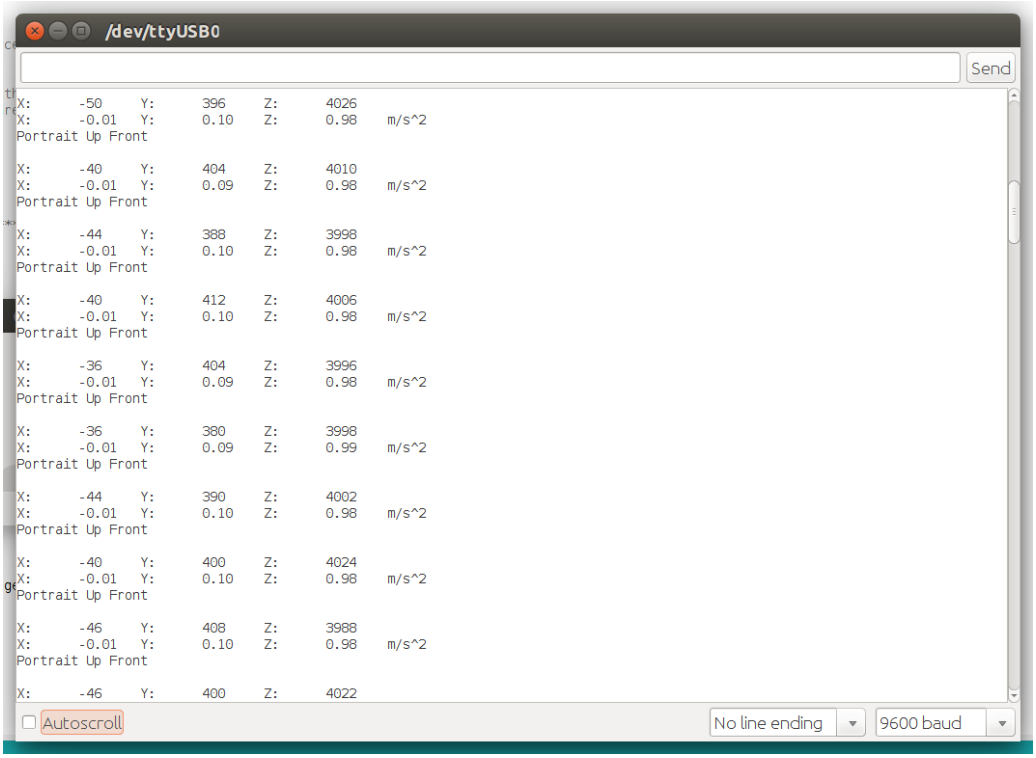


Figure 4.3: The output of the MMA8451 demo.

- `X: -0.01 Y: 0.10 Z: 0.98 m/s^2`
  This is the Adafruit_Sensor'ified nice output which is in m/s*s, the SI units for measuring acceleration. No matter what the range is set to, it will give you the same units, so this output is preferable to the raw one.

- `Portrait Up Front` This is the output of the orientaiton detection inside the chip. Since inexpensive accelerometers are often used to detect orientation and tilt, this sensor has it built in. The orientation can be Portrait or Landscape, then Up/Down or Left/Right and finally tilted forward or tilted back. Note that if the sensor is tilted less than 30 degrees it cannot determine the forward/back orientation. The various values returned from `mma.getOrientation()` are:

```
0: Portrait Up Front
1: Portrait Up Back
2: Portrait Down Front
3: Portrait Down Back
4: Landscape Right Front
5: Landscape Right Back
6: Landscape Left Front
7: Landscape Left Back
```

Task 4.6 *Proceed as follows:*

1. *Tilt the breadboard different ways: are the signs of the XYZ accelerations as you expected from the directions marked on the breakout board?*

2. *Get all 8 possible orientation outputs from the sensor by tilting in various directions.*

3. *There is something fishy in the units being printed out in the second line, e.g.* `X: -0.01 Y: 0.10 Z: 0.98 m/s^2`
   Is it really in `m/s^2`*? Why or why not?*

4.3.6 Show the Output on Processing

Task 4.7 *Proceed as follows:*

1. *Download the code Listing 4.1 from the lab web-site.*
2. *Look at the function* `parseStringForAccelerations` *in the code Listing 4.1 which runs on the* ***processing side****. What format of data does it expect? It expects that XYZ accelerations are in separate lines starting with* `X:`, `Y:`, `Z:` *respectively.*
3. ***On the Arduino side****, modify the code of the demo program, (Since the example is read-only, the IDE will ask you to save your modification at a new location) so that the Processing program in Listing 4.1 is able to read what it writes on the serial port. This involves commenting out all* `Serial.print` *statements, except the ones on lines 58-61. The latter should be modified so that they print XYZ accelerations on separate lines starting with* `X:`, `Y:`, `Z:` *respectively without units. Compile and upload it to Arduino as usual. Check the Serial Monitor to check what is being printed. Then close the Serial Monitor.*
4. ***On the processing side****, fill in code in all places in Listing 4.1 where you see the comment* `// Fill-in`*. You may have to recall the Accelerometer section from the GIMS-1 slides, especially the formulae for tilt determination. The yaw cannot be measured and hence can be set to* 0*.*
5. *Now run the GUI: It should appear as in Fig. 4.4. For seeing the correspondance between the accelerometer orientation and the box orientation, initially align the accelerometer (mounted on the breadboard) such that the axes schematic printed on it is on the left-bottom corner, i.e. X arrow is pointing right, and Y arrow is pointing away from you.*
6. *Explain the function* `parseStringForAccelerations` *to your supervisor. Is the state-variable* `nextToRead` *really needed?*

```
import processing.serial.*;

int lf = 10;    // Linefeed in ASCII
String receivedString = null;
Serial myPort;  // Serial port you are using
float[] acceleration= {0., 0., 0.};
float pitch, roll; // in radians

void setup() {
  size(640, 360, P3D);
  noStroke();
  colorMode(RGB, 1);

  myPort = new Serial(this, "COM1", 9600);
  // Replace "COM1" by the actual Arduino port.
  myPort.clear();
}


void draw_rgb_cube() {
  background (0.4, 0.4, 0.4) ;
  scale(100);
  beginShape(QUADS);
```

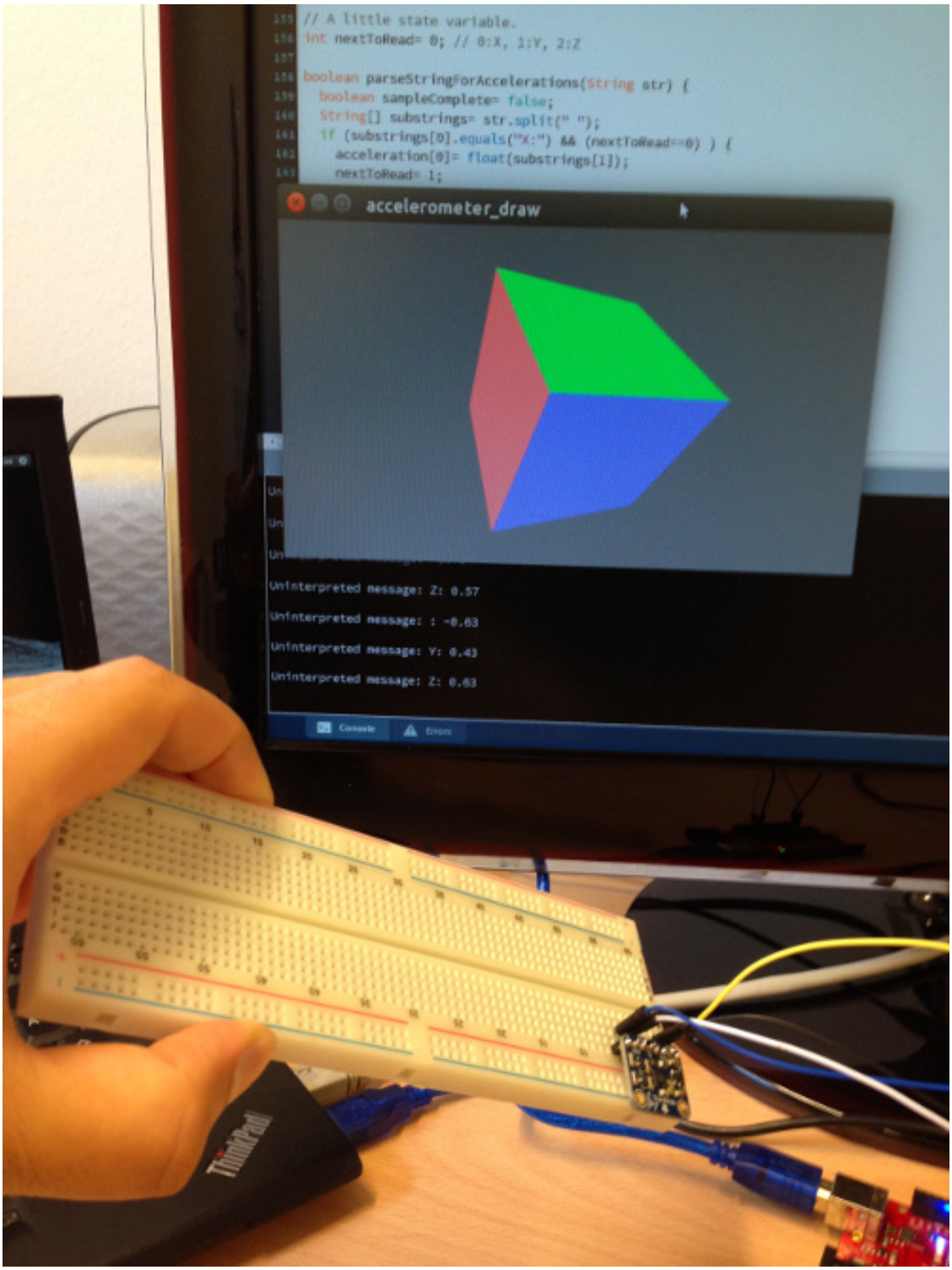


Figure 4.4: Connecting the accelerometer output to a processing GUI.

```
  // Front (As +Z is coming out of the screen) face: Red
  fill(0, 0, 1);
  vertex(-1, 1, 1);
  fill(0, 0, 1);
  vertex( 1, 1, 1);
  fill(0, 0, 1);
  vertex( 1, -1, 1);
  fill(0, 0, 1);
  vertex(-1, -1, 1);

  // Right face (As +X is on the right): White
  fill(1, 0, 0);
  vertex( 1, 1, 1);
  fill(1, 0, 0);
  vertex( 1, 1, -1);
  fill(1, 0, 0);
  vertex( 1, -1, -1);
  fill(1, 0, 0);
  vertex( 1, -1, 1);

  // Back face (-Z): Light Blue
  fill(0.5, 0.5, 1);
  vertex( 1, 1, -1);
  fill(0.5, 0.5, 1);
  vertex(-1, 1, -1);
  fill(0.5, 0.5, 1);
  vertex(-1, -1, -1);
  fill(0.5, 0.5, 1);
  vertex( 1, -1, -1);

  // Left face (-X): Pink
  fill(1, 0.5, 0.5);
  vertex(-1, 1, -1);
  fill(1, 0.5, 0.5);
  vertex(-1, 1, 1);
  fill(1, 0.5, 0.5);
  vertex(-1, -1, 1);
  fill(1, 0.5, 0.5);
  vertex(-1, -1, -1);

  // Bottom face (+Y is pointing down the screen): Green
  fill(0, 1, 0);
  vertex(-1, 1, -1);
  fill(0, 1, 0);
  vertex( 1, 1, -1);
  fill(0, 1, 0);
  vertex( 1, 1, 1);
  fill(0, 1, 0);
  vertex(-1, 1, 1);

  // Top face (-Y): Light Green
  fill(0.5, 1, 0.5);
  vertex(-1, -1, -1);
  fill(0.5, 1, 0.5);
  vertex( 1, -1, -1);
  fill(0.5, 1, 0.5);
  vertex( 1, -1, 1);
  fill(0.5, 1, 0.5);
  vertex(-1, -1, 1);

  endShape();
}
```

```
void draw() {
  while (myPort.available() > 0) {
    receivedString = myPort.readStringUntil(lf);
    if (receivedString != null) {
      boolean newSample= parseStringForAccelerations(receivedString);
      if (newSample) {
        /* Uncomment to see if the values received are plausible. */
        /*
        print("Got acceleration: ");
        print(acceleration[0]);
        print(", ");
        print(acceleration[1]);
        print(", ");
        print(acceleration[2]);
        println(".");
        */

        // ax, ay, az which are the components of the
        // unit-vector in the direction of g
        // in the accelerometer basis.
        float acc_norm= sqrt(acceleration[0]*acceleration[0] + acceleration[1]*
   acceleration[1] + acceleration[2]*acceleration[2]);
        // Compute normalized acceleration values.
        float ax= /* fill-in */;
        float ay= /* fill-in */;
        float az= /* fill-in */;

        if (abs(abs(ax)-1) <= 1.0e-1){
          println("Pitch getting too close to Gimbal lock!");
        }

        // **Hack: Processing XYZ basis is unfortunately LEFT-handed:
        az= -az;

        // Fill-in: compute roll in (-pi, pi] and pitch in [-pi/2, pi/2]
        // as functions of ax, ay, az using the GIMS-1 slides.
        pitch= /* fill-in as a function of ax */;
        roll= /* fill-in as a function of ay, az, and pitch */;

        // Draw the rotated object centered.
        translate ( width /2.0, height /2.0, -50) ;

        // Use rotateX and/or rotateY to rotate the object by
        // the computed pitch and roll.
        // With body-fixed rotations convention.
        rotateY(pitch);
        rotateX(roll);

        draw_rgb_cube();
      }
    }
  }
  myPort.clear();
}

// A little state variable.
int nextToRead= 0; // 0:X, 1:Y, 2:Z

boolean parseStringForAccelerations(String str) {
  boolean sampleComplete= false;
  String[] substrings= str.split(" ");
  if (substrings[0].equals("X:") && (nextToRead==0) ) {
```

```
    acceleration[0]= float(substrings[1]);
    nextToRead= 1;
  } else if (substrings[0].equals("Y:")  && (nextToRead==1) ) {
    acceleration[1]= float(substrings[1]);
    nextToRead= 2;
  } else if (substrings[0].equals("Z:")  && (nextToRead==2) ) {
    acceleration[2]= float(substrings[1]);
    nextToRead= 0;
    sampleComplete= true;
  } else {
    print("Uninterpreted message: ");
    println(receivedString);  // Prints status messages
    nextToRead= 0;
  }
  return sampleComplete;
}
```

Listing 4.1: Fill in the missing parts of this Processing code.

4.4 Inertial Measurement Unit (IMU)

In this experiment, we will connect an IMU to the Arduino. An IMU has essentially three sensors on board:

1. **An Accelerometer**: We are familiar already with this sensor from the last experiment and from the GIMS-I lecture. Essentially, it can provide us with the tilt (pitch, roll), but it cannot provide us with the yaw angle (also called heading or bearing angle). Another disadvantage is that the pitch and roll angles are noisy, since the accelerometer is actually measuring the g-forces and not the tilt directly. On the positive side, the pitch and roll angles found do not *drift* if you keep the sensor stationary.

2. **A Gyro**: This sensor measures the angular-velocity components in the gyro frame: You will learn about it in the Kinematics part of the GIMS-I lecture. The angular-velocity components can be integrated to compute all three rotational angles: yaw, roll, and pitch. The values look quite smooth in comparison to those from the accelerometer. However, since they are based on integration, the noise also gets integrated, and hence, the angular values may drift, even when the sensor is stationary.

3. **A Magnetometer**: You can think of a magnetometer as a compass, which gives the heading (i.e. yaw) of the sensor based on the magnetic field of the earth. This is relatively stable but may be affected by electromagnetic disturbances in the environment.

The IMU combines the advantages of all of the three sensors above by fusing their values. You will learn the details of sensor-fusion in the IMS Automation course. In some IMUs an additional sensor, a barometer, for measuring the pressure (and hence the altitude) is included as well.

4.4.1 Pololu MinIMU-9 v3

In this experiment, we will use the Pololu MinIMU-9 v3 (Refer to Fig. 4.5), a compact board that combines STs L3GD20H 3-axis gyroscope and LSM303D 3-axis accelerometer and 3-axis magnetometer to form an inertial measurement unit (IMU). It draws about 6 mA and can be connected to an Arduino via an I2C bus. All three sensors: the accelerometer, the gyro, and the magnetometer, provide a 16-bit reading per axis. The sensitivity range of each of these sensors can be set in code. The sensitivity ranges are:

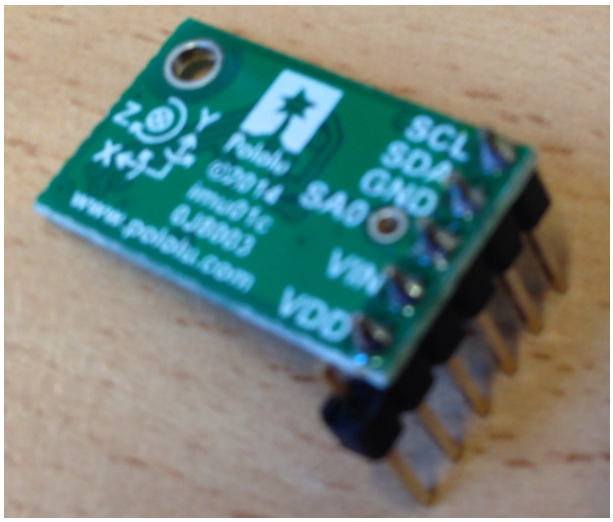


Figure 4.5: The Pololu MinIMU-9 v3.

- Accelerometer: ± 2, ± 4, ± 6, ± 8, or ± 16 g
- Gyro: $\pm 245^\circ/s$, $\pm 500^\circ/s$, or $\pm 2000^\circ/s$
- Magnetometer: ± 2, ± 4, ± 8, or ± 12 Gauss.

Warning: As seen in Fig. 4.5, there is an additional header attached to the left which is not connected anywhere. This can be ignored. The next pin is marked VDD: We will not use this in this experiment, so do **not** connect it to 5V.

From the product page: "The nine independent rotation, acceleration, and magnetic readings (sometimes called 9DOF) provide all the data needed to make an *attitude and heading reference system* (**AHRS**). With an appropriate algorithm, a microcontroller or computer can use the data to calculate the orientation of the MinIMU board. The gyro can be used to very accurately track rotation on a short timescale, while the accelerometer and compass can help compensate for gyro drift over time by providing an absolute frame of reference. The respective axes of the two chips are aligned on the board to facilitate these sensor fusion calculations."

4.4.2 Install the Libraries

Perform the following steps:

1. Find out the location of your Arduino sketchbook folder by opening File $\longrightarrow$ Preferences.
2. In the Arduino sketchbook folder, check if under the sub-folder "libraries' (create it, if it does not exist) you see the sub-folders L3G, LSM303, and MinIMU9AHRS. If yes, skip the rest of the steps.
3. Install Pololu L3G library: Download the zip file. Unzip it in a temporary folder and then copy the folder L3G into your Arduino libraries folder (step 1).
4. Install Pololu LSM303 library: Download the zip file. Unzip it in a temporary folder and then copy the folder LSM303 into your Arduino libraries folder (step 1).
5. Install Pololu MinIMU9AHRS library: Download zip file. Unzip it in a temporary folder and then copy the folder MinIMU9AHRS into your Arduino libraries folder (step 1).
6. Restart your Arduino IDE, closing all open windows.

7. Now check under Sketch ⟶ Include Library ⟶ (scroll down to the section Contributed Libraries). You should see L3G and LSM303. MinIMU9AHRS may not be listed there since the folder structure is different.

4.4.3 Make the Circuit

Now, insert the headers on which the sensor is mounted to the breadboard. Hold the sensor from the sides, align to the holes properly and press from the headers side: **Warning:** Do not use excessive force. Make the following connections.
Note: VDD on the sensor is not connected to anything.

Arduino	MinIMU-9
5V	VIN
GND	GND
SDA or A4	SDA
SCL or A5	SCL

4.4.4 Calibrate the Magnetometer

1. Open File ⟶ Examples ⟶ LSM303 ⟶ Calibrate, and upload it to the Arduino as usual.

2. Open the Serial Monitor and set it at 9600 baud.

3. You will see a scrolling output. Now rotate the sensor (by rotating the breadboard) in all possible orientations, till the Serial Monitor output does not change anymore. This could look, e.g. like

```
min: { -3332,  -3122,  -2326}    max: { +2195,  +2277,  +3049}
```

4.4.5 Run the Example Program

If you can find MinIMU9AHRS within File ⟶ Examples (scoll down), open it; Otherwise, from the libraries sub-folder of your sketch-book folder, open the sketch `MinIMU9AHRS.ino` (look into its subfolders of MinIMU9AHRS).

Search for calibration constants `M_X_MIN` through `M_Z_MAX` and change these to the values you found in the calibration step. The IDE will ask you to save your modified file under a different name since the examples are read-only. Upload it now to the Arduino as usual. As an example, if you got the calibration values shown in the previous section, the values in `MinIMU9AHRS.ino` will change to:

```
// LSM303 magnetometer calibration constants; use the Calibrate example from
// the Pololu LSM303 library to find the right values for your board
#define M_X_MIN -3332
#define M_Y_MIN -3122
#define M_Z_MIN -2326
#define M_X_MAX 2195
#define M_Y_MAX 2277
#define M_Z_MAX 3049
```

Warning: Do not move the breadboard for the first few seconds. When the AHRS program first starts running, it takes some readings to establish a baseline orientation, during which it expects both the roll and the pitch of the sensors to be zero. Therefore, it is important to keep

the MinIMU-9 level for a few seconds after powering on or resetting the Arduino or connecting to it from a computer.

Open the Serial Monitor and change its baud rate to **115200**. You should see output similar to the following:

```
 Pololu MinIMU-9 + Arduino AHRS
21
-62
34
-29
-21
18
!ANG:-0.00,0.00,0.00
!ANG:0.00,0.02,0.00
!ANG:0.03,0.04,0.00
!ANG:0.03,0.04,0.00
!ANG:-0.00,0.03,0.01
!ANG:0.02,0.03,0.00
!ANG:0.05,0.03,0.32
!ANG:0.07,0.03,0.64
!ANG:0.08,0.03,0.97
!ANG:0.07,0.02,1.32
```

By default, the Arduino code treats the positive X axis of the MinIMU-9 as forward and the *negative* Z axis as up. What you are seeing are the estimated roll, pitch, and yaw angles (in degrees) which are based on the data fused from the accelerometer, gyro, and the magnetometer.

Task 4.8 *Rotate the sensor to individually change the roll, pitch, and the yaw angles. Can you see the strange yaw and roll values near the gimbal-lock (pitch $\pm 90^\circ$)?*

4.4.6 Visualize Data in Processing

Task 4.9 *Now fill-in the missing parts (maked with "// Fill-in") in the Processing code Listing 4.2 to visualize the IMU data. Make use of the following hints:*

- *Use baud of 115200 when you create the serial interface.*
- *Now you also have yaw. The values you get from the sensor are in degrees which need to be converted to radians, e.g. as* `yaw = yaw*Math.PI/180.;`
- *To account for the fact that Processing coordinate system is left-handed, we have negated the yaw.*
- *Read and understand the function* `parseStringForRollPitchYaw`*. In this function, we parse the message coming from the sensor. You may wish to consult the documentation of some useful string functions: substring (variation 1, variation 2), split (you just need to split at ","). The split function will give an array of substrings: The length of array substring array* `A` *is simply* `A.length`*.*
- *Explain the working of this function to your supervisor.*
- *For the visualization to work, orient the sensor such that the X axis is pointing to the right, and the Y axis is pointing towards you. Now the processing view is looking straight down at the sensor.*

```
import processing.serial.*;

int lf = 10;    // Linefeed in ASCII
String receivedString = null;
Serial myPort;  // Serial port you are using

float roll= 0., pitch= 0., yaw= 0.; // in radians

void setup() {
  size(640, 360, P3D);
  noStroke();
  colorMode(RGB, 1);

  myPort = new Serial(this, "COM3", 115200);  // 115200 for the IMU
  // Replace "/dev/ttyUSB0" by the Arduino port: "COM1", "COM15", Serial.list()
    [0], etc
  myPort.clear();
}

void draw_axes() {
  stroke (1, 0, 0) ;
  line (0, 0, 0, 800, 0, 0) ; // Red X - Axis
  stroke (0, 1, 0) ;
  line (0, 0, 0, 0, 800, 0) ; // Green Y - Axis
  stroke (0, 0, 1) ;
  line (0, 0, 0, 0, 0, 800) ; // Blue Z - Axis
  noStroke();
}

void draw_rgb_cube() {
  background (0.4, 0.4, 0.4) ;
  scale(100);
  beginShape(QUADS);

  // Visible Top (As +Z is coming out of the screen)
  // Face normal to +Z of Processing: Blue
  fill(0, 0, 1);
  vertex(-1, 1, 1);
  fill(0, 0, 1);
  vertex( 1, 1, 1);
  fill(0, 0, 1);
  vertex( 1, -1, 1);
  fill(0, 0, 1);
  vertex(-1, -1, 1);

  // Face normal to (-Z) of Processing: Light Blue
  fill(0.5, 0.5, 1);
  vertex( 1, 1, -1);
  fill(0.5, 0.5, 1);
  vertex(-1, 1, -1);
  fill(0.5, 0.5, 1);
  vertex(-1, -1, -1);
  fill(0.5, 0.5, 1);
  vertex( 1, -1, -1);

  // Face normal to +X of Processing: Red
  fill(1, 0, 0);
  vertex( 1, 1, 1);
  fill(1, 0, 0);
  vertex( 1, 1, -1);
  fill(1, 0, 0);
  vertex( 1, -1, -1);
  fill(1, 0, 0);
```

```
  vertex( 1, -1, 1);


  // Face normal to (-X) of Processing: Pink
  fill(1, 0.5, 0.5);
  vertex(-1, 1, -1);
  fill(1, 0.5, 0.5);
  vertex(-1, 1, 1);
  fill(1, 0.5, 0.5);
  vertex(-1, -1, 1);
  fill(1, 0.5, 0.5);
  vertex(-1, -1, -1);

  // Face normal to +Y of Processing: Green
  fill(0, 1, 0);
  vertex(-1, 1, -1);
  fill(0, 1, 0);
  vertex( 1, 1, -1);
  fill(0, 1, 0);
  vertex( 1, 1, 1);
  fill(0, 1, 0);
  vertex(-1, 1, 1);

  // Face normal to (-Y) of Processing: Light Green
  fill(0.5, 1, 0.5);
  vertex(-1, -1, -1);
  fill(0.5, 1, 0.5);
  vertex( 1, -1, -1);
  fill(0.5, 1, 0.5);
  vertex( 1, -1, 1);
  fill(0.5, 1, 0.5);
  vertex(-1, -1, 1);

  endShape();
}

void draw() {
  while (myPort.available() > 0) {
    receivedString = myPort.readStringUntil(lf);
    if (receivedString != null) {
      boolean newSample= parseStringForRollPitchYaw(receivedString);
      if (newSample) {

        // **Hack: Processing XYZ basis is unfortunately LEFT-handed:
        // But the z-axis of the IMU is pointing down, so we do not need to do:
    yaw= -yaw;

        // Draw the rotated object here.
        translate ( width /2.0, height /2.0, -50) ;

        // With body-fixed rotations
        // Fillin: Use rotateX, etc.

        // printMatrix();

        draw_rgb_cube();
      }
    }
  }
  myPort.clear();
}
```

```
boolean parseStringForRollPitchYaw(String str) {
  boolean sampleComplete= false;
  boolean uninterpreted= true;
  uninterpreted= (str.length() < 10); // Actually, the threshold 10 can be
    increased.
  if (!uninterpreted) {
    uninterpreted= !(str.substring(0, 5 /* exclusive */).equals("!ANG:"));
  }

  uninterpreted= (str.split(",").length != 3);

  if (!uninterpreted) {
    String[] substrings= str.substring(5).split(","); // ignore the initial !ANG
    :
    if (substrings.length == 3) {
      roll= float(substrings[0]); // in degrees
      roll *= Math.PI/180.; // convert to radians

      pitch= float(substrings[1]);
      pitch *= Math.PI/180.;

      yaw= float(substrings[2]);
      yaw *= Math.PI/180.;

      sampleComplete= true;
    }
  } else {
    print("Uninterpreted message: ");
    println(receivedString);  // Prints status messages
  }

  return sampleComplete;
}
```

Listing 4.2: Fill in the missing parts of this Processing code.

Chapter 5

Lab 5

In this lab, we are going to do an integrated experiment in which:

- We use multiple sensors/actuators together;
- We illustrate the use of maintaining the system-state within the software. Such "state-machines" are indispensable for creating complex intelligent systems.

5.1 Tic Tac Toe

Most people are familiar with Tic Tac Toe depicted in Fig. 5.1(a). We will now do an Arduino project to play Tic Tac Toe using IR remotes where LEDs replace the noughts and crosses: This is shown in Fig. 5.1(b).

Due to the limited number of pins available on the Arduino, we want to minimize the number LEDs needed. Instead of having two differently colored LEDs in all 3×3 slots of the game, we just use one LED per slot. To distinguish between the noughts and the crosses, we replace them with the following players:

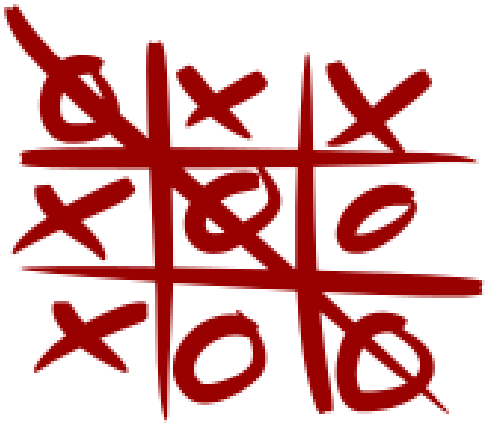
- Player Blinky: If a slot is "owned" by this player, its LED blinks.
- Player Steady: If a slot is "owned" by this player, its LED stays lit steadily.

In addition, if a slot is not owned by either, its LED stays off.

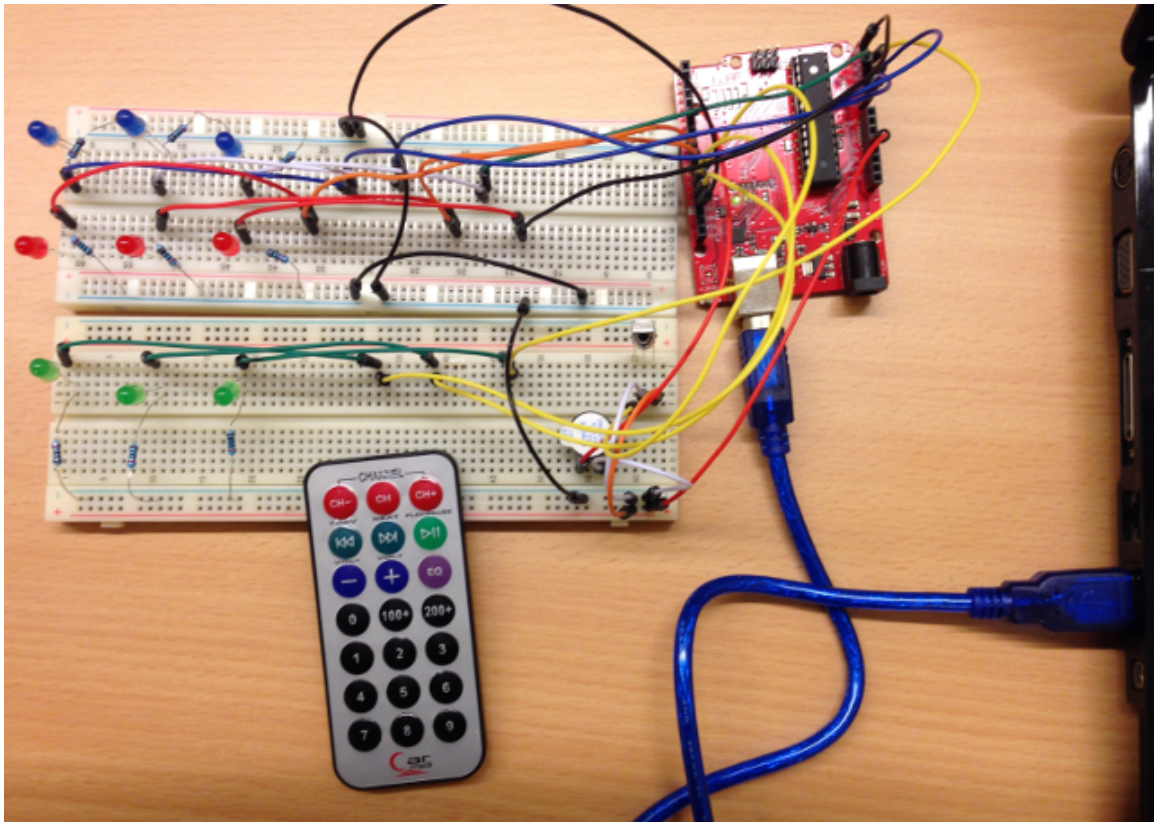
5.1.1 The Rules

The game proceeds as follows:

- Player Blinky starts by clicking the button on the IR remote corresponding to the slot she/he wants to occupy. Conveniently enough, the IR remote has 9 buttons numbered 1–9, already arranged in a 3×3 grid. The LED corresponding to the button starts blinking.
- The program now automatically expects player Steady to play. Player Steady now presses a button to occupy a slot – if that slot is already occupied, the piezo-buzzer beeps shortly as a warning, and the player is given another chance to make a choice. If the choice is valid, the corresponding LED lights up.
- The game continues like this, the players taking turns until either a player wins or the game is a draw. In either case, the win or the draw is automatically recognized by the Arduino program and the buzzer plays a tune to note this. A descriptive message is printed on the serial monitor. After this, the game is reinitialized and starts again.



(a) The 3×3 grid of the game. "Tic tac toe" by Symode09 - Own work. Licensed under Public Domain via Commons.



(b) The Experiment Setup. The colors of the LEDs have no meaning: You can also choose all LEDs of the same color.

Figure 5.1: The Tic-Tac-Toe Experiment.

5.2 The Tasks

Task 5.1 *Re-run the code in the IR remote experiment in Listing 3.6.2 and write down the hex-codes of buttons labeled 1–9.*

Hint: Note that to express an int/long in hexadecimal format, you need to prefix it by 0x, e.g. `long code= 0xFF30CF;`

Task 5.2 *Now design the circuit for the Tic-Tac-Toe experiment* ***using Fritzing***. *You are expected to come up with the design on your own, but we provide Fig. 5.1(b) as guidance. Try to minimize the number of Arduino pins and jumpers needed.*

Task 5.3 *Now using your design, assemble the circuit.*

Hints:

- You can use Arduino analog-input pins also as normal digital I/O pins.
- You cannot use Arduino pins 0 and 1 for digital I/O if you are using the serial monitor or any other serial communication.
- Arduino pins 2 and 3 are used for interrupts on the UNO, so do not use them unless you are sure that any libraries being used by you (such as IRremote) do not make use of them internally.

Task 5.4 *Now write the code to play the game as described in Sec. 5.1.1.*

Hint: It is best to store system states and pin-assignments in C/C++ 2D arrays, in this case, of size 3×3. You can revise them at this link.

Task 5.5 ***Bonus****: In case you have time left over after finishing the above tasks and all the previous labs, you can try to add an LCD to display the game-status, the name of the player whose turn it is next, and the score. With the addition of the LCD, one does not need the serial monitor anymore, and the game can be played even by powering the Arduino by an external supply such as a 9V battery.*