BUILD THE CIRCUIT

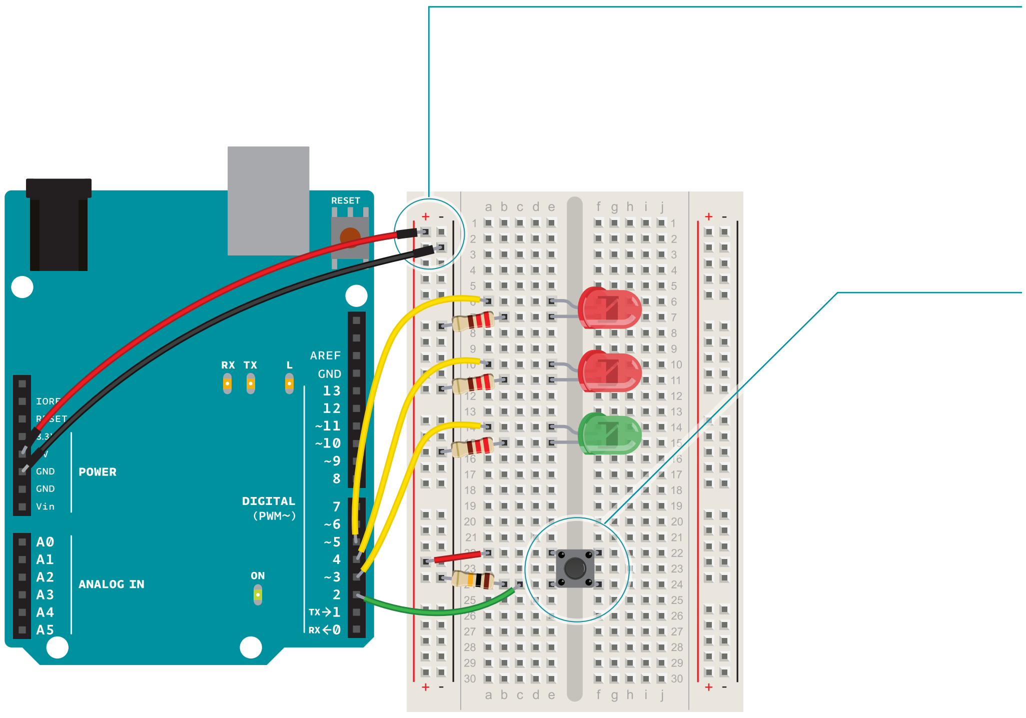


Fig. 1

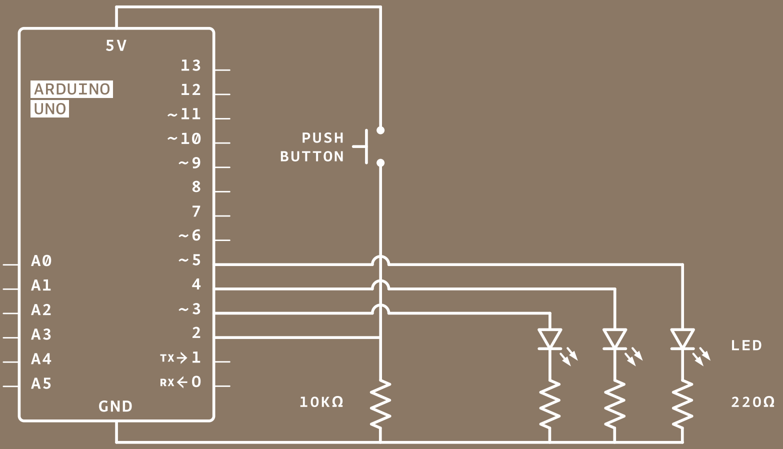
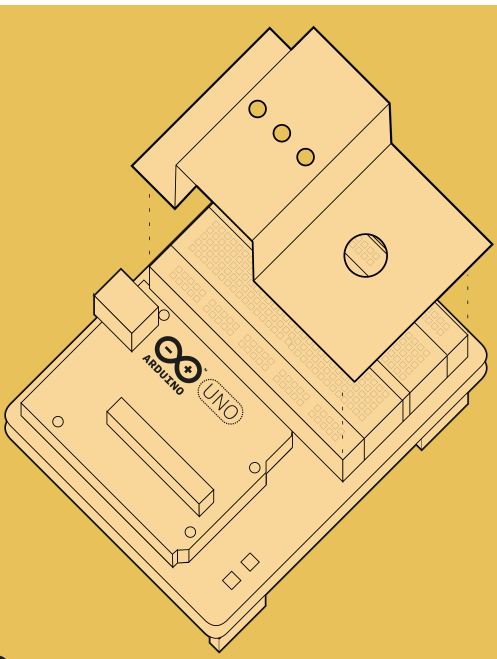


Fig. 2

1 Wire up your breadboard to the Arduino's 5V and ground connections, just like the previous project. Place the two red LEDs and one green LED on the breadboard. Attach the cathode (short leg) of each LED to ground through a 220-ohm resistor. Connect the anode (long leg) of the green LED to pin 3. Connect the red LEDs' anodes to pins 4 and 5, respectively.

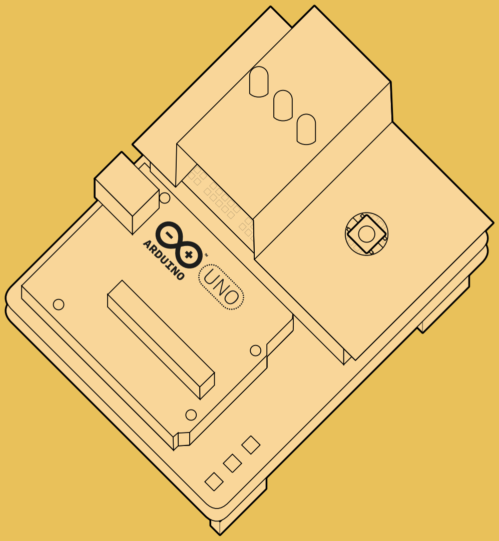
2 Place the switch on the breadboard just as you did in the previous project. Attach one side to power, and the other side to digital pin 2 on the Arduino. You'll also need to add a 10k-ohm resistor from ground to the switch pin that connects to the Arduino. That pull-down resistor connects the pin to ground when the switch is open, so it reads `LOW` when there is no voltage coming in through the switch.

You can cover the breadboard the template provided in the kit. Or you can decorate it to make your own launch system. The lights turning on and off mean nothing by themselves, but when you put them in a control panel and give them labels, they gain meaning. What do you want the green LED to mean? What do the flashing red LEDs mean? You decide!



1

Fold the pre-cut paper as shown.



2

Place the folded paper over the breadboard. The three LEDs and pushbutton will help keep it in place.

THE CODE

Some notes before you start

Every Arduino program has two main functions. Functions are parts of a computer program that run specific commands. Functions have unique names, and are "called" when needed. The necessary functions in an Arduino program are called `setup()` and `loop()`. These functions need to be declared, which means that you need to tell the Arduino what these functions will do. `setup()` and `loop()` are declared as you see on the right.
In this program, you're going to create a variable before you get into the main part of the program. Variables are names you give to places in the Arduino's memory so you can keep track of what is happening. These values can change depending on your program's instructions.
Variable names should be descriptive of whatever value they are storing. For example, a variable named `switchState` tells you what it stores: the state of a switch. On the other hand, a variable named "`x`" doesn't tell you much about what it stores.

Let's start coding

To create a variable, you need to declare what *type* it is. The *data type* `int` will hold a whole number (also called an *integer*); that's any number without a decimal point. When you declare a variable, you usually give it an initial value as well. The declaration of the variable as every statement must end with a semicolon (;).

Configure pin functionality

The `setup()` runs once, when the Arduino is first powered on. This is where you configure the digital pins to be either inputs or outputs using a function named `pinMode()`. The pins connected to LEDs will be OUTPUTs and the switch pin will be an INPUT.

Create the loop function

The `loop()` runs continuously after the `setup()` has completed. The `loop()` is where you'll check for voltage on the inputs, and turn outputs on and off. To check the voltage level on a digital input, you use the function `digitalRead()` that checks the chosen pin for voltage. To know what pin to check, `digitalRead()` expects an *argument*.
Arguments are information that you pass to functions, telling them how they should do their job. For example, `digitalRead()` needs one argument: what pin to check. In your program, `digitalRead()` is going to check the state of

```
void setup(){
}

void loop(){
}
```

{ Curly brackets }
Any code you write inside the curly brackets will be executed when the function is called.

```
int switchState = 0;
```

```
void setup(){
  pinMode(3,OUTPUT);
  pinMode(4,OUTPUT);
  pinMode(5,OUTPUT);
  pinMode(2,INPUT);
}
```

Case sensitivity
Pay attention to the `case sensitivity` in your code.
For example, `pinMode` is the name of a command, but pinmode will produce an error.

```
void loop(){
  switchState = digitalRead(2);
  // this is a comment
```

Comments
If you ever want to include natural language in your program, you can leave a comment.
Comments are notes you leave for yourself that the computer ignores. To write a comment, add two slashes `//`
The computer will ignore anything on the line after those slashes.

pin 2 and store the value in the switchState variable.
If there's voltage on the pin when `digitalRead()` is called, the `switchState` variable will get the value `HIGH` (or 1). If there is no voltage on the pin, switchState will get the value `LOW` (or 0).

The if statement

Above, you used the word if to check the state of something (namely, the switch position). An `if()` statement in programming compares two things, and determines whether the comparison is true or false. Then it performs actions you tell it to do. When comparing two things in programming, you use two equal signs ==. If you use only one sign, you will be setting a value instead of comparing it.

Build up your spaceship

`digitalWrite()` is the command that allows you to send 5V or 0V to an output pin. `digitalWrite()` takes two arguments: what pin to control, and what value to set that pin, `HIGH` or `LOW`. If you want to turn the red LEDs on and the green LED off inside your `if()` statement, your code would look like this .

If you run your program now, the lights will change when you press the switch. That's pretty neat, but you can add a little more complexity to the program for a more interesting output.

You've told the Arduino what to do when the switch is open. Now define what happens when the switch is closed. The `if()` statement has an optional `else` component that allows for something to happen if the original condition is not met. In this `case`, since you checked to see if the switch was `LOW`, write code for the `HIGH` condition after the `else` statement.

To get the red LEDs to blink when the button is pressed, you'll need to turn the lights off and on in the `else` statement you just wrote. To do this, change the code to look like this.

Now your program will flash the red LEDs when the switch button is pressed.

After setting the LEDs to a certain state, you'll want the Arduino to pause for a moment before changing them back. If you don't wait, the lights will go back and forth so fast that it will appear as if they are just a little dim, not on and off. This is because the Arduino goes through its `loop()` thousands of times each second, and the LED will be turned on and off quicker than we can perceive. The `delay()` function lets you stop the Arduino from executing anything for a period of time. `delay()` takes an argument that determines the number of milliseconds before it executes the next set of code. There are 1000 milliseconds in one second. `delay(250)` will pause for a quarter second.

```
  if (switchState == LOW) {
  // the button is not pressed

    digitalWrite(3, HIGH); // green LED
    digitalWrite(4, LOW);  // red LED
    digitalWrite(5, LOW);  // red LED
  }
```

It can be helpful to write out the flow of your program in pseudocode: a way of describing what you want the program to do in plain language, but structured in a way that makes it easy to write a real program from it. In this case you're going to determine if `switchState` is `HIGH` (meaning the button is pressed) or not. If the switch is pressed, you'll turn the green LED off and the red ones on. In pseudocode, the statement could look like this:

```
if the switchState is LOW:
  turn the green LED on
  turn the red LEDs off

if the switchState is HIGH:
  turn the green LED off
  turn the red LEDs on
```

```
  else {  // the button is pressed
    digitalWrite(3, LOW);
    digitalWrite(4, LOW);
    digitalWrite(5, HIGH);

    delay(250);  // wait for a quarter second
    // toggle the LEDs
    digitalWrite(4, HIGH);
    digitalWrite(5, LOW);
    delay(250); // wait for a quarter second

  }
} // go back to the beginning of the loop
```

USE IT

Once your Arduino is programmed, you should see the green light turn on. When you press the switch, the red lights will start flashing, and the green light will turn off. Try changing the time of the two `delay()` functions; notice what happens to the lights and how the response of the system changes depending on the speed of the flashing. When you call a `delay()` in your program, it stops all other functionality. No sensor readings will happen until that time period has passed. While delays are often useful, when designing your own projects make sure they are not unnecessarily interfering with your interface.

How would you get the red LEDs to be blinking when your program starts? How could you make a larger, or more complex interface for your interstellar adventures with LEDs and switches?

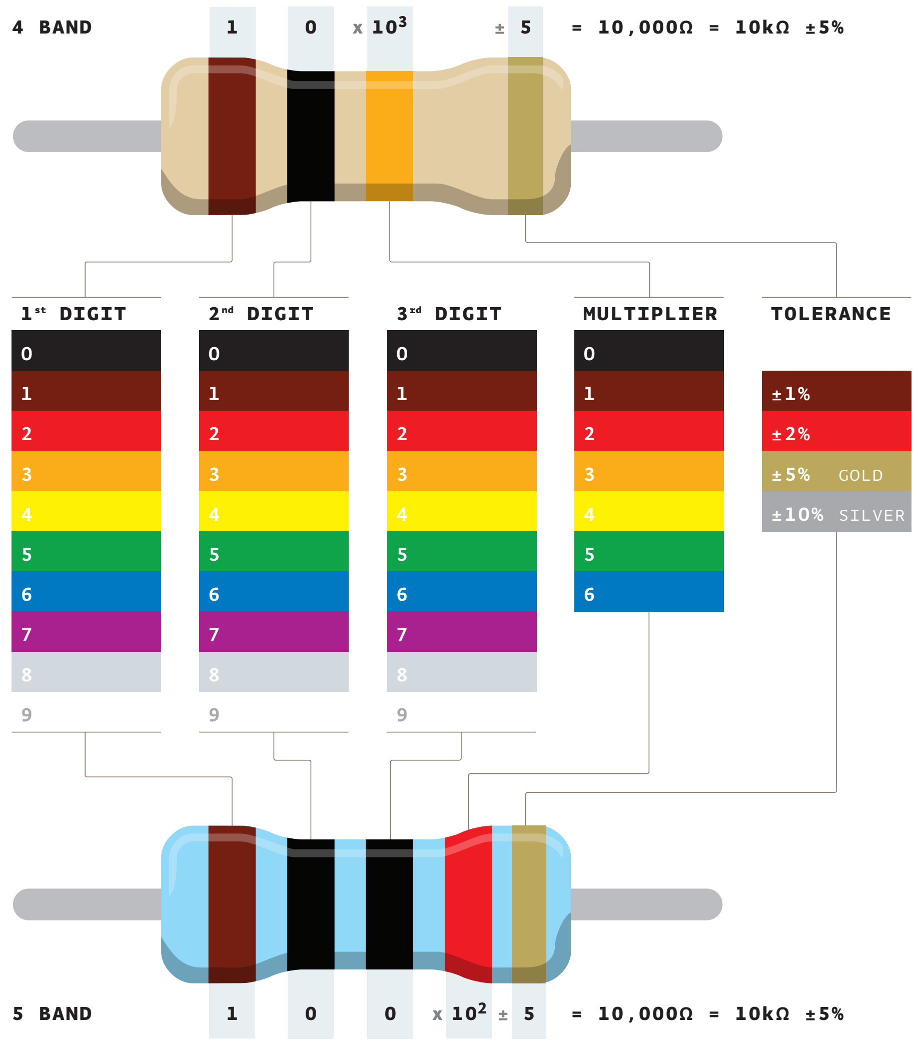
When you start creating an interface for your project, think about what people's expectations are while using it. When they press a button, will they want immediate feedback? Should there be a delay between their action and what the Arduino does? Try and place yourself in the shoes of a different user while you design, and see if your expectations match up to the reality of your project.

In this project, you created your first Arduino program to control the behavior of some LEDs based on a switch. You've used variables, an if()...else statement, and functions to read the state of an input and control outputs.

HOW TO READ RESISTOR COLOR CODES

Resistor values are marked using colored bands, according to a code developed in the 1920s, when it was too difficult to write numbers on such tiny objects.

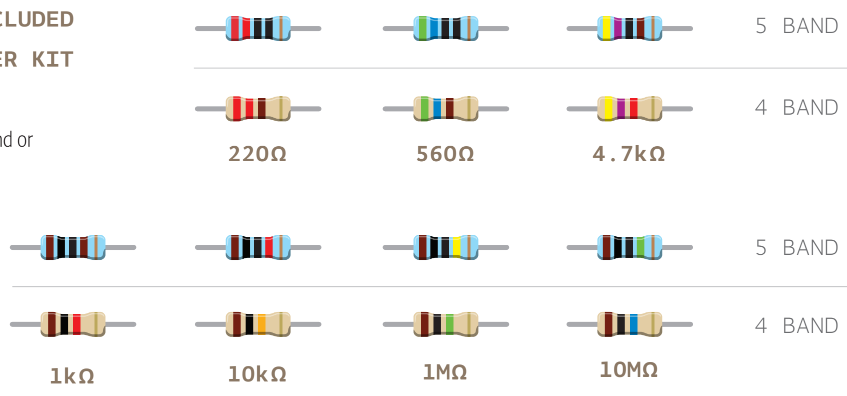
Each color corresponds to a number, like you see in the table below. Each resistor has either 4 or 5 bands. In the 4-band type, the first two bands indicate the first two digits of the value while the third one indicates the number of zeroes that follow (technically it reprents the power of ten). The last band specifies the tolerance: in the example below, gold indicates that the resistor value can be 10k ohm plus or minus 5%.



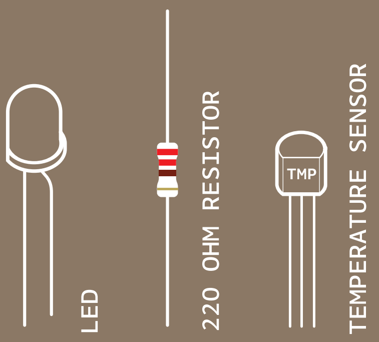
1st DIGIT	2nd DIGIT	3rd DIGIT	MULTIPLIER	TOLERANCE
⬛ 0	⬛ 0	⬛ 0	⬛ 0	
🟫 1	🟫 1	🟫 1	🟫 1	🟫 ±1%
🟥 2	🟥 2	🟥 2	🟥 2	🟥 ±2%
🟧 3	🟧 3	🟧 3	🟧 3	±5% GOLD
🟨 4	🟨 4	🟨 4	🟨 4	±10% SILVER
🟩 5	🟩 5	🟩 5	🟩 5	
🟦 6	🟦 6	🟦 6	🟦 6	
🟪 7	🟪 7	🟪 7		
▨ 8	▨ 8	▨ 8		
⬜ 9	⬜ 9	⬜ 9		

RESISTORS INCLUDED IN THE STARTER KIT

You'll find either a 4 band or a 5 band version.



Ø3



INGREDIENTS

LOVE-O-METER

TURN THE ARDUINO INTO A LOVE MACHINE. USING AN ANALOG INPUT, YOU'RE GOING TO REGISTER JUST HOW HOT YOU REALLY ARE!

Discover: analog Input, using the serial monitor

Time: **45 MINUTES**

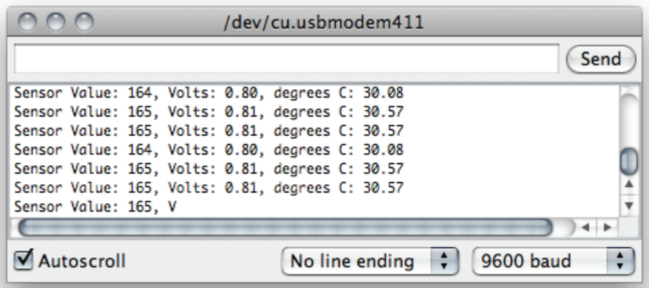
Level: ■□□□□

Builds on projects: **1, 2**

While switches and buttons are great, there's a lot more to the physical world than on and off. Even though the Arduino is a digital tool, it's possible for it to get information from analog sensors to measure things like temperature or light. To do this, you'll take advantage of the Arduino's built-in Analog-to-Digital Converter (ADC). Analog in pins A0-A5 can report back a value between 0-1023, which maps to a range from 0 volts to 5 volts.

You'll be using a ***temperature sensor*** to measure how warm your skin is. This component outputs a changing voltage depending on the temperature it senses. It has three pins: one that connects to ground, another that connects to power, and a third that outputs a variable voltage to your Arduino. In the sketch for this project, you'll read the sensor's output and use it to turn LEDs on and off, indicating how warm you are. There are several different models of temperature sensor. This model, the TMP36, is convenient because it outputs a voltage that changes directly proportional to the temperature in degrees Celsius.

The Arduino IDE comes with a tool called the ***serial monitor*** that enables you to report back results from the microcontroller. Using the serial monitor, you can get information about the status of sensors, and get an idea about what is happening in your circuit and code as it runs.



```
Sensor Value: 164, Volts: 0.80, degrees C: 30.08
Sensor Value: 165, Volts: 0.81, degrees C: 30.57
Sensor Value: 165, Volts: 0.81, degrees C: 30.57
Sensor Value: 164, Volts: 0.80, degrees C: 30.08
Sensor Value: 165, Volts: 0.81, degrees C: 30.57
Sensor Value: 165, Volts: 0.81, degrees C: 30.57
Sensor Value: 165, V
```

Serial monitor

Fig. 1

BUILD THE CIRCUIT

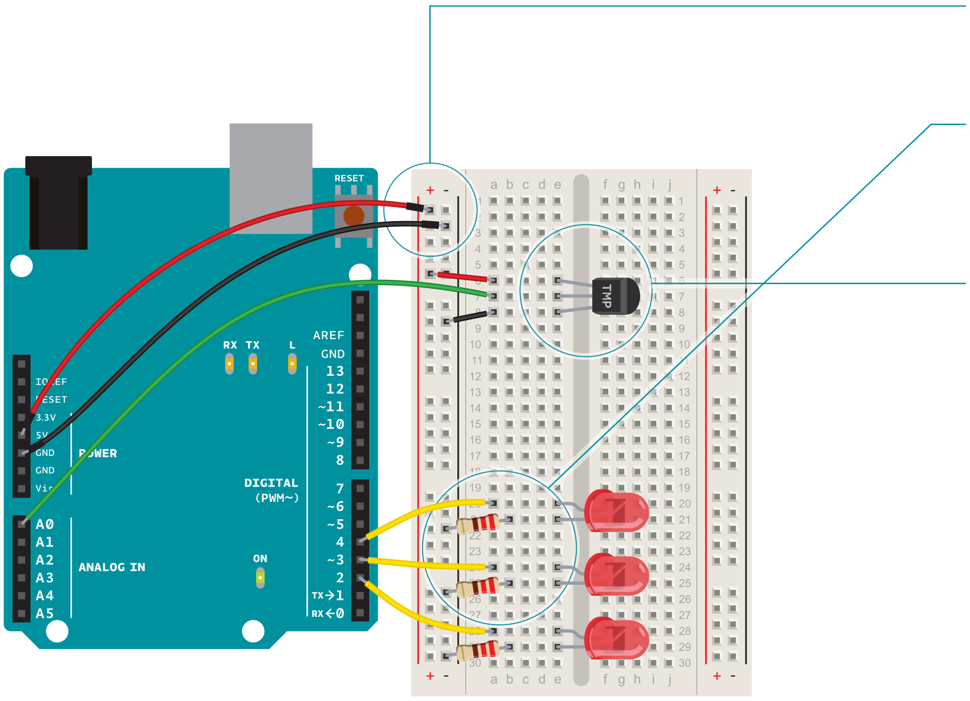


Fig. 2

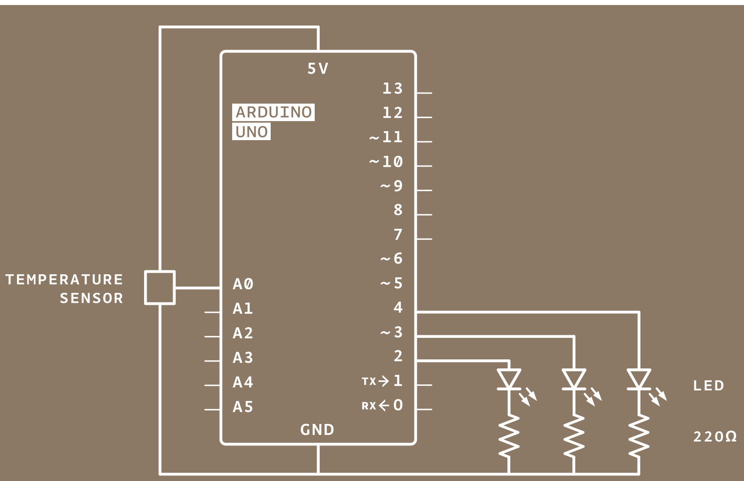


Fig. 3

In this project, you need to check the ambient temperature of the room before proceeding. You're checking things manually right now, but this can also be accomplished through calibration. It's possible to use a button to set the baseline temperature, or to have the Arduino take a sample before starting the `loop()` and use that as the reference point. Project 6 gets into details about this, or you can look at the Calibration example that comes bundled with the Arduino software:
arduino.cc/calibration

1 Just as you've been doing in the earlier projects, wire up your breadboard so you have power and ground.

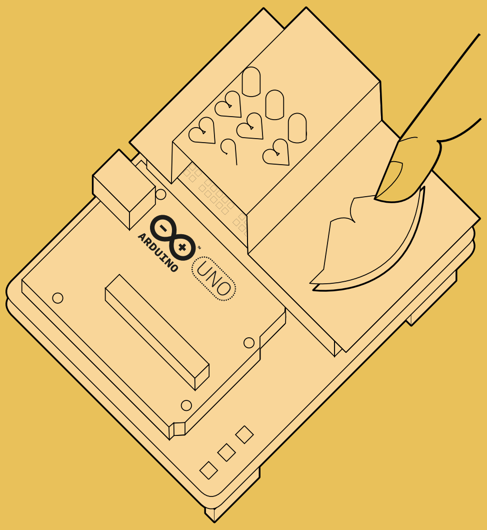
2 Attach the cathode (short leg) of each of the LEDs you're using to ground through a 220-ohm resistor. Connect the anodes of the LEDs to pins 2 through 4. These will be the indicators for the project.

3 Place the TMP36 on the breadboard with the rounded part facing away from the Arduino (the order of the pins is important!) as shown in Fig. 2. Connect the left pin of the flat facing side to power, and the right pin to ground. Connect the center pin to pin A0 on your Arduino. This is analog input pin 0.

Create an interface for your sensor for people interact with. A paper cutout in the shape of a hand is a good indicator. If you're feeling lucky, create a set of lips for someone to kiss, see how well that lights things up! You might also want to label the LEDs to give them some meaning. Maybe one LED means you're a cold fish, two LEDs means you're warm and friendly, and three LEDs means you're too hot to handle!

1

Cut out a piece of paper that will fit over the breadboard. Draw a set of lips where the sensor will be, and cut some circles for the LEDs to pass through.



2

Place the cutout over the breadboard so that the lips cover the sensor and the LEDs fit into the holes. Press the lips to see how hot you are!

THE CODE

A pair of useful constants

Constants are similar to variables in that they allow you to uniquely name things in the program, but unlike variables they cannot change. Name the analog input for easy reference, and create another named constant to hold the baseline temperature. For every 2 degrees above this baseline, an LED will turn on.
You've already seen the int datatype, used here to identify which pin the sensor is on. The temperature is being stored as a *float*, or floating-point number. This type of number has a decimal point, and is used for numbers that can be expressed as fractions.

Initialize the serial port to the desired speed

In the setup you're going to use a new command, `Serial.begin()`. This opens up a connection between the Arduino and the computer, so you can see the values from the analog input on your computer screen.
The argument `9600` is the speed at which the Arduino will communicate, 9600 bits per second. You will use the Arduino IDE's serial monitor to view the information you choose to send from your microcontroller. When you open the IDE's serial monitor verify that the baud rate is 9600.

Initialize the digital pin directions and turn off

Next up is a `for()` loop to set some pins as outputs. These are the pins that you attached LEDs to earlier. Instead of giving them unique names and typing out the `pinMode()` function for each one, you can use a `for()` loop to go through them all quickly. This is a handy trick if you have a large number of similar things you wish to iterate through in a program. Tell the `for()` loop to run through pins 2 to 4 sequentially.

Read the temperature sensor

In the `loop()`, you'll use a local variable named `sensorVal` to store the reading from your sensor. To get the value from the sensor, you call `analogRead()` that takes one argument: what pin it should take a voltage reading on. The value, which is between 0 and 1023, is a representation of the voltage on the pin.

Send the temperature sensor values to the computer

The function `Serial.print()` sends information from the Arduino to a connected computer. You can see this information in your serial monitor. If you give `Serial.print()` an argument in quotation marks, it will print out the text you typed. If you give it a variable as an argument, it will print out the value of that variable.

```
const int sensorPin = A0;
const float baselineTemp = 20.0;
```

```
void setup(){
  Serial.begin(9600); // open a serial port
```

```
  for(int pinNumber = 2; pinNumber<5; pinNumber++){
    pinMode(pinNumber,OUTPUT);
    digitalWrite(pinNumber, LOW);
  }
}
```

`for() loop tutorial`
arduino.cc/for

```
void loop(){
  int sensorVal = analogRead(sensorPin);
```

```
  Serial.print(“Sensor Value: ”);
  Serial.print(sensorVal);
```

Convert sensor reading to voltage

With a little math, it's possible to figure out what the real voltage on the pin is. The voltage will be a value between 0 and 5 volts, and it will have a fractional part (for example, it might be 2.5 volts), so you'll need to store it inside a `float`. Create a variable named voltage to hold this number. Divide `sensorVal` by 1024.0 and multiply by 5.0. The new number represents the voltage on the pin.

Just like with the sensor value, you'll print this out to the serial monitor.

Convert the voltage to temperature and send the value to the computer

If you examine the sensor's *datasheet*, there is information about the range of the output voltage. Datasheets are like manuals for electronic components. They are written by engineers, for other engineers. The datasheet for this sensor explains that every 10 millivolts of change from the sensor is equivalent to a temperature change of 1 degree Celsius. It also indicates that the sensor can read temperatures below 0 degrees. Because of this, you'll need to create an offset for values below freezing (0 degrees). If you take the voltage, subtract 0.5, and multiply by 100, you get the accurate temperature in degrees Celsius. Store this new number in a floating point variable called temperature.

Now that you have the real temperature, print that out to the serial monitor too. Since the temperature variable is the last thing you're going to be printing out in this loop, you're going to use a slightly different command: `Serial.println()`. This command will create a new line in the serial monitor after it sends the value. This helps make things easier to read in when they are being printed out.

Turn off LEDs for a low temperature

With the real temperature, you can set up an `if()…else` statement to light the LEDs. Using the baseline temperature as a starting point, you'll turn on one LED on for every 2 degrees of temperature increase above that baseline. You're going to be looking for a range of values as you move through the temperature scale.

```
// convert the ADC reading to voltage
float voltage = (sensorVal/1024.0) * 5.0;
```

```
Serial.print(“, Volts: ”);
Serial.print(voltage);
```

```
Serial.print(“, degrees C: ”);
// convert the voltage to temperature in degrees
float temperature = (voltage - .5) * 100;
Serial.println(temperature);
```

Starter Kit datasheets
arduino.cc/kitdatasheets

```
if(temperature < baselineTemp){
  digitalWrite(2, LOW);
  digitalWrite(3, LOW);
  digitalWrite(4, LOW);
```

Turn on one LED for a low temperature

The && operator means "and", in a logical sense. You can check for multiple conditions: "if the temperature is 2 degrees greater than the baseline, and it is less than 4 degrees above the baseline."

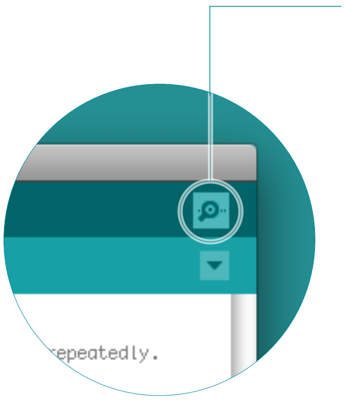
Turn on two LEDs for a medium temperature

If the temperature is between two and four degrees above the baseline, this block of code turns on the LED on pin 3 as well.

Turn on three LEDs for a high temperature

The Analog-to-Digital Converter can only read so fast, so you should put a small delay at the very end of your `loop()`. If you read from it too frequently, your values will appear erratic.

USE IT



With the code uploaded to the Arduino, click the serial monitor icon. You should see a stream of values coming out, formatted like this : `Sensor: 200, Volts: .70, degrees C: 17`

Try putting your fingers around the sensor while it is plugged into the breadboard and see what happens to the values in the serial monitor. Make a note of what the temperature is when the sensor is left in the open air.

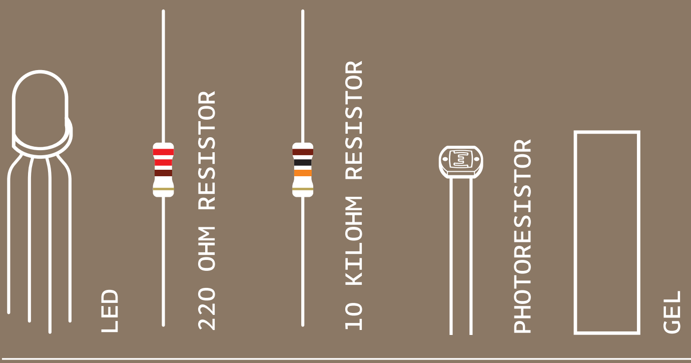
Close the serial monitor and change the baselineTemp constant in your program to the value you observed the temperature to be. Upload your code again, and try holding the sensor in your fingers. As the temperature rises, you should see the LEDs turn on one by one. Congratulations, hot stuff!

```
  }else if(temperature >= baselineTemp+2 &&
    temperature < baselineTemp+4){
    digitalWrite(2, HIGH);
    digitalWrite(3, LOW);
    digitalWrite(4, LOW);
  }else if(temperature >= baselineTemp+4 &&
    temperature < baselineTemp+6){
    digitalWrite(2, HIGH);
    digitalWrite(3, HIGH);
    digitalWrite(4, LOW);
  }else if(temperature >= baselineTemp+6){
    digitalWrite(2, HIGH);
    digitalWrite(3, HIGH);
    digitalWrite(4, HIGH);
  }
  delay(1);
}
```

Create an interface for two people to test their compatibility with each other. You get to decide what compatibility means, and how you'll sense it. Perhaps they have to hold hands and generate heat? Maybe they have to hug? What do you think?

Expanding the types of inputs you can read, you've used analogRead() and the serial monitor to track changes inside your Arduino. Now it's possible to read a large number of analog sensors and inputs.

04



INGREDIENTS

COLOR MIXING LAMP

USING A TRI-COLOR LED AND THREE PHOTORESISTORS, YOU'LL CREATE A LAMP THAT SMOOTHLY CHANGES COLORS DEPENDING ON EXTERNAL LIGHTING CONDITIONS

Discover: analog output, mapping values

Time: **45 MINUTES**
Level: ■■▫▫▫

Builds on projects: **1, 2, 3**

Blinking LEDs can be fun, but what about fading them, or mixing colors? You might expect that it's just a matter of providing less voltage to an LED to get it to fade.

The Arduino can't vary the output voltage on its pins, it can only output 5V. Hence you'll need to use a technique called ***Pulse Width Modulation (PWM)*** to fade LEDs. PWM rapidly turns the output pin high and low over a fixed period of time. The change happens faster than the human eye can see. It's similar to the way movies work, quickly flashing a number of still images to create the illusion of motion.

When you're rapidly turning the pin `HIGH` and `LOW`, it's as if you were changing the voltage. The percentage of time a pin is `HIGH` in a period is called *duty cycle*. When the pin is `HIGH` for half of the period and `LOW` for the other half, the duty cycle is 50%. A lower duty cycle gives you a dimmer LED than a higher duty cycle.

UNO
DIGITAL (PWM~)
7
~6
~5
4
~3
2
TX→1

The Arduino Uno has six pins set aside for PWM ***(digital pins 3, 5, 6, 9, 10, and 11)***, they can be identified by the ~ next to their number on the board.

For inputs in this project, you'll be using ***photoresistors*** (sensors that change their resistance depending on the amount of light that hits them, also known as photocells or light-dependent resistors). If you connect one end of the resistor to your Arduino, you can measure the change in resistance by checking the voltage on the pin.

BUILD THE CIRCUIT

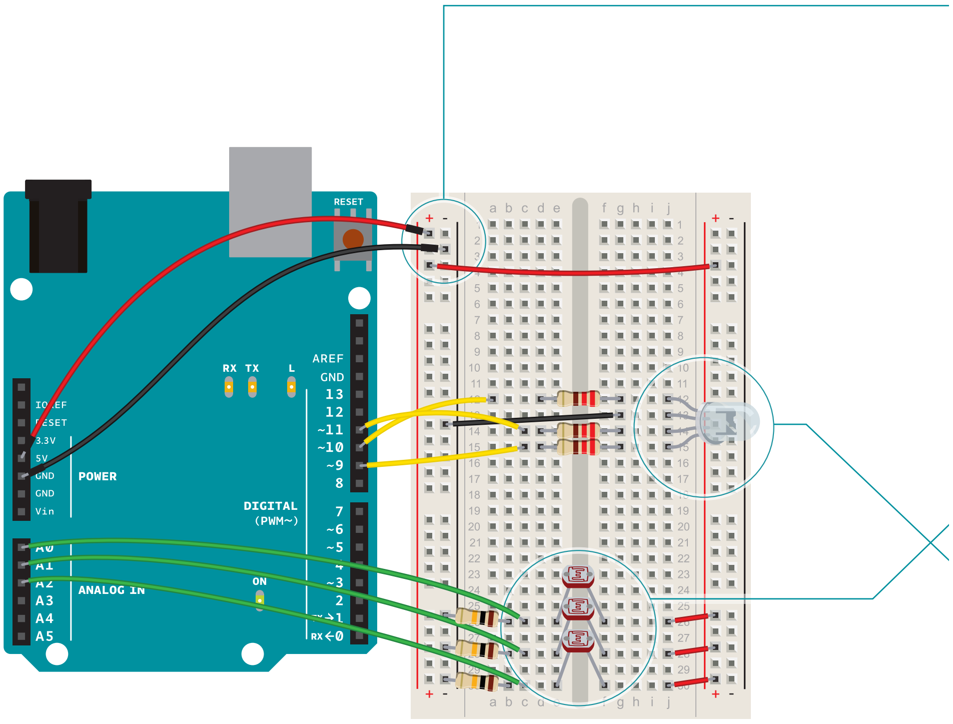


Fig. 1

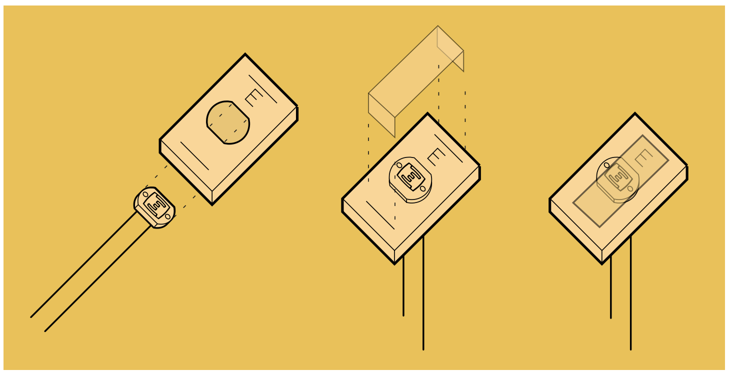


Fig. 2

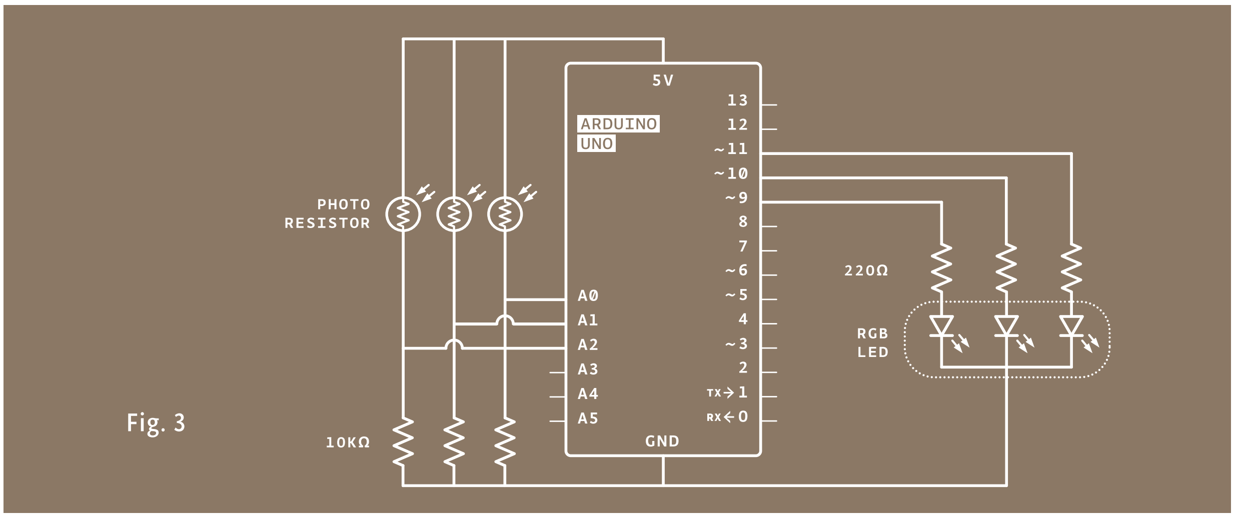
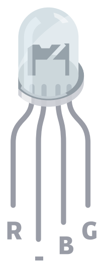


Fig. 3

1 Wire up your breadboard so you have power and ground on both sides, just like the earlier projects.

2 Place the three photoresistors on the breadboard so they cross the center divide from one side to the other, as shown in Fig. 1. Attach one end of each photoresistor to power. On the other side, attach a 10-kilohm resistor to ground. This resistor is in series with the photoresistor, and together they form a voltage divider. The voltage at the point where they meet is proportional to the ratio of their resistances, according to Ohm's Law (see Project 1 for more on Ohm's Law). As the resistance of the photoresistor changes when light hits it, the voltage at this junction changes as well. On the same side as the resistor, connect the photoresistors to Analog In pins 0, 1, and 2 with hookup wire.

3 Take the three colored gels and place one over each of the photoresistors. Place the red gel over the photoresistor connected to A0, the green over the one connected to A1, and the blue over the one connected to A2. Each of these filters lets only light of a specific wavelength through to the sensor it's covering. The red filter passes only red light, the green filter passes only green light, and the blue filter passes only blue light. This allows you to detect the relative color levels in the light that hits your sensors.



4 The LED with 4 legs is a common cathode RGB LED. The LED has separate red, green, and blue elements inside, and one common ground (the cathode). By creating a voltage difference between the cathode and the voltage coming out of the Arduino's PWM pins (which are connected to the anodes through 220-ohm resistors), you'll cause the LED to fade between its three colors. Make note of what the longest pin is on the LED, place it in your breadboard, and connect that pin to ground. Connect the other three pins to digital pins 9, 10 and 11 in series with 220-ohm resistors. Be sure to connect each LED lead to the correct PWM pin, according to the figure on the left.

THE CODE

Useful constants

Set up constants for the pins you're using for input and output, so you can keep track of which sensor pairs with which color on the LED. Use const int for the datatype.

Variables to store the sensor readings as well as the light level of each LED

Add variables for the incoming sensor values and for the output values you'll be using to fade the LED. You can use the `int` datatype for all the variables.

Setting the direction of the digital pins and setting up the serial port

In the `setup()`, begin serial communication at 9600 bps. Just like in the previous example, you will use this to see the values of the sensors in the serial monitor. Additionally, you will be able to see the mapped values you'll use to fade the LED. Also, define the LED pins as outputs with `pinMode()`.

Reading the value of each light sensor

In the `loop()` read the sensor values on A0, A1, and A2 with `analogRead()` and store the value in the appropriate variables. Put a small `delay()` between each `analogRead()` as the ADC takes a millisecond to do its work.

Report the sensor readings to the computer

Print out the sensor values on one line.
The "`\t`" is the equivalent of pressing the "`tab`" key on the keyboard.

```
const int greenLEDPin = 9;
const int redLEDPin = 11;
const int blueLEDPin = 10;

const int redSensorPin = A0;
const int greenSensorPin = A1;
const int blueSensorPin = A2;
```

```
int redValue = 0;
int greenValue = 0;
int blueValue = 0;

int redSensorValue = 0;
int greenSensorValue = 0;
int blueSensorValue = 0;
```

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

  pinMode(greenLEDPin,OUTPUT);
  pinMode(redLEDPin,OUTPUT);
  pinMode(blueLEDPin,OUTPUT);
}
```

```
void loop() {
  redSensorValue = analogRead(redSensorPin);
  delay(5);
  greenSensorValue = analogRead(greenSensorPin);
  delay(5);
  blueSensorValue = analogRead(blueSensorPin);
```

```
  Serial.print("Raw Sensor Values \t Red: ");
  Serial.print(redSensorValue);
  Serial.print("\t Green: ");
  Serial.print(greenSensorValue);
  Serial.print("\t Blue: ");
  Serial.println(blueSensorValue);
```

Converting the sensor readings

The function to change the LED's brightness via PWM is called `analogWrite()`. It needs two arguments: the pin to write to, and a value between 0-255. This second number represents the duty cycle the Arduino will output on the specified pin. A value of 255 will set the pin `HIGH` all the time, making the attached LED as bright as it can be. A value of 127 will set the pin `HIGH` half the time of the period, making the LED dimmer. 0 would set the pin `LOW` all the time, turning the LED off. To convert the sensor reading from a value between 0-1023 to a value between 0-255 for `analogWrite()`, divide the sensor reading by 4.

Report the calculated LED light levels

Print out the new mapped values on their own line.

Set the LED light levels

USE IT

Once you have your Arduino programmed and wired up, open the serial monitor. The LED will probably be an off-white color, depending on the predominant color of the light in your room. Look at the values coming from the sensors in the serial monitor, if you're in an environment with stable lighting, the number should probably be fairly consistent.

Turn off the light in the room you're in and see what happens to the values of the sensors. With a flashlight, illuminate each of the sensors individually and notice how the values change in the serial monitor, and notice how the LED's color changes. When the photoresistors are covered with a gel, they only react to light of a certain wavelength. This will give you the opportunity to change each of the colors independently.

```
redValue = redSensorValue/4;
greenValue = greenSensorValue/4;
blueValue = blueSensorValue/4;
```

```
Serial.print("Mapped Sensor Values \t Red: ");
Serial.print(redValue);
Serial.print("\t Green: ");
Serial.print(greenValue);
Serial.print("\t Blue: ");
Serial.println(blueValue);
```

```
analogWrite(redLEDPin, redValue);
analogWrite(greenLEDPin, greenValue);
analogWrite(blueLEDPin, blueValue);
}
```

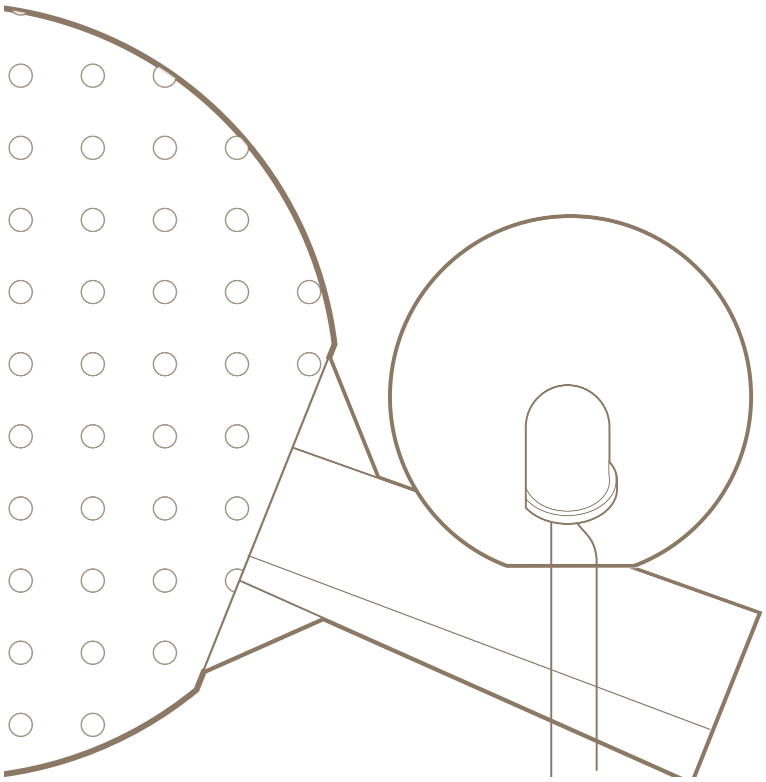
You may notice that the photoresistor's output doesn't range all the way from 0 to 1023. That's okay for this project, but for a more detailed explanation of how to calibrate for the range you're reading, see Project 6.

You'll probably notice that the LED's fading is not linear. When the LED is about at half brightness, it appears to stop getting much brighter. This is because our eyes don't perceive brightness linearly. The brightness of the light depends not only on the level that you `analogWrite()` but also on the distance of the light from the diffuser, the distance of your eye from the light, and the brightness of the light relative to other light in the room.

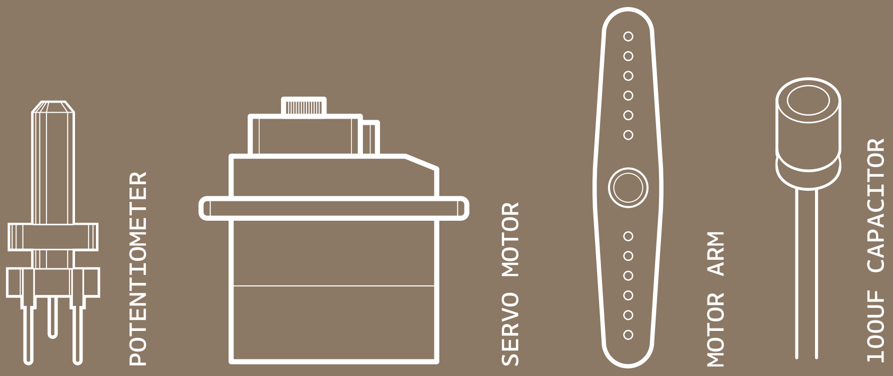
How could you use this to let you know if it's a nice day outside while you're working inside? What other sorts of sensors can you use to control the LED's color?

The LED on its own is pretty neat, but it's not much of a lamp. However, there are a number of different ways you can diffuse the light to make it resemble something like a traditional incandescent. A ping pong ball with a hole cut out for the LED to slide into makes for a nice diffuser. Other ways include covering the light in translucent glue, or sanding the surface of the light. No matter what route you take, you're going to lose at least a little brightness when it's diffused, but it will probably look a lot nicer.

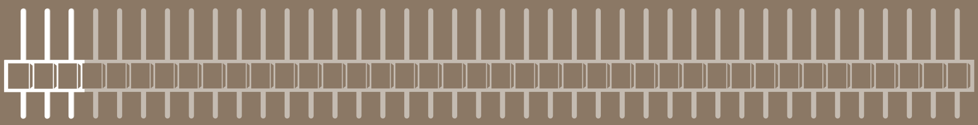
No longer limited to just turning lights on and off, you now have control over how bright or dim something will be. analogWrite() is the function that allows you to PWM components attached to pins 3, 5, 6, 9, 10, or 11, varying the duty cycle.


The ping pong ball cut in order to accommodate the LED
Fig.4

05



MALE HEADER PIN (3 pins)



INGREDIENTS

MOOD CUE

USE A SERVO MOTOR TO MAKE A MECHANICAL GAUGE TO POINT OUT WHAT SORT OF MOOD YOU'RE IN THAT DAY

Discover: mapping values, servo motors, using built-in libraries

Time: **1 HOUR**
Level: ■■□□□

Builds on projects: **1, 2, 3, 4**

Servo motors are a special type of motor that don't spin around in a circle, but move to a specific position and stay there until you tell them to move again. Servos usually only rotate 180 degrees (one half of a circle). Combining one of these motors with a little cardboard craft, you'll be able to let people know if they should come and ask for your help on their next project or not.

Similar to the way you used pulses to PWM an LED in the Color Mixing Lamp Project, servo motors expect a number of pulses that tell them what angle to move to. The pulses always come at the same time intervals, but the width varies between 1000 and 2000 microseconds. While it's possible to write code to generate these pulses, the Arduino software comes with a library that allows you to easily control the motor.

Because the servo only rotates 180 degrees, and your analog input goes from 0-1023, you'll need to use a function called `map()` to change the scale of the values coming from the potentiometer.

One of the great things about the Arduino community are the talented people who extend its functionality through additional software. It's possible for anyone to write libraries to extend the Arduino's functionality. There are libraries for a wide variety of sensors and actuators and other devices that users have contributed to the community. A software library expands the functionality of a programming environment. The Arduino software comes with a number of libraries that are useful for working with hardware or data. One of the included libraries is designed to use with servo motors. In your code, you'll import the library, and all of its functionality will be available to you.

BUILD THE CIRCUIT

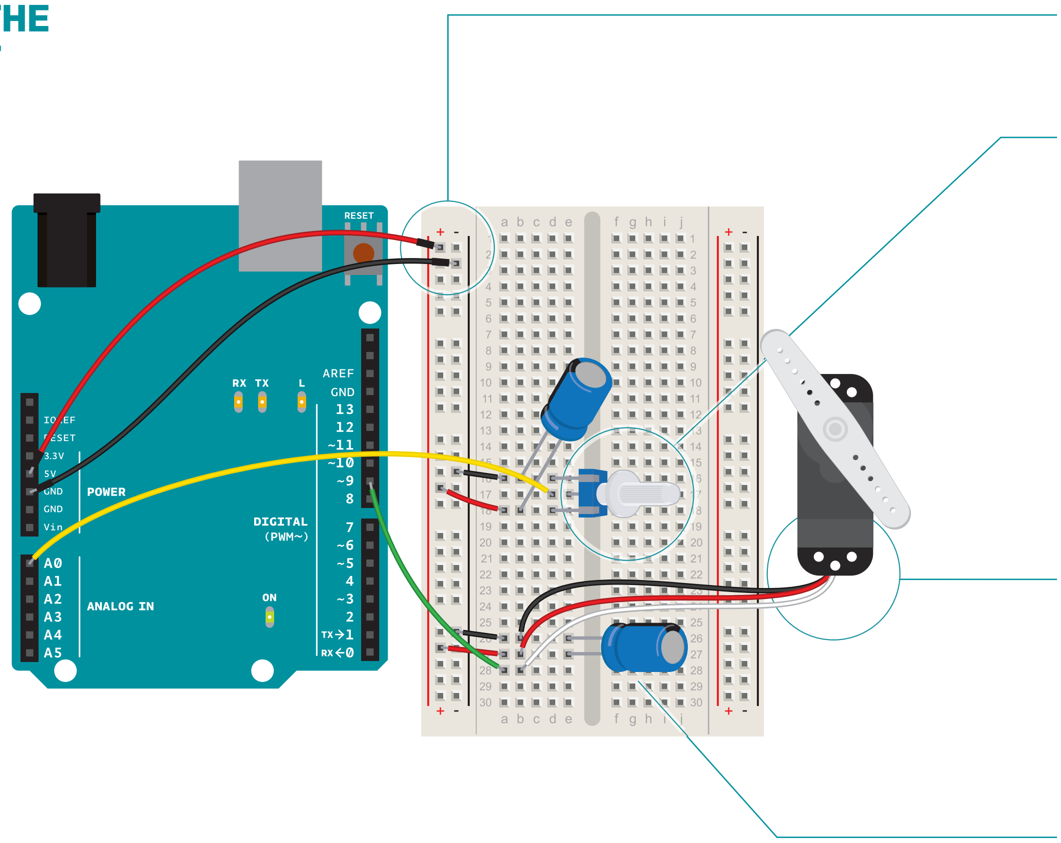


Fig. 1

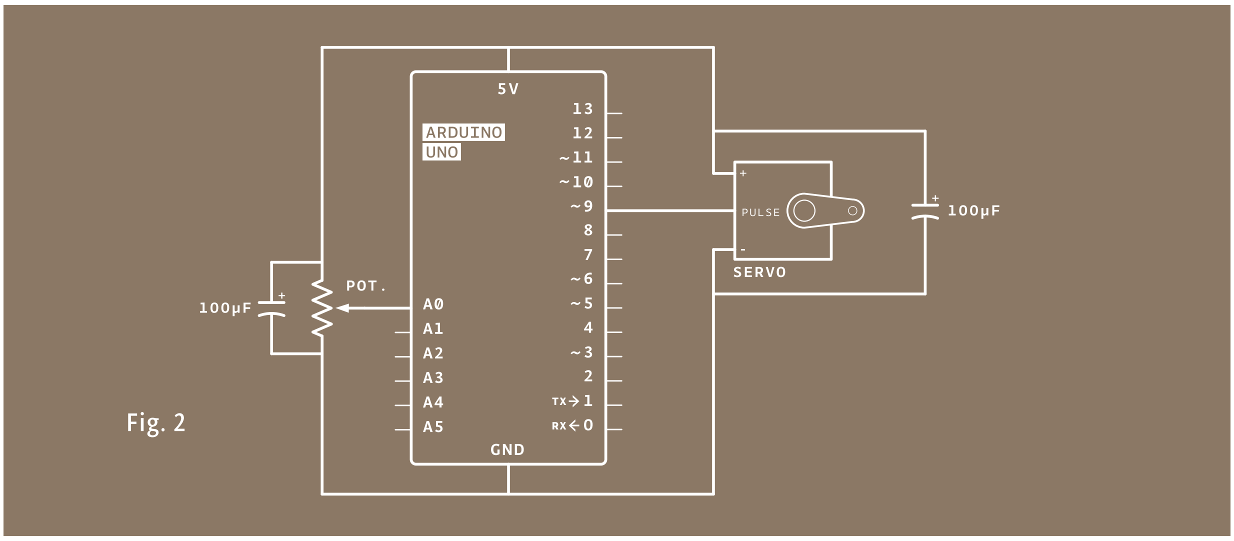


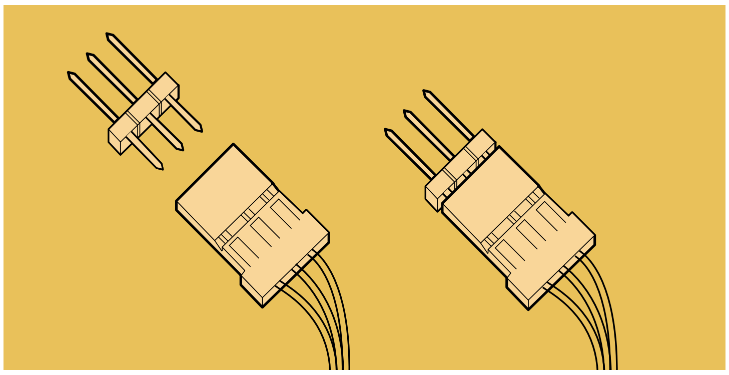
Fig. 2

1 Attach 5V and ground to one side of your breadboard from the Arduino.

2 Place a potentiometer on the breadboard, and connect one side to 5V, and the other to ground. A potentiometer is a type of voltage divider. As you turn the knob, you change the ratio of the voltage between the middle pin and power. You can read this change on an analog input. Connect the middle pin to analog pin 0. This will control the position of your servo motor.

3 The servo has three wires coming out of it. One is power (red), one is ground (black), and the third (white) is the control line that will receive information from the Arduino. Plug three male headers into the female ends of the servo wires (see Fig. 3). Connect the headers to your breadboard so that each pin is in a different row. Connect 5V to the red wire, ground to the black wire, and the white wire to pin 9.

4 When a servo motor starts to move, it draws more current than if it were already in motion. This will cause a dip in the voltage on your board. By placing a 100uf capacitor across power and ground right next to the male headers as shown in Fig. 1, you can smooth out any voltage changes that may occur. You can also place a capacitor across the power and ground going into your potentiometer. These are called ***decoupling capacitors*** because they reduce, or decouple, changes caused by the components from the rest of the circuit. Be very careful to make sure you are connecting the cathode to ground (that's the side with a black stripe down the side) and the anode to power. If you put the capacitors in backwards, they can explode.



Your servo motor comes with female connectors, so you'll need to add header pins to connect it to the breadboard.
Fig. 3

THE CODE

Import the library

To use the servo library, you'll first need to import it. This makes the additions from the library available to your sketch.

Creating the Servo object

To refer to the servo, you're going to need to create a named instance of the servo library in a variable. This is called an ***object***. When you do this, you're making a unique name that will have all the functions and capabilities that the servo library offers. From this point on in the program, every time you refer to `myServo`, you'll be talking to the servo object.

Variable declaration

Set up a named constant for the pin the potentiometer is attached to, and variables to hold the analog input value and angle you want the servo to move to.

Associating the Servo object with the Arduino pin, initializing the serial port

In the `setup()`, you're going to need to tell the Arduino what pin your servo is attached to.

Include a serial connection so you can check the values from the potentiometer and see how they map to angles on the servo motor.

Reading the potentiometer value

In the `loop()`, read the analog input and print out the value to the serial monitor.

Mapping potentiometer value to the servo values

To create a usable value for the servo motor from your analog input, it's easiest to use the `map()` function. This handy function scales numbers for you. In this `case` it will change values between 0-1023 to values between 0-179. It takes five arguments : the number to be scaled (here it's potVal), the minimum value of the input (0), the maximum value of the input (1023), the minimum value of the output (0), and the maximum value of the output (179). Store this new value in the angle variable.
Then, print out the mapped value to the serial monitor.

Rotating the servo

Finally, it's time to move the servo. The command `servo.write()` moves the motor to the angle you specify.
At the end of the `loop()` put a delay so the servo has time to move to its new position.

```
#include <Servo.h>
```

Note that #include instractions have not semicolon at the end of the line.

```
Servo myServo;
```

```
int const potPin = A0;
int potVal;
int angle;
```

```
void setup() {
  myServo.attach(9);

  Serial.begin(9600);
}
```

```
void loop() {
  potVal = analogRead(potPin);
  Serial.print(“potVal: ”);
  Serial.print(potVal);
```

```
  angle = map(potVal, 0, 1023, 0, 179);
  Serial.print(“, angle: ”);
  Serial.println(angle);
```

```
  myServo.write(angle);
  delay(15);
}
```

USE IT

Once your Arduino has been programmed and powered up, open the serial monitor. You should see a stream of values similar to this:

```
potVal : 1023, angle : 179
potVal : 1023, angle : 179
```

When you turn the potentiometer, you should see the numbers change. More importantly, you should see your servo motor move to a new position. Notice the relationship between the value of potVal and angle in the serial monitor and the position of the servo. You should see consistent results as you turn the pot.

One nice thing about using potentiometers as analog inputs is that they will give you a full range of values between 0 and 1023. This makes them helpful in testing projects that use analog input.

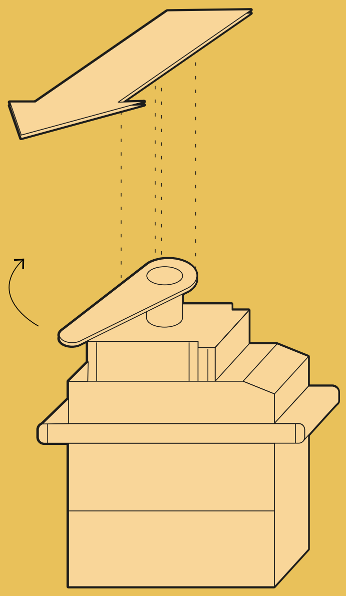
Servo motors are regular motors with a number of gears and some circuits inside. The mechanics inside provide feedback to the circuit, so it is always aware of its position. While it may seem like this is a limited range of motion, it's possible to get it to make a wide variety of different kinds of movements with some additional mechanics. There are a number of resources that describe mechanisms in detail like ***robives.com/mechs*** and the book ***Making Things Move*** by *Dustyn Roberts*.

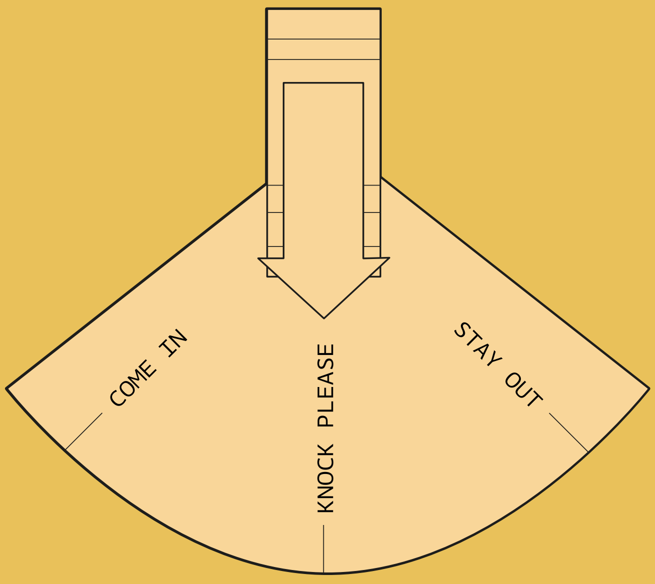
The potentiometer is not the only sensor you can use for controlling the servo. Using the same physical setup (an arrow pointing to a number of different indicators) and a different sensor, what sort of indicator can you make? How would this work with temperature (like in the Love-o-Meter)? Could you tell the time of day with a photoresistor? How does mapping values come into play with those types of sensors?

Servo motors can easily be controlled by the Arduino using a library, which is a collection of code that extends a programming environment. Sometimes it is necessary to repurpose values by mapping them from one scale to another.

Now that you're up and running with motion, it's time to let people know if you're available to help them on their projects, or if you want to be left alone to plan your next creation.

With scissors, cut out a piece of cardboard in the shape of an arrow. Position your servo to 90 degrees (check the angle value in the serial monitor if you're unsure). Tape the arrow so it's oriented in the same direction as the motor's body. Now you should be able to rotate the arrow 180 degrees when turning the potentiometer. Take a piece of paper that is larger than the servo with the arrow attached and draw a half circle on it. On one end of the circle, write "Stay Out". On the other end, write "Come in". Put "Knock please!" in the middle of the arc. Place the servo with the arrow on top of the paper. Congratulations, you've got a way to tell people just how busy you are with your projects!





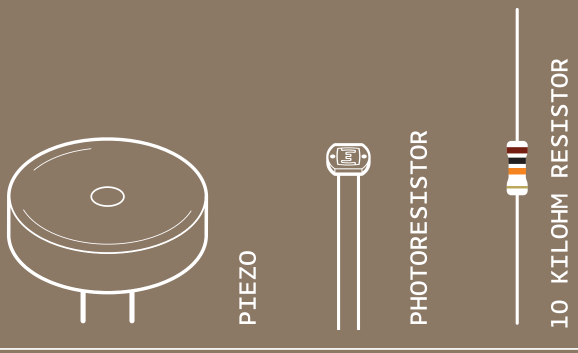
1

Attach a paper arrow to the servo arm.

2

Design a paper base and place it under the servo.

06



INGREDIENTS

LIGHT THEREMIN

TIME TO MAKE SOME NOISE! USING A PHOTORESISTOR AND A PIEZO ELEMENT, YOU'RE GOING TO MAKE A LIGHT-BASED THEREMIN

Discover: making sound with the tone() function, calibrating analog sensors

Time: **45 MINUTES**

Level: ■■■■□

Builds on projects: **1, 2, 3, 4**

A *theremin* is an instrument that makes sounds based on the movements of a musician's hands around the instrument. You've probably heard one in scary movies. The theremin detects where a performer's hands are in relation to two antennas by reading the capacitive change on the antennas. These antennas are connected to analog circuitry that create the sound. One antenna controls the frequency of the sound and the other controls volume. While the Arduino can't exactly replicate the mysterious sounds from this instrument, it is possible to emulate them using the `tone()` function. Fig. 1 shows the difference between the pulses emitted by `analogWrite()` and `tone()`. This enables a transducer like a speaker or piezo to move back and forth at different speeds.

Notice how the signal is low most of the time, but the frequency is the same as PWM 200.

Notice how the voltage is high most of the time, but the frequency is the same as PWM 50.

The duty cycle is 50% (on half the time, off half the time), but the frequency changes.

Same duty cycle as Tone 440; but twice the frequency.

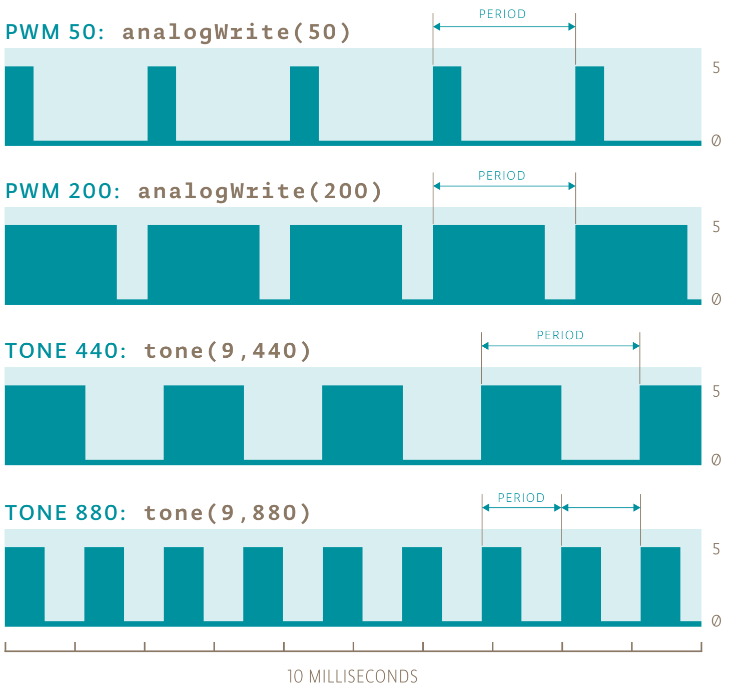
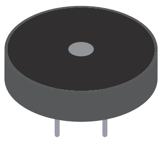


Fig. 1

Instead of sensing capacitance with the Arduino, you'll be using a photoresistor to detect the amount of light. By moving your hands over the sensor, you'll change the amount of light that falls on the photoresistor's face, as you did in Project 4. The change in the voltage on the analog pin will determine what frequency note to play.
You'll connect the photoresistors to the Arduino using a voltage divider circuit like you did in Project 4. You probably noticed in the earlier project that when you read this circuit using `analogRead()`, your readings didn't range all the way from 0 to 1023. The fixed resistor connecting to ground limits the low end of the range, and the brightness of your light limits the high end. Instead of settling for a limited range, you'll calibrate the sensor readings getting the high and low values, mapping them to sound frequencies using the `map()` function to get as much range out of your theremin as possible. This will have the added benefit of adjusting the sensor readings whenever you move your circuit to a new environment, like a room with different light conditions.



A ***piezo*** is a small element that vibrates when it receives electricity. When it moves, it displaces air around it, creating sound waves.

BUILD THE CIRCUIT

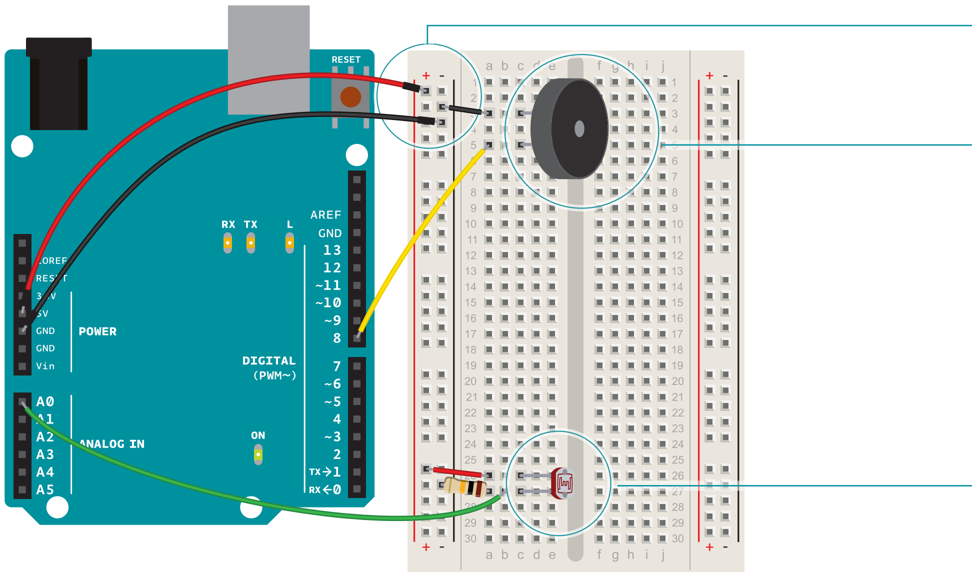


Fig. 2

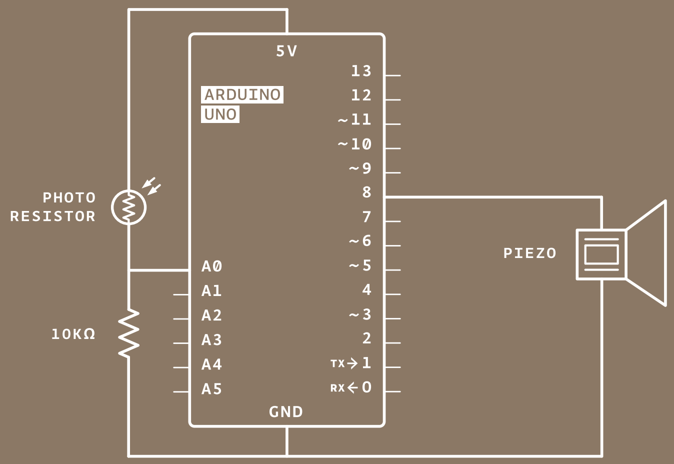


Fig. 3

Traditional theremins can control the frequency and the volume of sound. In this example, You'll be able to control the frequency only. While you can't control the volume through the Arduino, it is possible to change the voltage level that gets to the speaker manually. What happens if you put a potentiometer in series with pin 8 and the piezo? What about another photoresistor?

1. On your breadboard, connect the outer bus lines to power and ground.

2. Take your piezo, and connect one end to ground, and the other to digital pin 8 on the Arduino.

3. Place your photoresistor on the breadboard, connecting one end to 5V. Connect the other end to the Arduino's analogIn pin 0, and to ground through a 10-kilohm resistor. This circuit is the same as the voltage divider circuit in Project 4.

THE CODE

Create variables for calibrating the sensor

Create a variable to hold the `analogRead()` value from the photoresistor. Next, create variables for the high and low values. You're going to set the initial value in the sensorLow variable to 1023, and set the value of the `sensorHigh` variable to 0. When you first run the program, you'll compare these numbers to the sensor's readings to find the real maximum and minimum values.

Name a constant for your calibration indicator

Create a constant named `ledPin`. You'll use this as an indicator that your sensor has finished calibrating. For this project, use the on-board LED connected to pin 13.

Set digital pin direction and turn it high

In the `setup()`, change the `pinMode()` of ledPin to OUTPUT, and turn the light on.

Use a while() loop for calibration

The next steps will calibrate the sensor's maximum and minimum values. You'll use a `while()` statement to run a loop for 5 seconds. `while()` loops run until a certain condition is met. In this case you're going to use the `millis()` function to check the current time. `millis()` reports how long the Arduino has been running since it was last powered on or reset.

Compare sensor values for calibration

In the loop, you'll read the value of the sensor; if the value is less than `sensorLow` (initially 1023), you'll update that variable. If it is greater than `sensorHigh` (initially 0), that gets updated.

Indicate calibration has finished

When 5 seconds have passed, the while() loop will end. Turn off the LED attached to pin 13. You'll use the sensor high and low values just recorded to scale the frequency in the main part of your program.

```
int sensorValue;
int sensorLow = 1023;
int sensorHigh = 0;
```

```
const int ledPin = 13;
```

```
void setup() {

  pinMode(ledPin, OUTPUT);
  digitalWrite(ledPin, HIGH);
```

```
  while (millis() < 5000) {
```

`while()`
arduino.cc/while

```
    sensorValue = analogRead(A0);
    if (sensorValue > sensorHigh) {
      sensorHigh = sensorValue;
    }
    if (sensorValue < sensorLow) {
      sensorLow = sensorValue;
    }
  }
```

```
  digitalWrite(ledPin, LOW);
}
```

Read and store the sensor value

In the `loop()`, read the value on A0 and store it in `sensorValue`.

Map the sensor value to a frequency

Create a variable named `pitch`. The value of `pitch` is going to be mapped from `sensorValue`. Use `sensorLow` and `sensorHigh` as the bounds for the incoming values. For starting values for output, try 50 to 4000. These numbers set the range of frequencies the Arduino will generate.

Play the frequency

Next, call the `tone()` function to play a sound. It takes three arguments : what pin to play the sound on (in this case pin 8), what frequency to play (determined by the `pitch` variable), and how long to play the note (try 20 milliseconds to start).

Then, call a `delay()` for 10 milliseconds to give the sound some time to play.

USE IT

When you first power the Arduino on, there is a 5 second window for you to calibrate the sensor. To do this, move your hand up and down over the photoresistor, changing the amount of light that reaches it. The closer you replicate the motions you expect to use while playing the instrument, the better the calibration will be.

After 5 seconds, the calibration will be complete, and the LED on the Arduino will turn off. When this happens, you should hear some noise coming from the piezo! As the amount of light that falls on the sensor changes, so should the frequency that the piezo plays.

```
void loop() {
  sensorValue = analogRead(A0);
```

```
  int pitch =
    map(sensorValue,sensorLow,sensorHigh, 50, 4000);
```

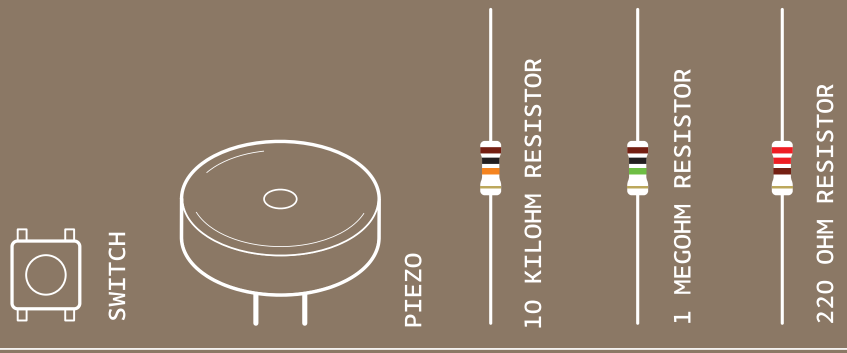
```
  tone(8,pitch,20);



  delay(10);
}
```

The range in the `map()` function that determines the `pitch` is pretty wide, try changing the frequencies to find ones that are the right fit for your musical style.

The `tone()` function operates very much like the PWM in `analogWrite()` but with one significant difference. In `analogWrite()` the frequency is fixed; you change the ratio of the pulses in that period of time to vary the duty cycle. With `tone()` you're still sending pulses, but changing the frequency of them. `tone()` always pulses at a 50% duty cycle (half the time the pin is high, the other half the time it is low).

The tone() function gives you the ability to generate different frequencies when it pulses a speaker or piezo. When using sensors in a voltage divider circuit, you probably won't get a full range of values between 0-1023. By calibrating sensors, it's possible to map your inputs to a useable range.

07



INGREDIENTS

KEYBOARD INSTRUMENT

WITH FEW RESISTORS AND BUTTONS YOU ARE GOING TO BUILD A SMALL MUSICAL KEYBOARD

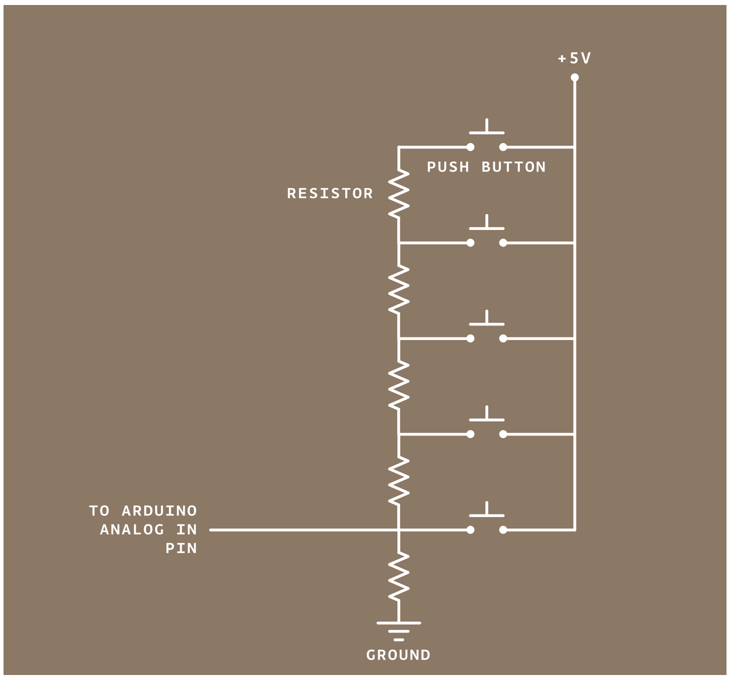
Discover: resistor ladders, arrays

Time: **45 MINUTES**
Level: ■■■■□

Builds on projects: **1, 2, 3, 4, 6**

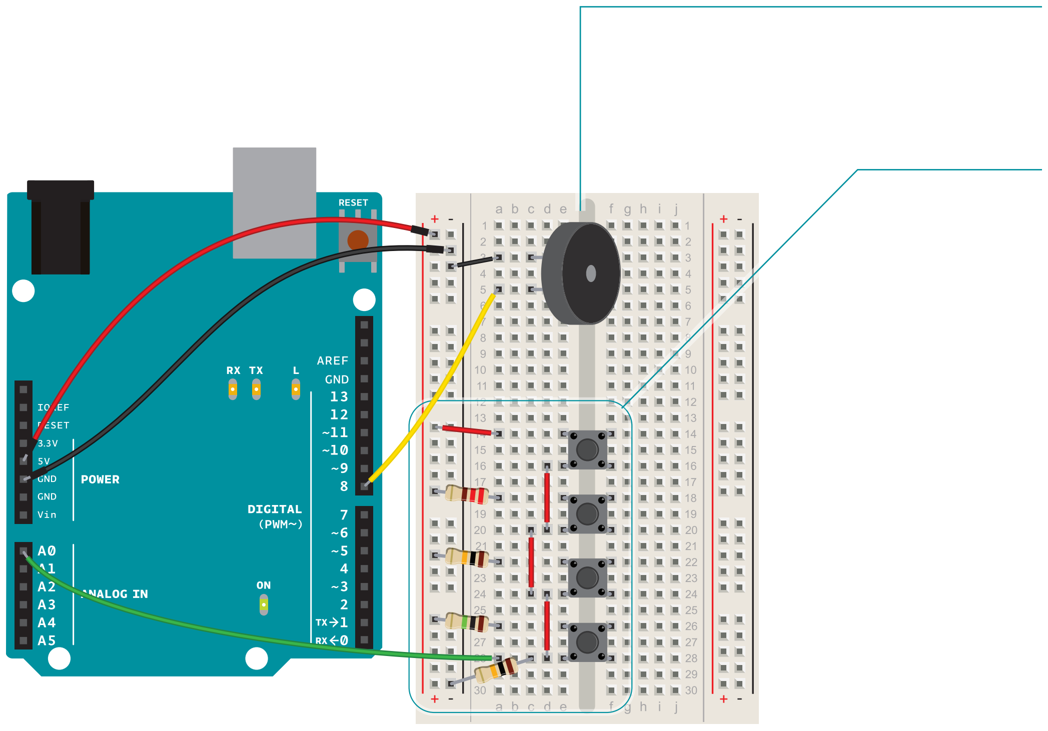
While it's possible to simply hook up a number of momentary switches to digital inputs to key of different tones, in this project, you'll be constructing something called a ***resistor ladder****.*

This is a way to read a number of switches using the analog input. It's a helpful technique if you find yourself short on digital inputs. You'll hook up a number of switches that are connected in parallel to analog in 0. Most of these will connect to power through a resistor. When you press each button, a different voltage level will pass to the input pin. If you press two buttons at the same time, you'll get a unique input based on the relationship between the two resistors in parallel.



A resistor ladder and five switches as analog input.
Fig. 1

BUILD THE CIRCUIT



The arrangement of resistors and switches feeding into an analog input is called a resistor ladder.

Fig. 2

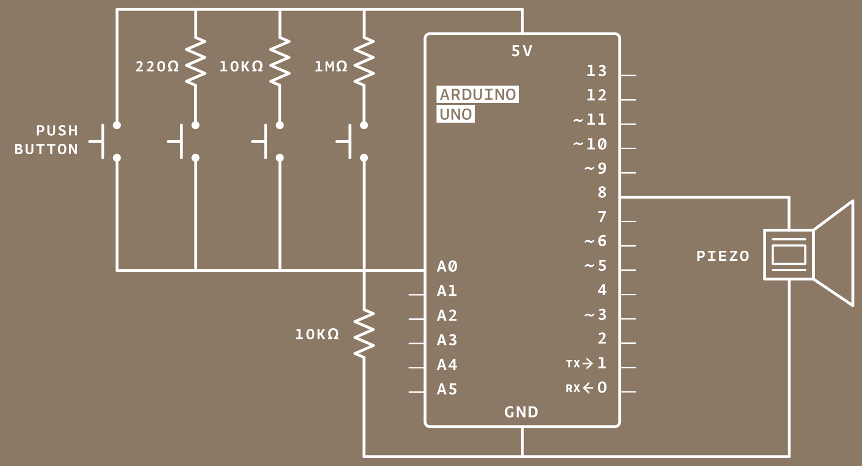
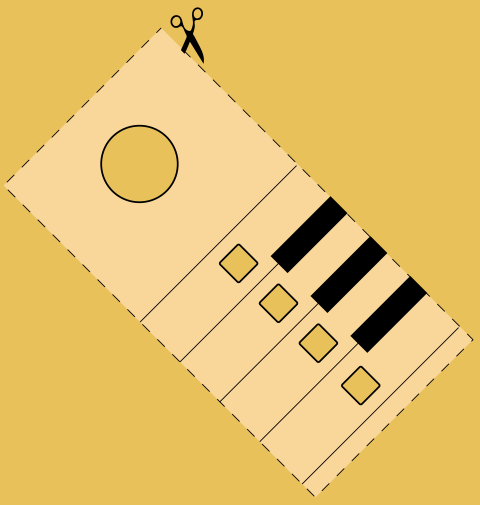


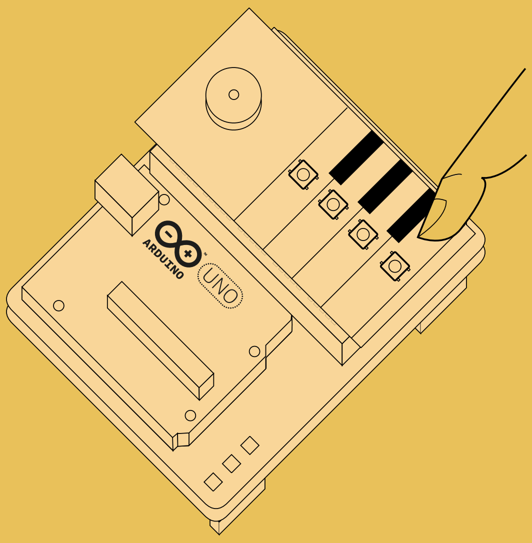
Fig. 3

1 Wire up your breadboard with power and ground as in the previous projects. Connect one end of the piezo to ground. Connect the other end to pin 8 on your Arduino.

2 Place your switches on the breadboard as shown in the circuit. The arrangement of resistors and switches feeding into an analog input is called a resistor ladder. Connect the first one directly to power. Connect the second, third and fourth switches to power through a 220-ohm, 10-kilohm and 1-megohm resistor, respectively. Connect all the switches' outputs together in one junction. Connect this junction to ground with a 10-kilohm resistor, and also connect it to Analog In 0. Each of these acts as a voltage divider.

Think about an enclosure for the keyboard. While old analog synthesizers had wires poking out all over the place, your keyboard is sleek and digital. Prepare a small piece of cardboard that can be cut out to accommodate your buttons. Label the keys, so you know what notes are triggered by each key.





1 Draw and cut a piece of paper with holes for the four buttons and piezo. Decorate it to look like a piano keyboard.

2 Position the paper over the buttons and piezo. Enjoy your creation!

THE CODE

The array

In this program, you'll need to keep a list of frequencies you want to play when you press each of your buttons. You can start out with the frequencies for middle C, D, E and F (262Hz, 294Hz, 330Hz, and 349Hz). To do this, you'll need a new kind of variable called an array.
An array is a way to store different values that are related to each other, like the frequencies in a musical scale, using only one name. They are a convenient tool for you to quickly and efficiently access information. To declare an array, start as you would with a variable, but follow the name with a pair of square brackets: []. After the equals sign, you'll place your elements in curly brackets.
To read or change the elements of the array, you reference the individual element using the array name and then the index of the item you want to address. The index refers to the order in which the items appear when the array is created. The first item in the array is item 0, the second is item 1, and so forth.

Create an array of frequencies

Set up an array of four notes using the frequencies listed above. Make this array a global variable by declaring it before the `setup()`.

Begin serial communication

In your `setup()`, start serial communication with the computer.

Read the analog value and send it to the serial monitor

In the `loop()`, declare a local variable to hold the value read on pin A0. Because each switch has a different resistor value connecting it to power, each will have a different value associated with it. To see the values, add the line `Serial.println(keyVal)` to send to the computer.

Use an if()...else statement to determine what note to play

Using an `if()`...`else` statement, you can assign each value to a different tone. The values included in the example program are ballpark figures for these resistor sizes. As all resistors have some tolerance for error, these may not work exactly for you. Use the information from the serial monitor to adjust as necessary.

```
int buttons[6];
// set up an array with 6 integers

int buttons[0] = 2;
// give the first element of the array the value 2
```

```
int notes[] = {262,294,330,349};
```

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

```
void loop() {
  int keyVal = analogRead(A0);
  Serial.println(keyVal);
```

```
  if(keyVal == 1023){
    tone(8, notes[0]);
  }
```

Play the notes that correspond to the analog value

After each `if()` statement, call the `tone()` function. The program references the array to determine what frequency to play. If the value of A0 matches one of your if statements, you can tell the Arduino to play a tone. It's possible your circuit is a little "noisy" and the values may fluctuate a little bit while pressing a switch. To accommodate for this variation, it's a good idea to have a small range of values to check against. If you use the comparison `"&&"`, you can check multiple statements to see if they are true.

If you press the first button, notes[0] will play. If you press the second, notes[1] will play, and if you press the third, notes[2] will play. This is when arrays become really handy.

Stop playing the tone when nothing is pressed

Only one frequency can play on a pin at any given time, so if you're pressing multiple keys, you'll only hear one sound.

To stop playing notes when there is no button being pressed, call the `noTone()` function, providing the pin number to stop playing sound on.

USE IT

If your resistors are close in value to the values in the example program, you should hear some sounds from the piezo when you press the buttons. If not, check the serial monitor to make sure each of the buttons is in a range that corresponds to the notes in the `if()`...`else` statement. If you're hearing a sound that seems to stutter, try increasing the range a little bit.

Press multiple buttons at the same time, and see what sort of values you get in the serial monitor. Use these new values to trigger even more sounds. Experiment with different frequencies to expand your musical output. You can find frequencies of musical notes on this page: ***arduino.cc/frequencies***

If you replace the switches and resistor ladder with analog sensors, can you use the additional information they give you to create a more dynamic instrument? You could use the value to change the duration of a note or, like in the Theremin Project, create a sliding scale of sounds.

```
  else if(keyVal >= 990 && keyVal <= 1010){
    tone(8, notes[1]);
  }
  else if(keyVal >= 505 && keyVal <= 515){
    tone(8, notes[2]);
  }
  else if(keyVal >= 5 && keyVal <= 10){
    tone(8, notes[3]);
  }
```

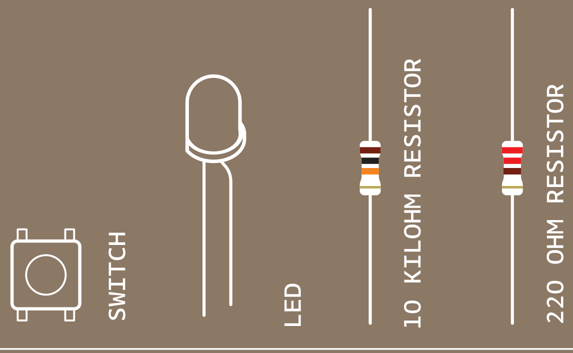
```
  else{
    noTone(8);
  }
}
```

The `tone()` function is fun for generating sounds, but it does have a few limitations. It can only create square waves, not smooth sine waves or triangles. Square waves don't look much like waves at all. As you saw in Fig. 1 in Project 6, it's a series of on and off pulses.

As you start your band, keep some things in mind : only one tone can play at a time and `tone()` will interfere with `analogWrite()` on pins 3 and 11.

Arrays are useful for grouping similar types of information together; they are accessed by index numbers which refer to individual elements. Resistor ladders are an easy way to get more digital inputs into a system by plugging into an analog input.

08



INGREDIENTS

DIGITAL HOURGLASS

IN THIS PROJECT, YOU'LL BUILD A DIGITAL HOURGLASS THAT TURNS ON AN LED EVERY TEN MINUTES. KNOW HOW LONG YOU'RE WORKING ON YOUR PROJECTS BY USING THE ARDUINO'S BUILT-IN TIMER

Discover: long data type, creating a timer

Time: **30 MINUTES**
Level: ■■■□□

Builds on projects: **1, 2, 3, 4**

Up to now, when you've wanted something to happen at a specific time interval with the Arduino, you've used delay(). This is handy, but a little confining. When the Arduino calls delay(), it freezes its current state for the duration of the delay. That means there can be no other input or output while it's waiting. Delays are also not very helpful for keeping track of time. If you wanted to do something every 10 seconds, having a 10 second delay would be fairly cumbersome.

The `millis()` function helps to solve these problems. It keeps track of the time your Arduino has been running in milliseconds. You used it previously in Project 6 when you created a timer for calibration.

So far you've been declaring variables as `int`. An `int` (integer) is a 16-bit number, it holds values between -32,768 and 32,767. Those may be some large numbers, but if the Arduino is counting 1000 times a second with `millis()`, you'd run out of space in less than a minute. The `long` datatype holds a 32-bit number (between -2,147,483,648 and 2,147,483,647). Since you can't run time backwards to get negative numbers, the variable to store `millis()` time is called an `unsigned long`. When a datatype is called *unsigned*, it is only positive. This allows you to count even higher. An `unsigned long` can count up to 4,294,967,295. That's enough space for `milis()` to store time for almost 50 days. By comparing the current `millis()` to a specific value, you can see if a certain amount of time has passed.
When you turn your hourglass over, a tilt switch will change its state, and that will set off another cycle of LEDs turning on.
The tilt switch works just like a regular switch in that it is an on/off sensor. You'll use it here as a digital input. What makes tilt switches unique is that they detect orientation. Typically they have a small cavity inside the housing that has a metal