Welcome to ArduinoFest!

ArduinoFest is a festival of electronics and programming traditionally held in December. It is an intensive series of projects using microcontrollers, electronics components, and C programming that will prepare you for post-New-Year projects in Entrepreneurship, Engineering, CSRSEF, and life.

It is very important that you understand expectations from the beginning—sophomores will confirm most of the following, some from painful experience:

Projects: This handbook gives you instructions and specifications for completing 16 projects and, if you desire, a capstone project. The first 9 projects are required. You may do additional projects to earn the additional XP that you want to earn from this unit. You do not have to read this entire book, but at least get familiar with the introduction and the list of projects, particularly the required ones.

Points: Each project is worth approximately 20 XP for AP CSP and 15 XP for Entrepreneurship. Completing a capstone project is worth three times as much as a normal project. In Skyward, the target for this entire unit will be 240 XP for AP CSP and 180 XP for Entrepreneurship.

Proof: To earn XP for a project you must have it checked off by Ms. Wrenchey, Mr. Christensen, or Zach (if he is available). There are two ways to do this:

1. Sign up through Forms and have your project checked off during class or office hours. If things are busy you may have to wait until it is your turn. This is the preferred way to get checked off.

2. Alternatively, create a video and submit it through Flipgrid. Your video must include a slow walk-through of your breadboard and code that makes it easy for us to see that you have done the work and understand the project. You should make an appearance in the video so that we can see it is actually you, and your breadboard should clearly have your initials written on it in permanent marker. Your video should not be more than 90 seconds long.

Assistance: Your primary source for help should be Ms. Wrenchey, Mr. Christensen, or Zach (if he is available). If you have fallen behind or need more urgent help, ask Mr. Christensen or Ms. Wrenchey and we will assign you one of the “mentors” that we have hand-picked for this job. If you need help, ask for it. You might even get a friend out of the deal.

Procrastination: We will measure attendance based on your regular completion of projects. For your sake (not ours!), do not fall behind. The three weeks go by quickly and you must be disciplined about getting a project done every day or two.

Preparation: Before the first day of ArduinoFest you must:

1. Be in physical possession of your Tesla STEM Arduino kit;
2. Be able to connect your Arduino to a USB port on your LWSD PC; and
3. Have downloaded the Arduino IDE software to your LWSD PC from Software Center.

We hope you enjoy these three weeks of Entrepreneurship and Computer Science!

Ms. Wrenchey, Mr. Christensen, and Zach Barth

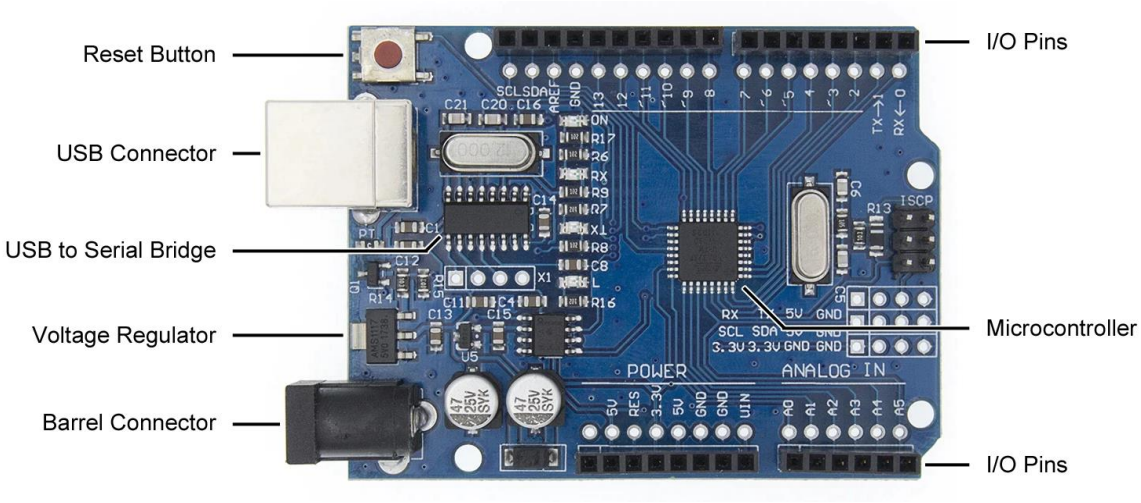
Table of Contents

The Arduino Uno

The most important part in your kit is the Arduino Uno microcontroller board. A *microcontroller* is a very small computer that is inexpensive enough to be used for a single purpose, like a television remote control, quadcopter drone, or blood glucose meter. Contrast this with a general-purpose computer like your laptop or cellphone, which is much more powerful and expensive and can be used to do many more things. The Arduino can store and run exactly one program at a time, so every project you do will require you to write and upload a new program to replace the previous one.

The name “Arduino Uno” refers to the board as a whole, which was designed by an organization that was founded in Italy. There are other boards in the family, like the Arduino Nano (which is smaller) and the Arduino Mega (which is more powerful). Although you can buy an official Arduino board from the Arduino website, the design is *open-source* and can be manufactured and sold by anyone. The Arduino Uno in your kit was manufactured by an unknown company in China and purchased for about one tenth the price of an official one.

The actual microcontroller on your Arduino Uno is a small *integrated circuit* (or “IC”, or “chip”) called the ATmega328P, made by a company called Atmel. It runs at 16 MHz (megahertz) and has 2 KB (kilobytes) of memory and 32 KB of storage. For comparison, your laptop is about 150x faster (but 1,000x to 1,000,000x more powerful) with 2,000,000x the memory and 15,000,000x the storage.

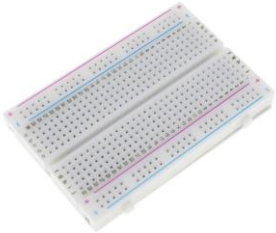


Power: Your board must be powered, using either USB (from your PC or a phone charger) or the *barrel connector* (with the included battery pack). Without power the board will not work. When plugged into a PC using a USB cable the Arduino can also be programmed using the Arduino IDE.

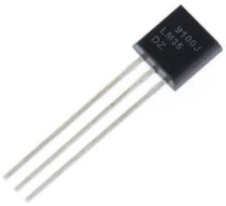
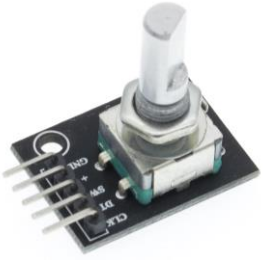
Reset Button: If you press this it will reset the Arduino and restart your program. Downloading a new program to the board will also cause it to reset and start running the new program.

I/O Pins: Unlike PCs, which are designed to interface with humans, microcontrollers are designed to interface directly with electronics components like the other parts in your kit. To do this they have many *input/output pins* (or “I/O pins”) that can be controlled by your program to measure and generate electrical signals. For our purposes, a *digital* signal is either 5 volts (one, true, on, high, etc.) or 0 volts (zero, false, off, low, etc.), while an *analog* signal can be any voltage in-between.

Other Components

A-to-B USB cable: This is used to connect your Arduino to your PC so that it has power and can be programmed. As you will learn in Project 0, it can also be used to send text from the Arduino back to your PC, which will be useful for debugging.	
Breadboard: Also called a *solderless breadboard*, this allows you to construct temporary circuits by pushing wires into the holes, as opposed to soldering together a more permanent circuit board. Behind the holes are strips of metal that electrically connect groups of holes together as if they were a single wire, as shown in the second picture to the right. The long strips on the sides of the breadboard are typically used to provide power to parts by connecting them to power (5 V) and ground (0 V). Many years ago, before solderless breadboards were invented, people would build circuits by attaching components and wires to wooden boards, some of which were literally cutting boards used for slicing bread. The Wikipedia page for breadboards has a lot of great pictures of breadboards and the circuit construction techniques that preceded them.	 
Jumper wires: These are the wires that you will use to create electrical connections between your microcontroller, breadboard, and other parts. Your kit has two types, short M-to-M and long M-to-F, which both come in many colors. They all do exactly the same thing: create an electrical connection between two points.	
Battery pack: This battery pack takes 4x alkaline AA batteries and allows you to power your Arduino without having a PC (or USB charger) attached. Although it seems wasteful, you should **not** use rechargeable batteries (NiCad or NiMH), as their voltage (1.2 V, vs. 1.5 V for alkaline batteries) is not enough to power the Arduino and its voltage regulator. You should also **not** use the battery pack at the same time you are using the USB cable.	

The rest of the parts in your kit are introduced as they are used. The numbers in parentheses below indicate which projects introduce which parts. Note that the quantities shown in the images below do not necessarily match the quantities of parts in your kit.

LEDs (1) 	Resistors (1) 	Switches (2) 	Potentiometers (3) 
Photoresistors (4) 	RGB LEDs (5) 	Piezo buzzers (6) 	Buttons (7) 
Servo motors (8) 	Temperature sensor (9) 	X/Y joystick (10) 	Touch sensor (11) 
7-segment display (12) 	Ultrasonic ranger (13) 	Rotary encoder (14) 	Character LCD (15) 

Project 0: Serial Monitor

Universal Serial Bus (or “USB”) was invented in the 1990s as an attempt to unify the various interfaces used to connect peripherals (external devices) to PCs, like PS/2 ports (keyboards and mice), parallel ports (printers and scanners), serial ports (modems), and game ports (joysticks and gamepads). USB can be used to connect almost any type of peripheral to a PC. It removes the need for manual configuration (a huge hassle in pre-2000s PCs) and even provides a modest amount of power, removing the need for external power supplies for many devices.

The downside is that USB is exceptionally complicated, at least compared to the ports listed above that it replaced. Fortunately, your Arduino includes a neat little integrated circuit called the CH340G that actually *simulates* a serial port (yes, the outdated one mentioned above) and allows you to send ASCII text, one byte at a time, back and forth between your Arduino and the PC using a USB cable (which also conveniently provides power to the Arduino and allows it to run).

In many of the projects that follow you will use the USB serial connection to send text from your Arduino to your PC, which can then be viewed in the Arduino IDE’s *serial monitor*. This is incredibly useful for debugging, as otherwise it can be very difficult to tell what an Arduino program is doing. Although it is possible to send data in the other direction (from the PC to the Arduino), you will not do that in any of these projects.

Example Program:

Connect your Arduino to your PC and upload the following program:

```
void setup()
{
      // Open the serial connection:
      Serial.begin(9600);
}

void loop()
{
      // Print a message twice a second:
      Serial.println("Hello, world!");
      delay(500);
}
```

If you open the serial monitor in the Arduino IDE (“Tools > Serial Monitor”, or Ctrl+Shift+M) you should see the text “Hello, world!” (without quotes) printed to the screen twice each second. This data is sent over the USB cable, so if your Arduino is not connected to your PC you will not see any text appear!

If you are getting errors in the Arduino IDE, make sure that you have typed `println` with a lowercase `L` and not `printIn` with an uppercase `I`. The two letters look very similar in many fonts. Also, make sure that you have used double quotes like `"` and not single quotes like `'`. Although they are interchangeable in Python they mean something completely different in C.

- Although the C programming language has a lot in common with Python they look very different. C uses semi-colons (`;`) to mark the end of a statement in the same way that Python uses newlines (the thing that happens when you hit the “Enter” key). C also uses curly braces (`{` and `}`) to enclose blocks of code in the same way that Python requires you to keep blocks of code indented to the same level. This book will point out more similarities and differences between the two programming languages in future projects.
- Any text that follows `//` in a C program is a *comment* and is ignored by the compiler. By now you have probably been told too many times that you should be using comments to document your code. But what does that *mean*? The author of this document writes code for a living, and typically uses comments to summarize and separate blocks of code, the same way you might use headers and section dividers in a long text document (like this one) to make it easier to read. Comments are also useful for explaining non-obvious code, like a complicated algorithm.
- Every Arduino program must contain two special functions: `setup()` and `loop()`. The `setup()` function is called exactly once when your program first starts and is where your program should perform its one-time initialization tasks. After that finishes, your `loop()` function will be called repeatedly until the Arduino is unplugged or restarted. This is where your main program logic will go.
- `Serial.println()` is a function that will send whatever you specify (text, numbers, etc.) from your Arduino to your PC to be displayed in the serial monitor. In the code above it is used to send a specific *text string*, `"Hello, world!"`, to your PC.
- There is also a function called `Serial.print()` that does exactly the same thing except that it does not automatically advance to the next line like `Serial.println()`.
- If you want to use any of the `Serial` functions you will need to first call `Serial.begin()` to open the connection to your PC in your `setup()` function. The number specified in parenthesis is something called the *bit rate* or *baud*, which is the speed at which data is sent (in bits per second). You should leave this at 9600.
- The `delay()` function is used to pause the program for a specified amount of time (in milliseconds). There are 1000 milliseconds in a second, so this program prints its message twice per second.

Assignment:

Using the program above as a starting point, create a program that prints the worst “Dad joke” you know (or can find on the internet) to the serial monitor. Use the `delay()` function to ensure that your joke has the correct comedic timing (i.e., delay between the joke and the punchline).

Additional XP: Add an additional creative flair, like a few animated dots between the joke and the punchline to build up the suspense for someone watching. Remember that you can use `Serial.print()` to print text without automatically advancing to the next line.

Project 1: LEDs

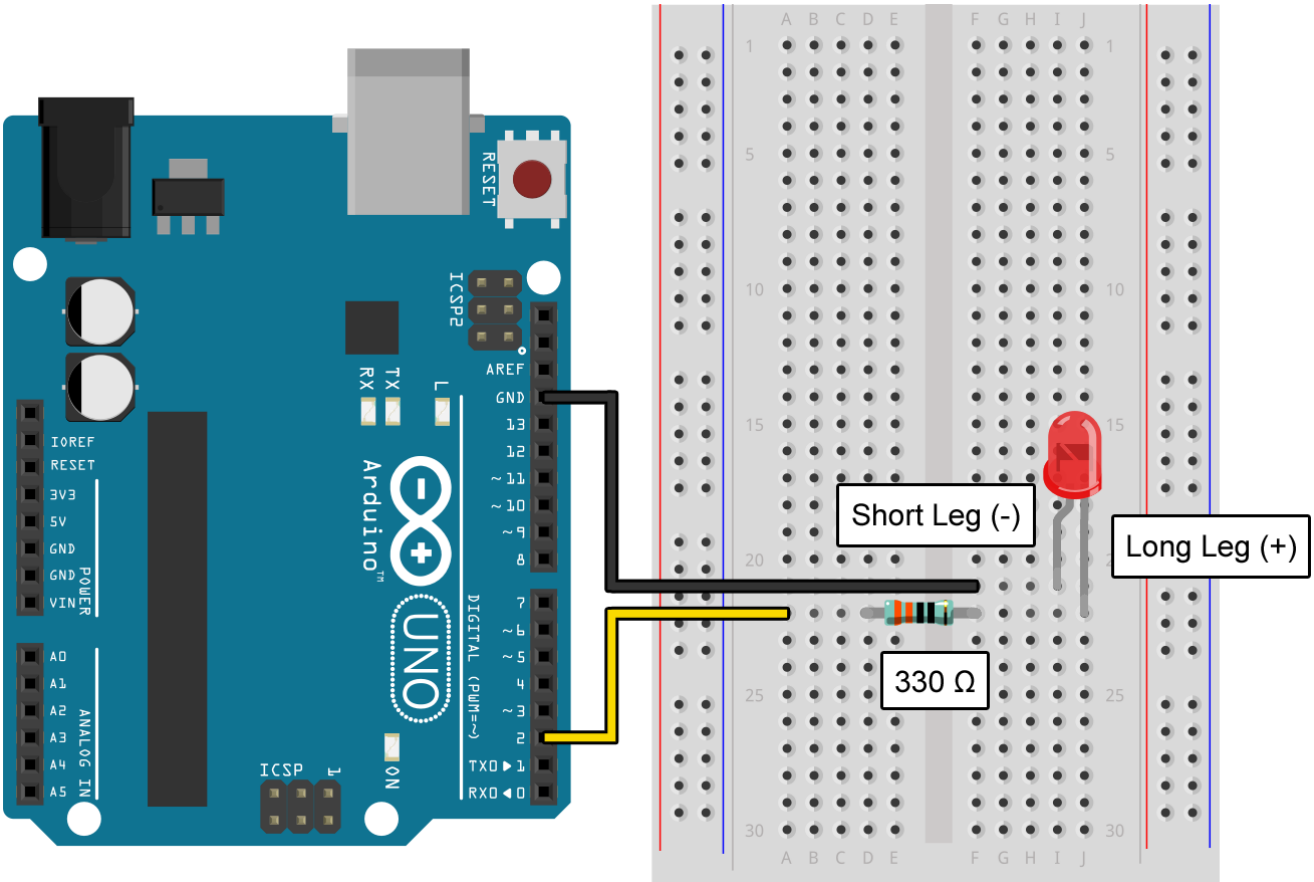
Diodes are a type of *semiconductor* that only allow current to flow in one direction. They are very useful, but not for our purposes, so they are not included in your kit.

Light-emitting diodes (or “LEDs”), on the other hand, light up when current flows through them, which is much more fun. Unlike old-fashioned *incandescent* lightbulbs they do not limit the amount of current that passes through them, and so they must always be used in conjunction with a *resistor* so that they do not “burn out.” The LEDs in your kit have two “legs” of different length; the shorter leg should always be wired closer to *ground* (0 V or GND).

Resistors “resist” the flow of current and are used to limit the amount of current in a circuit. The amount of resistance of each resistor is measured in ohms (Ω) and is indicated by the difficult to read colored stripes on the resistor. Your kit contains three types of resistors (330 Ω, 10 kΩ, and 100 kΩ) and a helpful card for telling them apart.

Example Program:

Assemble the following circuit and upload the program to your Arduino:



```
void setup()
{
      pinMode(2, OUTPUT);
}

void loop()
{
      // Blink the LED on pin 2 once per second:
      digitalWrite(2, HIGH);
      delay(500);
      digitalWrite(2, LOW);
      delay(500);
}
```

If your program has uploaded without any errors you should see the LED blinking on and off once per second. If your LED is not blinking you may have it connected backward; verify that the longer leg is connected to pin 2 and the shorter leg is connected to ground (“GND”).

- By using a resistor in series with the LED you have limited the amount of current that will flow through it so that it will not “burn out.” Although it would be most precise to measure it with an *ammeter* (a device that measures current), you can estimate the amount of current flowing through the LED with Ohm’s law: V = I * R, so I = V / R = 5 V / 330 Ω = 15 mA. It is surprisingly difficult to identify the maximum current for generic LEDs like the ones in your kit, but anything below 20 mA is probably fine.

- The `pinMode()` function is used to set an I/O pin to either input mode (which measures voltages) or output mode (which generates voltages). In this example we set the pin to `OUTPUT` mode because we are using it to generate a voltage that will power the LED.

- The `digitalWrite()` function is used to set the voltage generated by a pin configured for `OUTPUT` mode by a previous call to `pinMode()`. Because it is a **digital** write there are two possible values: `HIGH` (5 V) and `LOW` (0 V). In this example we alternate between setting the pin to `HIGH` and `LOW` to make the LED turn on and off.

- The `delay()` function is used to pause the program for a specified amount of time (in milliseconds). In this program, calls to `delay()` control the frequency at which the LED blinks by forcing the Arduino to wait between turning the LED on and off.

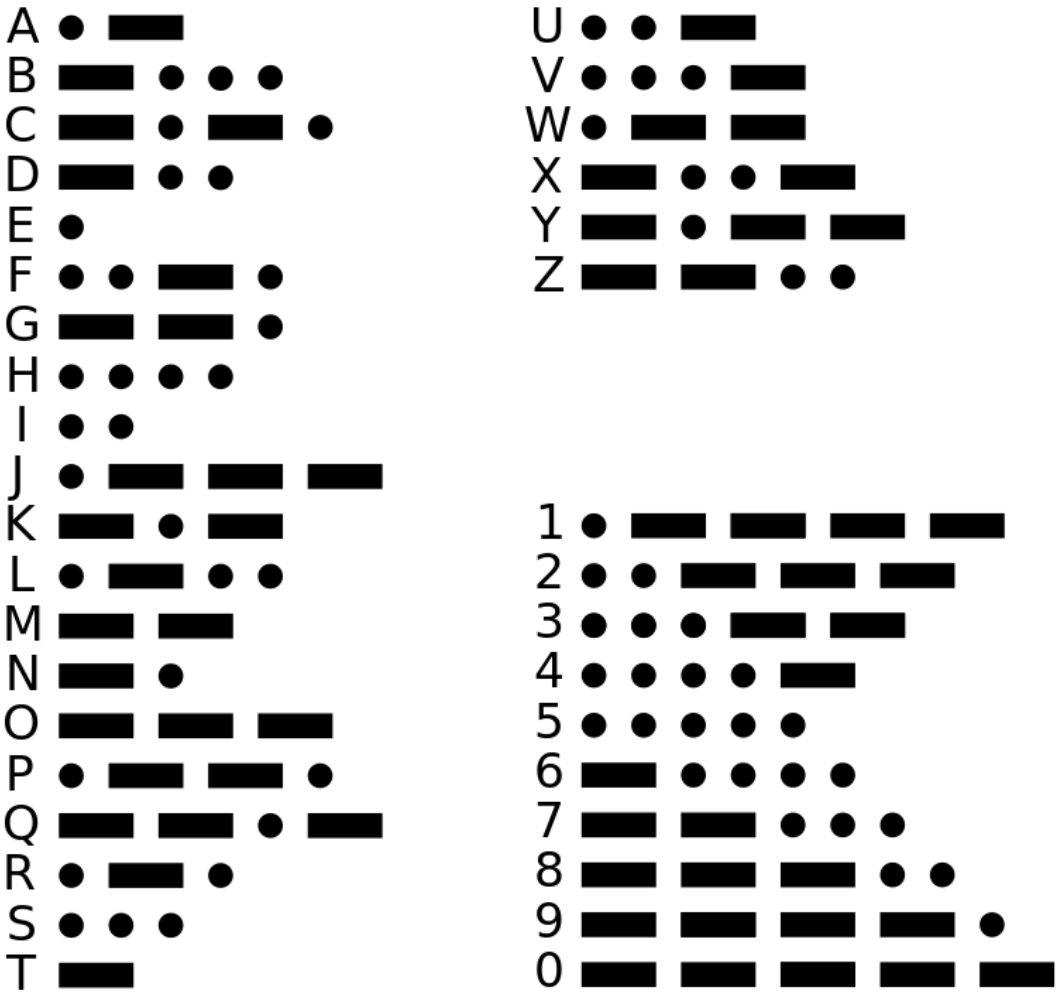
Assignment:

Using the circuit and program above as a starting point, configure your Arduino to blink out your name in Morse code on an LED. An introduction to Morse code is provided below, but you can also read an exhaustive explanation and history on the Wikipedia page for Morse code.

Your name probably has quite a few letters, and those letters probably have quite a few dots and dashes, and those dots and dashes probably require quite a few `digitalWrite()` and `delay()` calls. Check out the *deep dives* on the next pages for some hot C tips that can help make your programs less tedious to write (and better prepare you for the AP exam). At a minimum, consider creating a function for dots and a function for dashes, and use them to construct your Morse code message.

International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.



Deep Dive: `for` Loops

In this project's example program we made an LED blink. But what if we wanted to blink the LED three times and then pause before starting over again? We could change our `loop()` function to the following:

```
void loop()
{
      digitalWrite(2, HIGH);
      delay(100);
      digitalWrite(2, LOW);
      delay(100);

      digitalWrite(2, HIGH);
      delay(100);
      digitalWrite(2, LOW);
      delay(100);

      digitalWrite(2, HIGH);
      delay(100);
      digitalWrite(2, LOW);
      delay(100);

      delay(400);
}
```

This would do exactly what we wanted but is verbose and difficult to change. What if we wanted to blink ten times instead of three? In C we can create loops that run a block of code multiple times, just like in Scratch and Python. Let us rewrite our code to use a `for` loop:

```
void loop()
{
      for (int i = 0; i < 3; i++)
      {
            digitalWrite(2, HIGH);
            delay(100);
            digitalWrite(2, LOW);
            delay(100);
      }

      delay(400);
}
```

This code is more complex looking but makes it easier to change the number of blinks and their timing. The `for` loop in this example creates a variable called `i`, sets `i` to 0, and then blinks the LED 3 times because it increments `i` each time and stops when `i` reaches 3 (when `i < 3` is no longer true).

The Structure of `for` Loops

The structure of `for` loops in C is more complex than in other programming languages. Once you know what you are looking at, however, they are not so bad. A `for` loop has 4 parts:

```
for (A; B; C)
{
      D
}
```

(A) is the *initializer* of the loop and runs exactly once at the beginning of the loop. This is where you typically define and set an initial value for your *loop variable*.

(B) is the *condition* of the loop and is checked each time the loop runs. When it is no longer true the loop will stop. If it is *always* true your loop will run forever and you will be stuck in an *infinite loop*. Your test will almost always be a comparison involving your loop variable.

(C) is run after each time through the loop and is typically used to modify the loop variable to get closer to a stopping point. In our example we used `i++`, which is equivalent to `i = i + 1`, or the slightly more compact `i += 1`. Any of these would increase `i` by one.

(D) is the *body* of the loop and is code that will be run once each time the loop is run until the loop is finished.

Using `for` Loops

This is still kind of confusing! An easy way to get started is to use the following template:

```
for (int i = N; i < M; i++)
{
}
```

This will run the body of the loop M - N times, with `i` being set equal to each of the values N through M-1 in each of the times the body of the loop is run. All you need to do is specify the values of N, M, and the body.

```
for (int i = 2; i < 7; i++)
{
      Serial.println(i);
}
```

Running the above code will print 2, 3, 4, 5, and 6 to the serial monitor. Here is why:

(A) creates the loop variable `i` and sets it equal to 2. (B) checks whether `i` is less than 7 and, if it is (if that statement is `true`), then the code in the curly braces (D) is run once and, in this example, it prints the value of `i` to the serial monitor. After that, (C) adds one to the variable `i` and we loop back through (B), (D), and (C) until the condition in (B) is no longer `true`. When `i` equals 7, the loop stops because `i < 7` is now `false`.

Deep Dive: Functions

We can also create functions that *encapsulate* code into blocks that can easily be called from anywhere in a program, just like in Scratch and Python. The following code does the same thing as our `for` loop example above, but this time uses functions:

```
void loop()
{
      blinkOnce();
      blinkOnce();
      blinkOnce();
      delay(400);
}

void blinkOnce()
{
      digitalWrite(2, HIGH);
      delay(100);
      digitalWrite(2, LOW);
      delay(100);
}
```

Functions can also be written so that they have *parameters* that allow you to customize (or "parameterize") the behavior of the function when you call it. Consider the following revision of the example, which varies the length of the three blinks between 50 and 150 ms (milliseconds):

```
void loop()
{
      blinkOnce(50);
      blinkOnce(100);
      blinkOnce(150);
      delay(400);
}

void blinkOnce(int delayLength)
{
      digitalWrite(2, HIGH);
      delay(delayLength);
      digitalWrite(2, LOW);
      delay(delayLength);
}
```

Data Types for Functions

The keywords `void` and `int` are *data types*. When placed in front of a function declaration, a data type specifies what type of data the function returns. The functions `setup()`, `loop()`, and `blinkOnce()` all have the return type `void`, which means they do not return anything. If you wanted to create a function that returns an integer, you could put `int` in front of it, like this function that adds two numbers and returns the sum:

```
int add(int a, int b)
{
      return a + b;
}
```

Data types are also used with function parameters to specify what kind of data the function accepts. The parameters of `blinkOnce()` and `add()` are all integers, and so they are prefixed with the `int` data type.

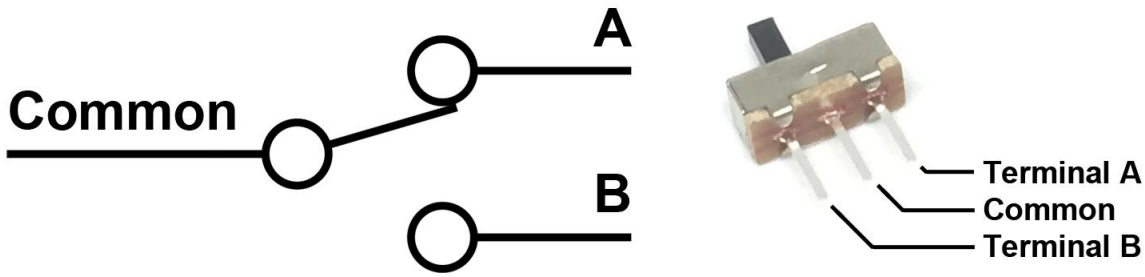
In addition to `int` and `void` there are lots of other data types supported by the C compiler used in the Arduino IDE. Here are some common ones, taken from the Arduino website:

`bool`	A boolean value that can be either `true` or `false`.
`byte`	An 8-bit unsigned integer, from 0 to 255.
`int`	A 16-bit signed integer, from -32,768 to 32,767.
`unsigned int`	A 16-bit unsigned integer, from 0 to 65,535.
`long`	A 32-bit signed integer, from -2,147,483,648 to 2,147,483,647.
`unsigned long`	A 32-bit unsigned integer, from 0 to 4,294,967,295.
`float`	A 32-bit floating point number (also known as a decimal number).
`double`	A 64-bit floating point number (also known as a decimal number).

Some programming languages, like C, C#, and Java, are *statically typed* and require you to explicitly state the data type of every function, parameter, and variable like we have done here. Other programming languages, like Python and Javascript, are *dynamically typed* and instead infer types based on how functions, parameters, and variables are used in your code (e.g., if you try to multiply two variables they presumably contain numbers). There are strengths and weaknesses of each, but for now you are writing C so you do not have much of a choice!

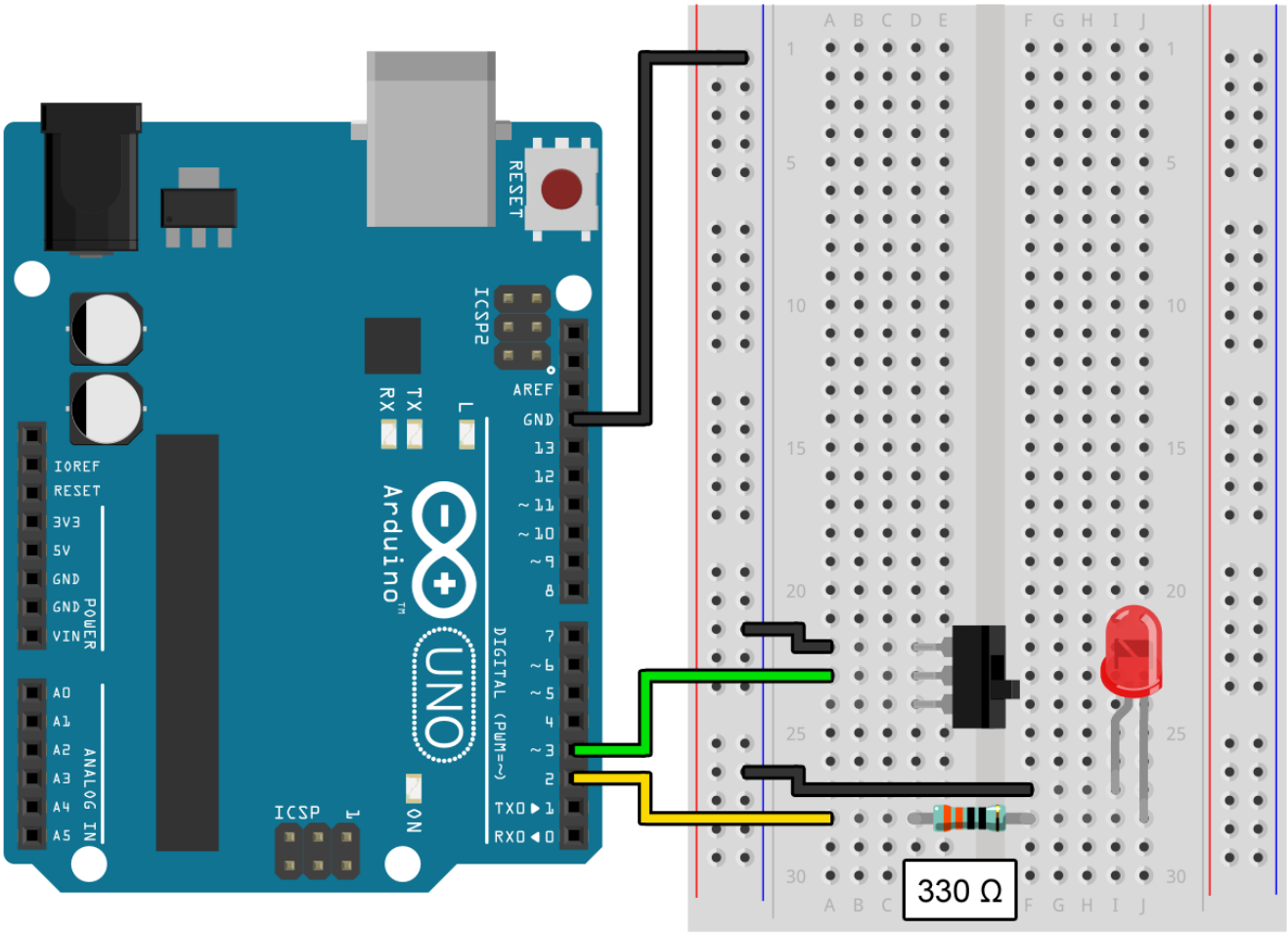
Project 2: Switches

A switch is an electrical component that changes the path of electricity through a circuit as it is moved between different physical positions. The particular switch in your kit is known as a “single-pole, dual-throw” (SPDT) switch because it switches a single circuit between two terminals.



Example Program:

Assemble the following circuit and upload the program to your Arduino:



```
void setup()
{
	pinMode(2, OUTPUT);
	pinMode(3, INPUT_PULLUP);
}

void loop()
{
	if (digitalRead(3) == LOW)
	{
		// The switch is on, so blink the LED on pin 2 quickly:
		digitalWrite(2, HIGH);
		delay(100);
		digitalWrite(2, LOW);
		delay(100);
	}
	else
	{
		// The switch is off, so blink the LED on pin 2 slowly:
		digitalWrite(2, HIGH);
		delay(200);
		digitalWrite(2, LOW);
		delay(200);
	}
}
```

Your LED should be blinking at a rate determined by the position of the switch, with one position causing it to blink twice as fast as the other position.

- Just like in the previous project we are using `pinMode()` to configure pin 2 as an `OUTPUT` that can turn an LED on and off. We are also doing something new, however, and hooking up a switch to pin 3 so that we can use it as an input. This is done by setting the pin mode to `INPUT_PULLUP`.

- Similar to how we used `digitalWrite()` to control an output pin and set it to `HIGH` or `LOW`, we can use `digitalRead()` with an input pin and it will return either `HIGH` or `LOW` depending on the voltage measured by the pin. By configuring a pin for `INPUT_PULLUP` it will be `HIGH` by default (due to something inside the microcontroller called a *pullup resistor*) and will change to `LOW` when the pin is connected to ground by the switch.

- By combining `digitalRead()` with `if` / `else` statements we can run different code depending on the state of an input. In the example above we blink quickly with a 100 ms delay when the switch is “on” (i.e. connected to ground) or blink slowly with a 200 ms delay when the switch is “off” (i.e. not connected to ground). Although you will not necessarily need it for this project, there is a deep dive for `if` / `else` statements in the next project with more information.

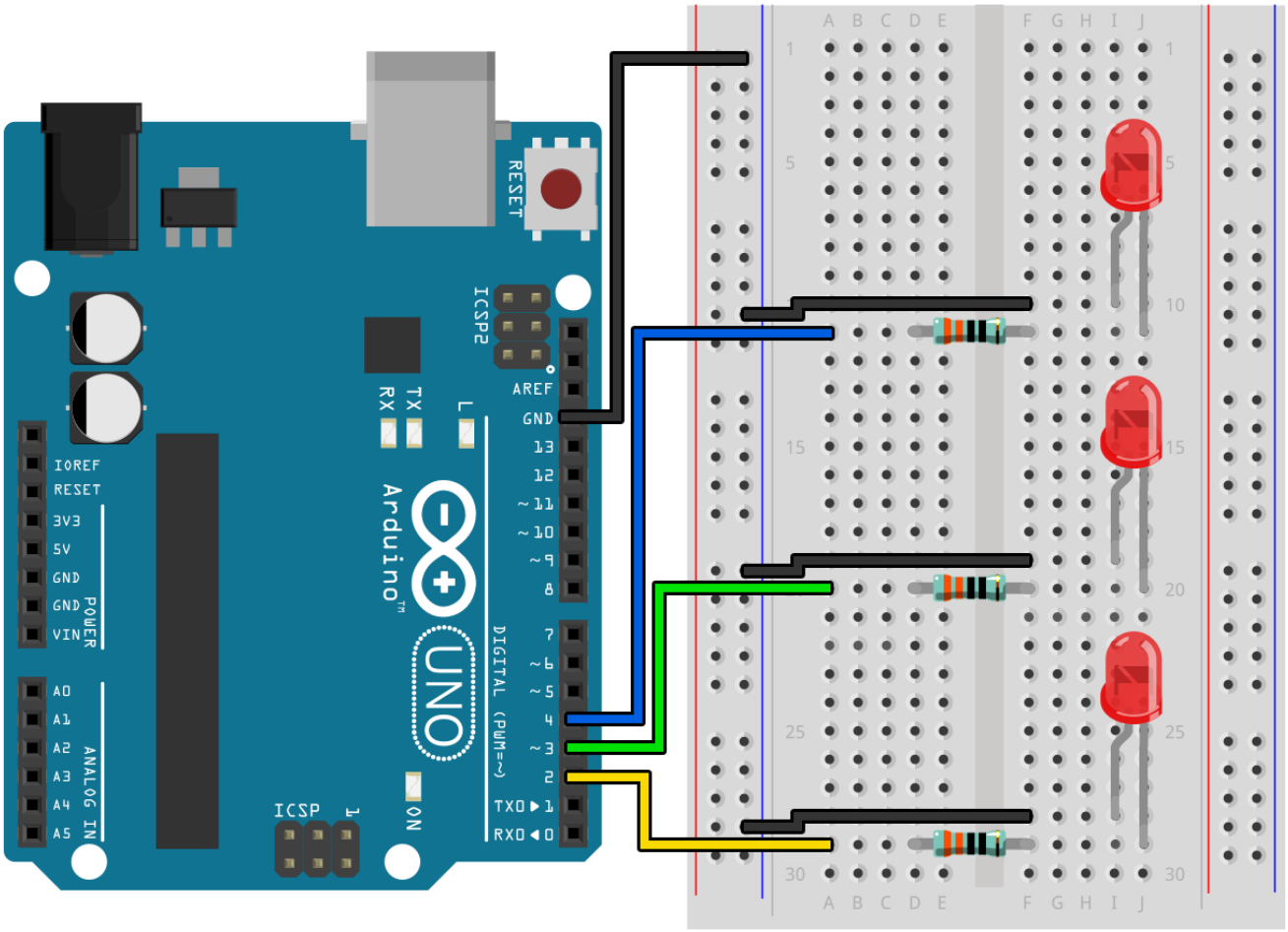
- This code is a little verbose and could be improved using variables. Look a few pages ahead for a deep dive into how variables work in C and how they could help.

Assignment:

Create an interesting animation using at least five LEDs that changes in a meaningful way when a switch is toggled. For example, you could have an animation that cycles between the five LEDs in sequence but goes in reverse when the first switch is toggled.

None of our examples so far have shown you how to hook up more than one LED or switch. The way to do this is simple: copy the wiring for a single component as many times as you need but use a different I/O pin on the Arduino so that you can “talk” to each of the components separately. Check out the deep dive on the next page for instructions on how to use choose which I/O pin to use.

In the example diagram below we have hooked up three LEDs to pins 2, 3, and 4. Each LED is hooked up in the same way as Project 1 (I/O pin → 330 Ω resistor → LED → GND) except that we have used three different I/O pins so that the LEDs can be turned on and off independently. We have also used the *ground rail* to connect the LEDs back to GND without having to use all three GND connections on the Arduino, two of which are on the other side of the board and are hard to reach.



Additional XP: Add a second switch to your circuit that changes the behavior of both of your animation modes. For example, if you made an animation that cycles the LEDs in forward or reverse, your second switch could toggle between normal speed and double speed for either direction.

Deep Dive: I/O Pins

In the example above we hooked up an LED to pin 2 and a switch to pin 3. But why did we use these pins? Could we have used different pins? How do we hook up more components?

There are many I/O pins that can be used on your Arduino, but, confusingly, they are not all interchangeable. **When you are connecting components to your Arduino you must select pins that support the functions you want to use with them.**

When we connected an LED to our Arduino we set the pin to `OUTPUT` mode and used `digitalWrite()` to turn it on and off. We used pin 2, but if we look at the table below we can see that `digitalWrite()` is supported by pins 2 through 12 and A0 through A5.

Similarly, when we connected the switch we set the pin to `INPUT_PULLUP` mode and used `digitalRead()` to read its state. We used pin 3, but if we look at the table below we can see that `digitalRead()` is supported by the same pins as `digitalWrite()`.

If we wanted to hook up five LEDs and two switches we could connect the LEDs to pins 2 through 6 and the switches to pins 7 and 8. Or we could connect the switches to pins 2 and 3 and the LEDs to pins 4 through 8. Or we could connect the LEDs to pins 2, 4, 6, 8, and 10 and the switches to pins A0 and A5. Any of these configurations are valid, as long as you make sure you use the corresponding pin name in your program when calling `pinMode()`, `digitalRead()`, and `digitalWrite()`.

In later projects we will use the `analogRead()` and `analogWrite()` functions, which are much more restrictive about which pins they can be used with. Consult the table below to ensure you are using the right pins for the job!

If you do not see a pin listed here (like 3.3V, VIN, IOREF, and AREF) you should not use it.

Pin	digitalRead / digitalWrite	analogRead	analogWrite	Description
5V				Provides power for components (5 volts)
GND				Provides ground for components
A0	X	X		
A1	X	X		
A2	X	X		
A3	X	X		
A4	X	X		Also used for “SDA” connection on LCD
A5	X	X		Also used for “SCL” connection on LCD
0				Used by the USB serial connection
1				Used by the USB serial connection
2	X			
3	X		X	
4	X			
5	X		X	
6	X		X	
7	X			
8	X			
9	X		X	
10	X		X	
11	X		X	
12	X			
13				Used by an LED soldered to the Arduino

Deep Dive: Variables

Similar to Scratch and Python, C allows you to create variables to store temporary values. In the example for Project 2 we used this code to blink the LED at a rate dependent on the state of a switch:

```
void loop()
{
	if (digitalRead(3) == LOW)
	{
		// The switch is on, so blink the LED on pin 2 quickly:
		digitalWrite(2, HIGH);
		delay(100);
		digitalWrite(2, LOW);
		delay(100);
	}
	else
	{
		// The switch is off, so blink the LED on pin 2 slowly:
		digitalWrite(2, HIGH);
		delay(200);
		digitalWrite(2, LOW);
		delay(200);
	}
}
```

We can rewrite this code to read the switch state, store the corresponding delay time in a variable (`delayTime`), and then blink the light using that delay time instead of hardcoding the `delay()` calls:

```
void loop()
{
	// Determine the delay time by reading the switch on pin 3:
	int delayTime;
	if (digitalRead(3) == LOW)
	{
		delayTime = 100;
	}
	else
	{
		delayTime = 200;
	}

	// Blink the LED on pin 2 using the delay we determined above:
	digitalWrite(2, HIGH);
	delay(delayTime);
	digitalWrite(2, LOW);
	delay(delayTime);
}
```

Much like function parameters, variables in C must be declared before they can be used and must specify a data type. The variable used above, `delayTime`, is an `int` that can store an integer between -32,768 and 32,767, but there are many other data types available. Here are some common ones, taken from the Arduino website:

`bool`	A boolean value that can be either `true` or `false`.
`byte`	An 8-bit unsigned integer, from 0 to 255.
`int`	A 16-bit signed integer, from -32,768 to 32,767.
`unsigned int`	A 16-bit unsigned integer, from 0 to 65,535.
`long`	A 32-bit signed integer, from -2,147,483,648 to 2,147,483,647.
`unsigned long`	A 32-bit unsigned integer, from 0 to 4,294,967,295.
`float`	A 32-bit floating point number (also known as a decimal number).
`double`	A 64-bit floating point number (also known as a decimal number).

Variable Scope

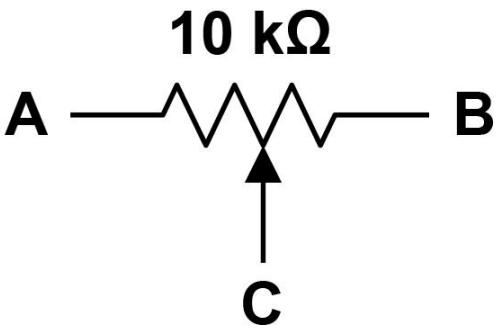
In C, matters are further complicated by a concept called *variable scope*. If a variable is declared within a set of curly braces it can only be used within those curly braces, and then only after it is declared. Consider the following example, in which a variable is declared within a function and can be accessed from some places but not from others:

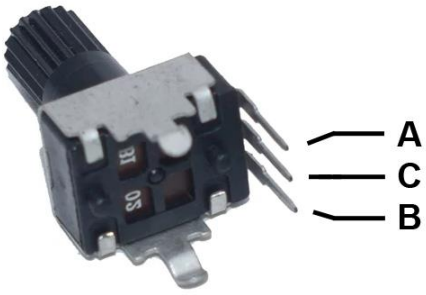
```
void loop()
{
	// We *CANNOT* use delayTime here; it has not yet been declared.

	int delayTime = 0;

	// We *CAN* use delayTime here.

	if (digitalRead(7) == LOW)
	{
		// We *CAN* also use delayTime here because we are still inside
		// the outer set of curly braces that it was declared within.
	}
}

// We *CANNOT* use delayTime here because it is no longer in scope.
```

If you want to be able to use a variable from anywhere in your program you should put it at the top of your program, before `setup()` and `loop()`. This is sometimes called a *global variable*.

Project 3: Potentiometers

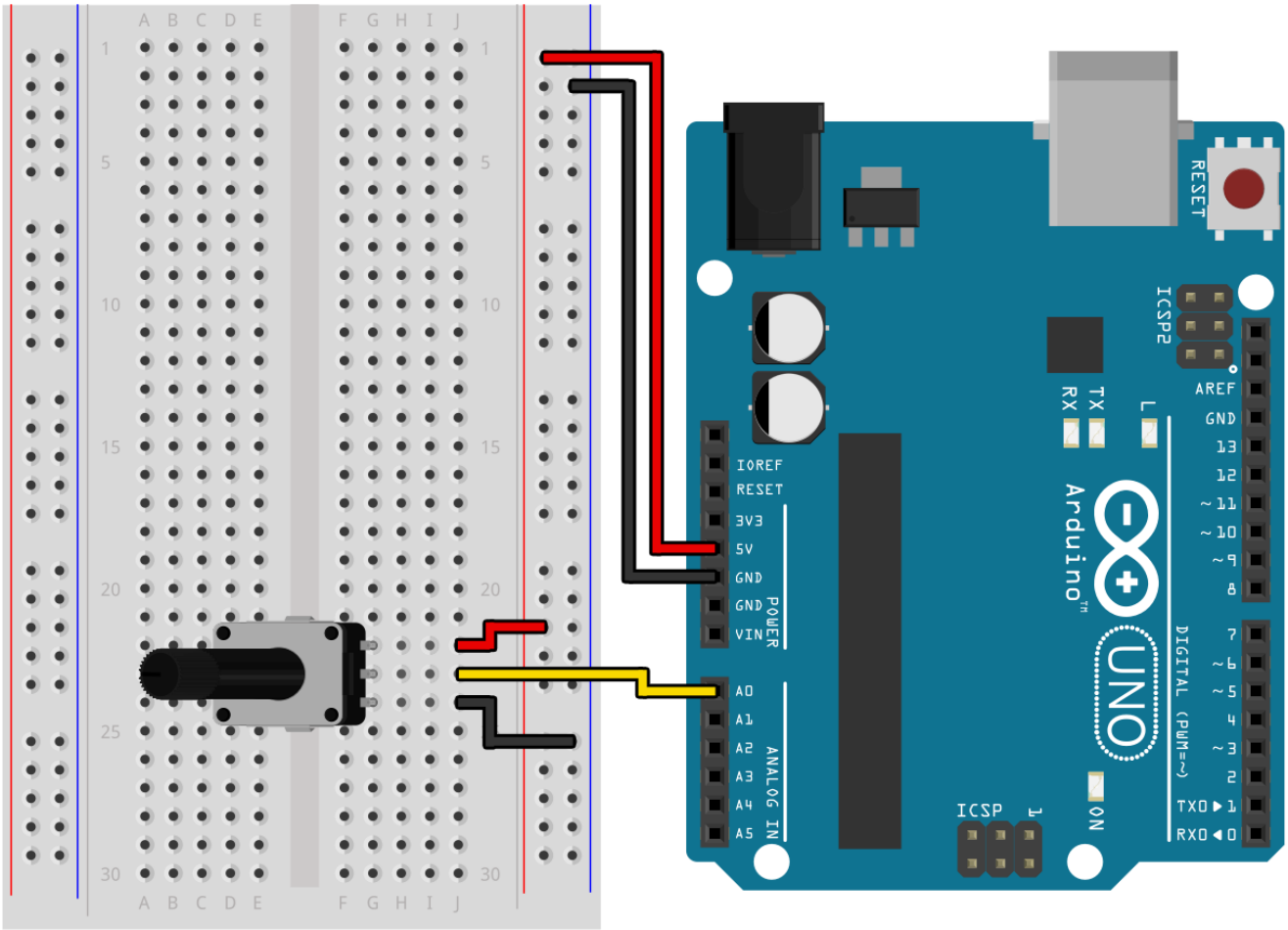
A *potentiometer* is a type of variable resistor. For the potentiometers in your kit the resistance between the two side pins (A and B in the diagram below) is always 10 kΩ (10,000 Ω). The center pin (C in the diagram below) is electrically connected to a point along the resistor that varies as the knob is turned one way or the other, which in turn varies the resistance between A and C (and B and C).



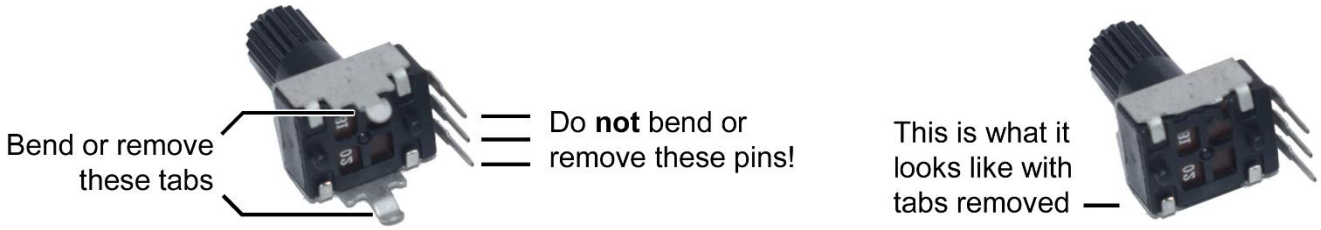


Example Program:

Assemble the following circuit and upload the program to your Arduino:



Before you can connect the potentiometer to your breadboard you will need to bend or remove the two metal tabs on the bottom of the potentiometer so that they do not physically block you from inserting the pins into the breadboard. Although they are not useful for our purposes, these tabs exist so that the potentiometer could be firmly attached to a circuit board.




```
void setup()
{
      Serial.begin(9600);
      pinMode(A0, INPUT);
}

void loop()
{
      Serial.println(analogRead(A0));
}
```

If you open the serial monitor in the Arduino IDE you should see a stream of numbers being printed to it, with values between 0 and 1023 that change as you twist the knob from one end to the other.

- When the two side pins of a potentiometer are connected to 5 V and 0 V, this creates a circuit called a *voltage divider* that causes the voltage on the center pin to vary between 5 V and 0 V as the knob is turned from one end to the other. It is not important to understand why this works (although it has an interesting answer), but know that when a potentiometer is hooked up in this way the voltage on the center pin will follow the position of the knob.
- In the previous project we used `digitalRead()` to measure a voltage and convert it to either `HIGH` or `LOW` depending on if it was closer to 5 V or 0 V. This time we are using `analogRead()` to do the measuring, which has the ability to return a continuous value mapping 0 V to 0 and 5 V to 1023. Although this is not truly “analog” it is much closer, with 10 bits of precision (1024 possible values from 0 V to 5 V) instead of just 1 bit (2 possible values, either 0 V or 5 V).
- Before you use a pin as an *analog input* by calling `analogRead()` you must make sure the pin is configured as an `INPUT` (and **not** `INPUT_PULLUP`) in your `setup()` function. Using `INPUT_PULLUP` will cause `analogRead()` to return inaccurate values.
- Not all pins can be used with `analogRead()`! If you look back at the table of pin capabilities following Project 2 you will see that only pins A0 through A5 support the `analogRead()` function.

Assignment:

Create an “LED bar graph” using at least 4 LEDs (you choose the number and colors) that “fills up” as you turn the potentiometer clockwise. When the potentiometer is turned all the way counter-clockwise all the LEDs should be off, and then as you turn it clockwise they should turn on one by one until the potentiometer is all the way clockwise and all of the LEDs have turned on.

In the previous project we used an `if` / `else` statement to do one of two different things depending on the state of a switch.

```
if (digitalRead(3) == LOW)
{
      // The switch is on.
}
else
{
      // The switch is off.
}
```

We can do something similar with the return value of an `analogRead()` call by using comparison operators like < (less than) and > (greater than). Keeping in mind that the range of values returned by `analogRead()` is somewhere between 0 and 1023:

```
if (analogRead(A0) < 512)
{
      // The potentiometer is turned one way.
}
else
{
      // The potentiometer is turned the other way.
}
```

You can do more complicated things with `if` and `else` than merely choose between two options. See the deep dive that follows for examples of more complex usage.

Deep Dive: `if` / `else` Statements

The following examples assume that you have two switches hooked up to pins 2 and 3 of the Arduino and configured for `INPUT_PULLUP`. None of what follows is specific to switches or `digitalRead()`; you could just as easily have used potentiometers and `analogRead()` like in the example above.

```
// Example #1 - The classic "if/else":
void loop()
{
      if (digitalRead(2) == LOW)
      {
            // The switch on pin 2 is on.
      }
      else
      {
            // The switch on pin 2 is off.
      }
}

// Example #2 - A lone "if" statement:
void loop()
{
      if (digitalRead(3) == LOW)
      {
            // The switch on pin 3 is on.
      }
}

// Example #3 - Adding an "else if" into the mix:
void loop()
{
      if (digitalRead(2) == LOW)
      {
            // The switch on pin 2 is on.
      }
      else if (digitalRead(3) == LOW)
      {
            // The switch on pin 2 is not on, but the switch on pin 3 is.
      }
      else
      {
            // The switches are both off.
      }
}
```

Note that you can have any number of `else if` statements after an `if`. The `else` on the end is optional, but if you use it, it must be at the end.

Logical Operators

Similar to Scratch and Python, C allows you to test multiple conditions in a single `if` statement by combining them with *logical operators*:

```
void loop()
{
      if (digitalRead(2) == LOW && digitalRead(3) == LOW)
      {
            // Both switches are on.
      }

      if (digitalRead(2) == LOW || digitalRead(3) == LOW)
      {
            // At least one (or both) of the switches are on.
      }
}
```

The *and operator* (`&&`) combines two tests together and requires both of them to be true. This is equivalent to the `and` keyword in Python.

The *or operator* (`||`) combines two tests together and requires at least one of them to be true. This is equivalent to the `or` keyword in Python.

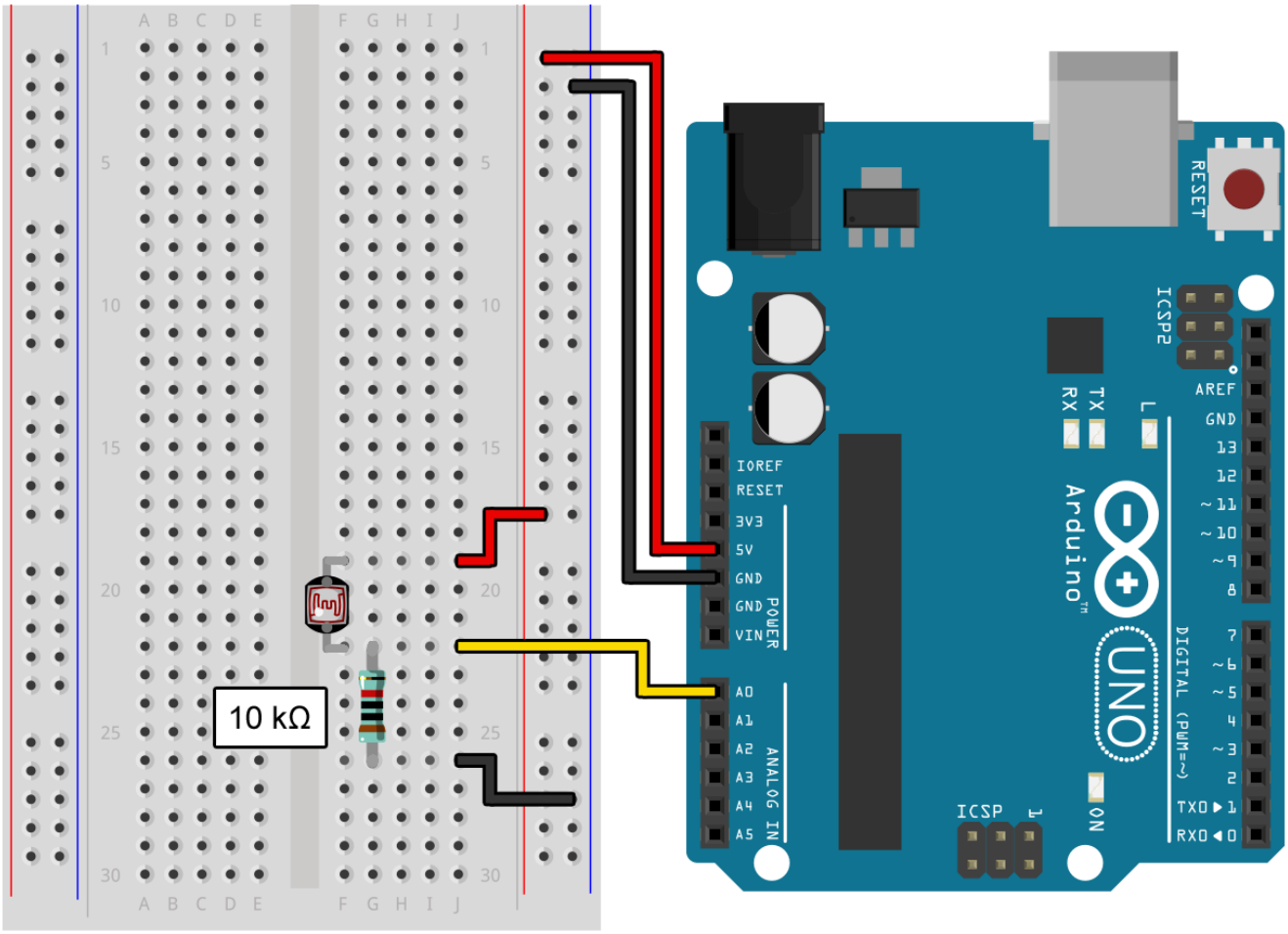
If you intend to use `&&` or `||`, make sure you do not accidentally forget a character and use `&` or `|` by themselves. C is confusing and considers both of those to be valid, although they will do something completely different than what `&&` and `||` do.

Project 4: Photoresistors

A *photoresistor*, or *photocell*, is a resistor that changes resistance based on light. As more light shines on a photoresistor its resistance decreases. The photoresistor in your kit has a resistance of around 100 kΩ in the dark and 1 kΩ in the light. By combining it with another resistor you can create a light sensor, which you will do in this project.

Example Program:

Assemble the following circuit and upload the program to your Arduino:



Make sure that you use a **10 kΩ resistor** as indicated, not a 330 Ω resistor like in previous projects.

```
void setup()
{
      Serial.begin(9600);
      pinMode(A0, INPUT);
}

void loop()
{
      Serial.println(analogRead(A0));
}
```

If you open the serial monitor in the Arduino IDE you should see a stream of numbers being printed to it, with values between 0 and 1023 that change as you cover and uncover the photoresistor and change the amount of light shining on it.

- Although this circuit looks nothing like the potentiometer we used in the previous project, it is actually doing something very similar. The combination of the resistor and the photoresistor creates another voltage divider that causes the voltage measured by an analog input pin to change as the resistance of the photoresistor changes.

Assignment:

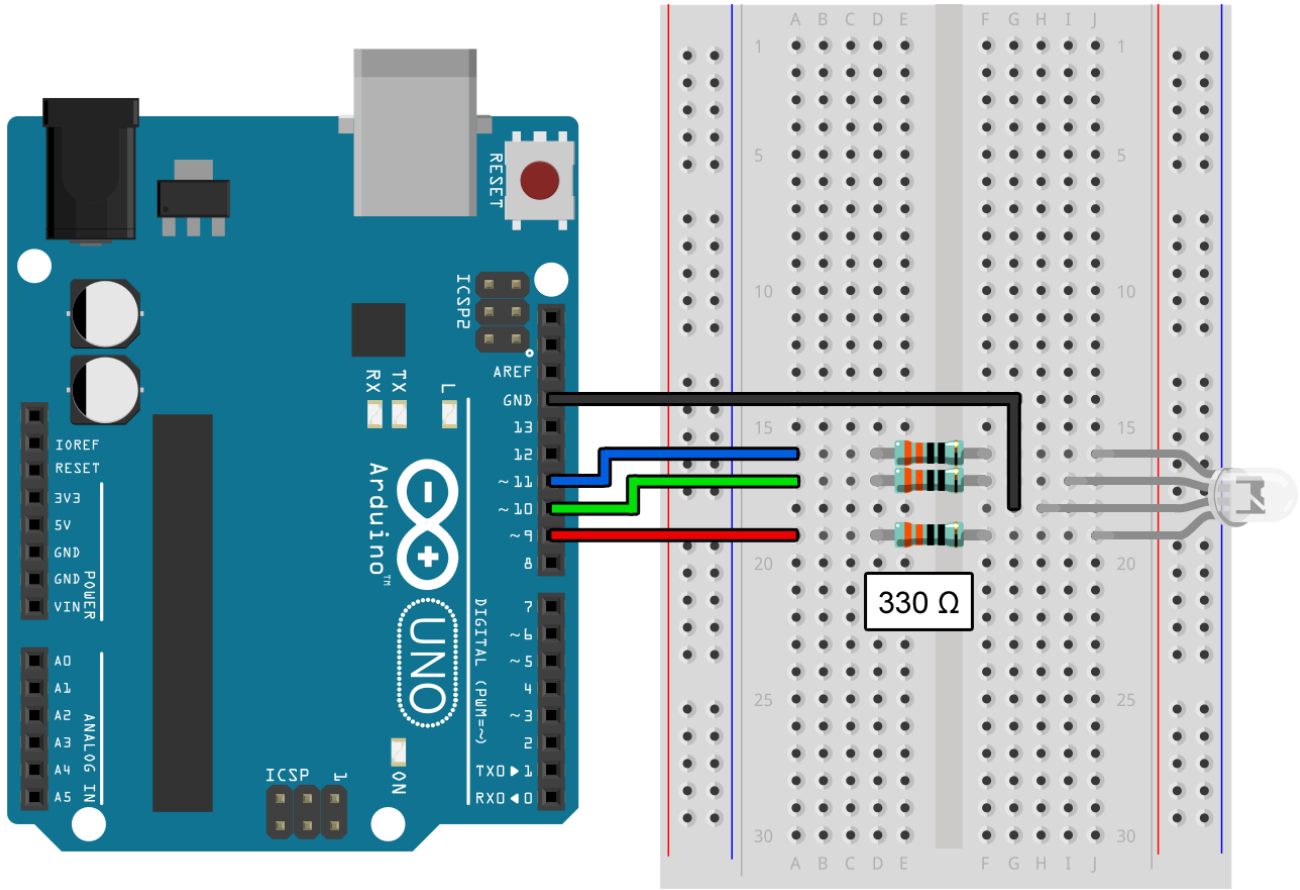
Create a circuit that detects when someone puts their hand in front of the photoresistor and blinks a pattern (of your choosing) on an LED, sort of like a hands-free paper towel dispenser. You will need to experimentally “calibrate” your photoresistor and determine the threshold at which it should turn on. Printing the values returned by `analogRead()` to the serial monitor like in the example above is an easy way to do this.

Project 5: RGB LEDs

An RGB LED looks a lot like a normal LED, except that it has four pins instead of two. They contain three independent LEDs (red, green, and blue) that can be turned on separately or together to create many different colors, like a pixel on a screen. If you turned on the red and blue LEDs at the same time the light would be magenta. If you turned on all the LEDs at the same time the light would be white. Each LED in the RGB LED is wired separately and needs its own current-limiting resistor to not “burn out.”

Example Program:

Assemble the following circuit and upload the program to your Arduino:



```
void setup()
{
      pinMode(9, OUTPUT);     // Red
      pinMode(10, OUTPUT);    // Green
      pinMode(11, OUTPUT);    // Blue
}

void loop()
{
      // Increase the brightness of the blue LED:
      for (int i = 0; i < 256; i++)
      {
            analogWrite(11, i);
            delay(4);
      }

      // Turn off the blue LED:
      analogWrite(11, 0);

      // Increase the brightness of the red LED:
      for (int i = 0; i < 256; i++)
      {
            analogWrite(9, i);
            delay(4);
      }

      // Turn off the red LED:
      analogWrite(9, 0);
}
```

Your RGB LED should alternate between red and blue, each time ramping up linearly from fully off to fully on over about a second. If you are seeing this but with the wrong colors it means you have hooked up the pins in the wrong order; check your wiring and try again.

- In previous projects we used `digitalWrite()` to turn LEDs on and off. In this project we are using `analogWrite()` to power LEDs at less than full brightness. Instead of writing `HIGH` or `LOW`, we write an integer value between 0 (fully off) and 255 (fully on). In the example above we use it to ramp up the brightness of red and blue separately, but you could (and will) also use it to blend between colors, like turning on 100% blue and 50% red to create a blueish-magenta color.
- Much like `analogRead()`, the `analogWrite()` function is very particular about which pins it can be used with. If you consult the table in Project 2 you will see that it can only be used on pins 3, 5, 6, 9, 10, and 11. This is also indicated on the Arduino itself by a '~' symbol next to the pin number.

- Although this project has you using `analogWrite()` with RGB LEDs, you can also use `analogWrite()` with single-color LEDs. In real-world electronics this is a common way to vary the power of analog devices like lights and motors when using a microcontroller.
- If `analogRead()` *measures* a continuous analog value between 0 V and 5 V, does `analogWrite()` *generate* a continuous analog value between 0 V and 5 V too? Not quite! For a proper explanation check out the deep dive into *pulse-width modulation* (or “PWM”) on the next page.

Assignment:

Create a circuit that blends the output of an RGB LED between two different colors as a potentiometer is turned from one end to the other. You can choose whatever colors you want, but they need to be distinct enough that the color blending is clearly visible.

Although it is possible to do the math yourself, this is a great opportunity to use the `map()` function to convert a value from one range to another. As you learned in Project 3, the `analogRead()` function returns a value between 0 and 1023. If you wanted to remap this to a smaller range, say from 0 to 255, you could use the following code instead, which returns the remapped value and could be stored in a variable or passed to another function:

```
map(analogRead(), 0, 1023, 0, 255)
```

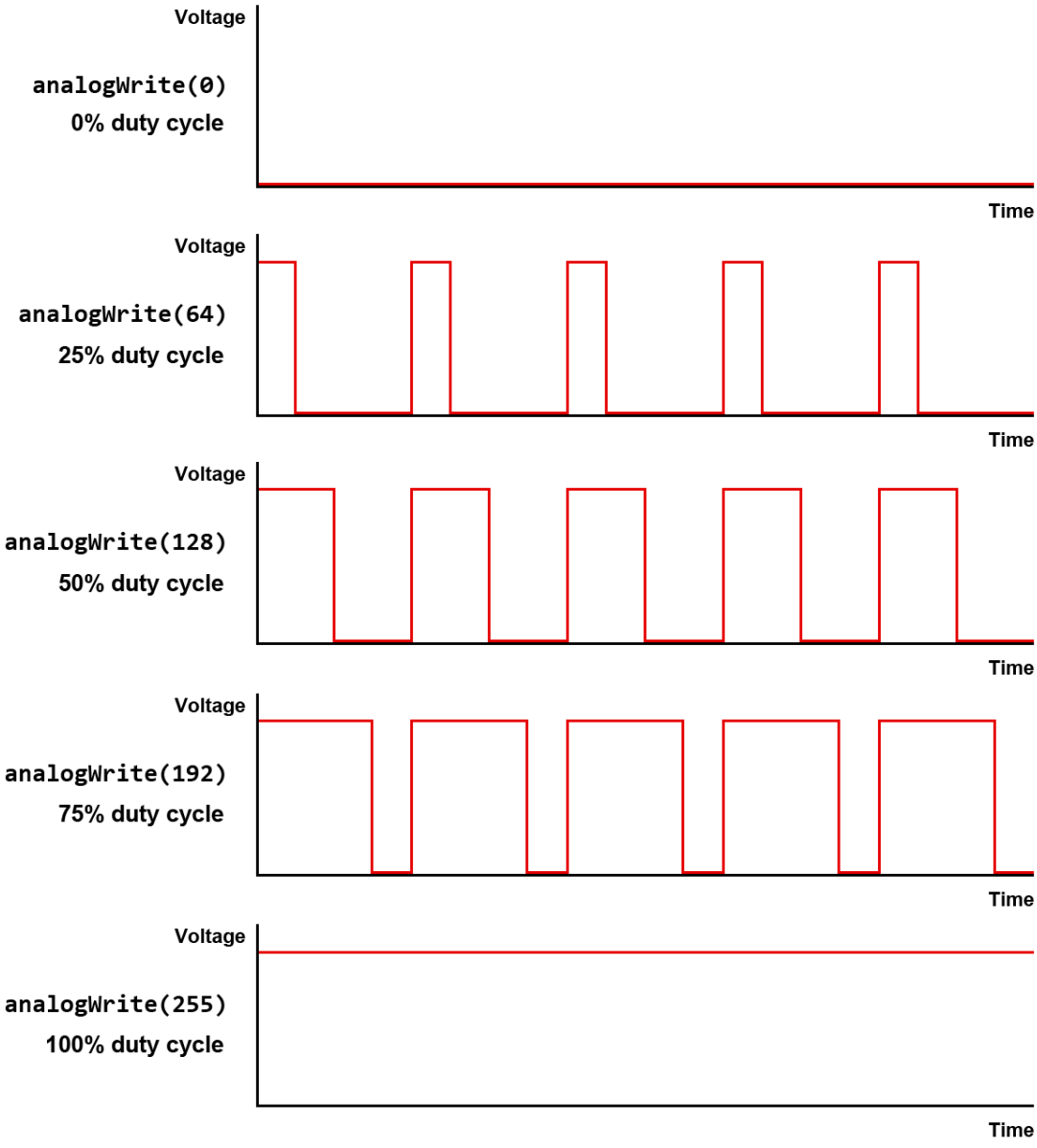
A full explanation of the `map()` function is available here, but a quick list of the parameters follows:

```
map(value, fromLow, fromHigh, toLow, toHigh)
```

Additional XP: Hook up your second potentiometer and use it to vary the brightness of the RGB LED between 0% and 100%, and have it work correctly for the full range of input from the potentiometer that sets the color. You will need to add additional calls to `map()` to make this work.

Deep Dive: PWM

Although the name of `analogWrite()` may lead you to believe it is generating an analog value (between 0 V and 5 V) on an output pin, the microcontroller in your Arduino (the ATmega328P) does not have the ability to do this. What it is really doing is something called *pulse-width modulation* (or "PWM") which involves quickly turning the pin on and off while varying the amount of time spent on or off (the *duty cycle*) depending on the "analog value" written.



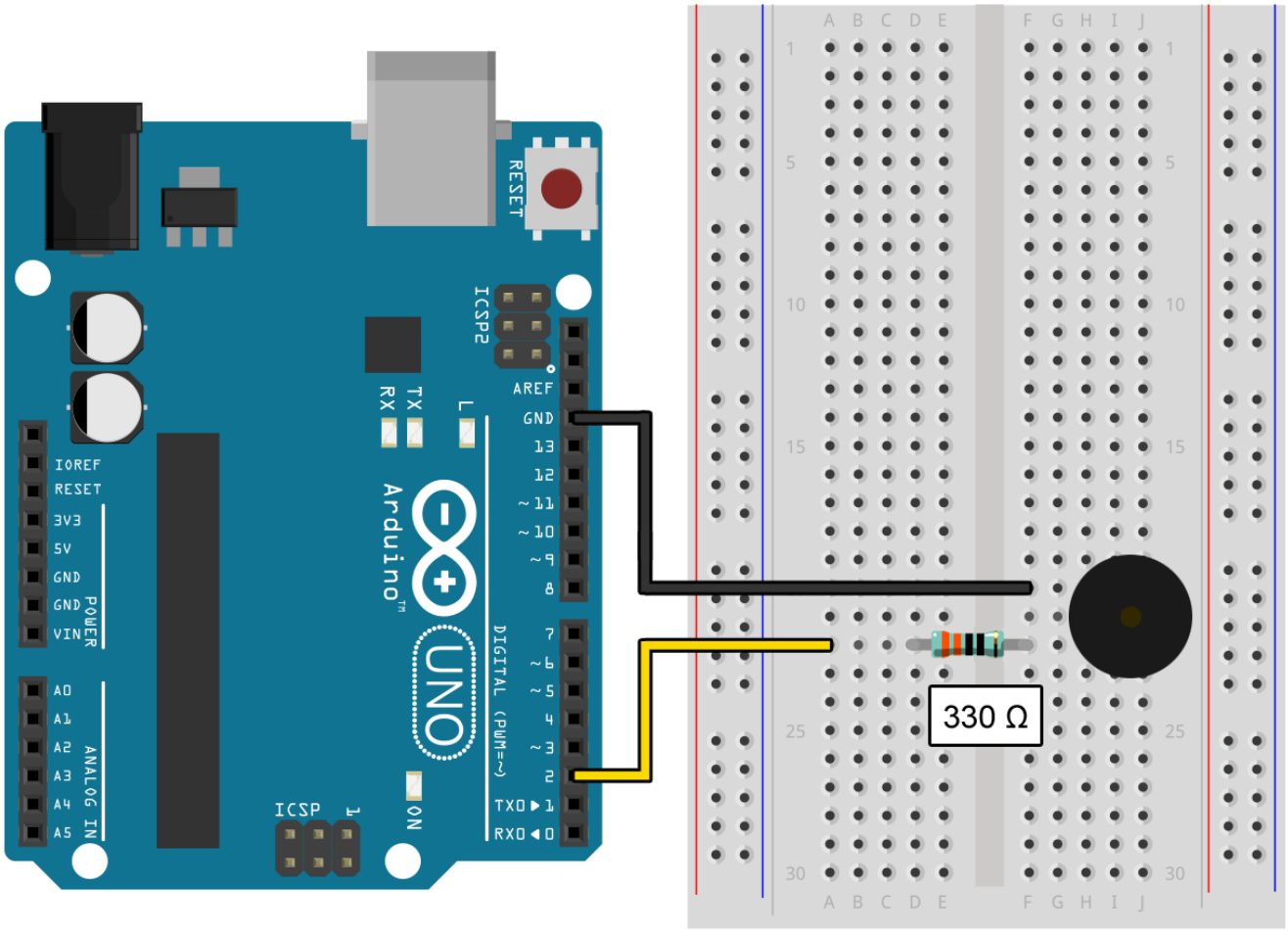
This is very different from a true *analog output* that can generate any voltage between 0 V and 5 V, and PWM can not be used in every situation that calls for a proper analog signal. For something like an LED, however, this is perfectly fine; although the LED is technically flashing on and off, it is going so fast that persistence of vision prevents you from seeing it. Instead, you just see the light get brighter or dimmer as the LED spends more or less time turned on.

Project 6: Piezo Buzzers

Like all speakers, piezo buzzers convert an electrical signal into physical vibrations that vibrate the air to create sounds that we can hear. Most speakers (like the kind in a phone, stereo, or car) use electromagnets, which use a lot of power and are difficult to control. Piezo buzzers use small crystals that vibrate when voltage is applied and are so low power that they can be connected directly to an I/O pin on an Arduino.

Example Program:

Assemble the following circuit and upload the program to your Arduino:



```
void setup()
{
      pinMode(2, OUTPUT);
}

void loop()
{
      // Play a 554 Hz (C#) tone for 250 ms:
      tone(2, 554);
      delay(250);

      // Play a 440 Hz (A) tone for 250 ms:
      tone(2, 440);
      delay(250);

      // Silence the buzzer for 500 ms:
      noTone(2);
      delay(500);
}
```

You should hear the piezo buzzer repeatedly play two tones (C# and A) with a delay between repetitions. If you put a small piece of clear tape over the top of the buzzer it will greatly improve the sound quality and make it easier to hear. If the piezo buzzer is still too quiet, or for some reason you just want it to be really loud, you can remove the resistor and directly connect the piezo buzzer between pin 2 and ground. Just do not ever do this with an LED!

- When the `tone()` function is called the Arduino will begin turning the specified pin on and off at the specified frequency. In our example above, `tone(2, 440)` tells the Arduino to turn pin 2 on and off at 440 hertz. The `tone()` function is similar to `digitalWrite()` in that it expects the pin to be configured as an `OUTPUT` pin, which we do using `pinMode()` in `setup()` above.
- To change the tone being played on a pin you can simply call `tone()` again with the same pin and a different frequency. There is no need to stop it first. To silence the tone being played on a pin you should call the `noTone()` function as shown in the example above.
- The `tone()` function can only be used to play a single note at a time. If you start playing a note it will stop any previously playing notes, even if they are on a different pin. By using the external Tone library, however, you can play up to three notes at once using three separate piezo buzzers. Perhaps an ambitious goal for a capstone project?

Assignment:

In this assignment you will create a program that plays a song on a piezo buzzer. If you were to write this out naively you would have far too many `tone()` and `noTone()` calls, so you will implement the following function to play a specific note for a specific duration of time:

```
void playNote(int frequency, int beats)
{
      // Play a note with the specified frequency and number of beats.
      // If the frequency is 0, rest for the specified number of beats instead.
}
```

If you have implemented that correctly, you will then be able to make the following function calls in your `loop()` function to play a song that will hopefully be familiar to you. For the song to sound correct, the length of a beat should be about 150 ms.

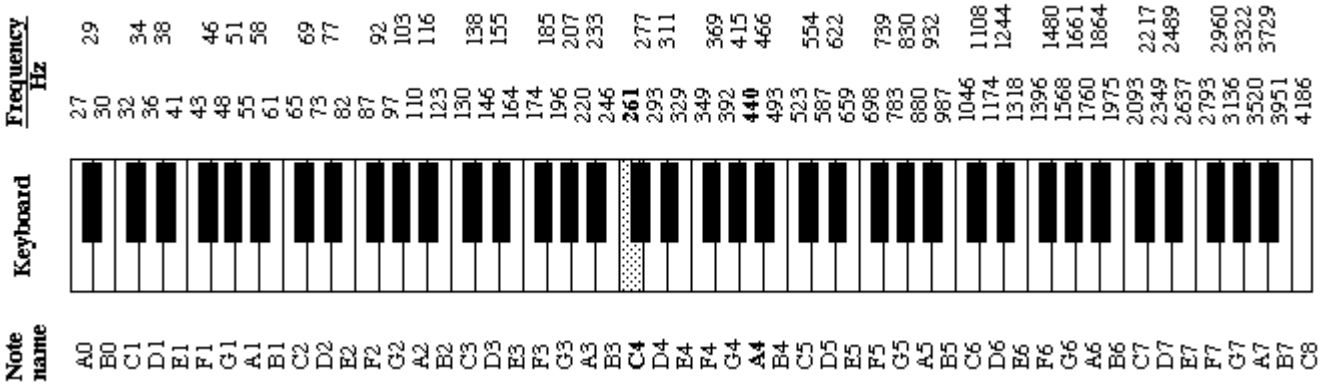
```
void loop()
{
      playNote(415, 1); // A♭
      playNote(466, 1); // B♭
      playNote(554, 1); // D♭
      playNote(466, 1); // B♭
      playNote(698, 3); // F
      playNote(698, 3); // F
      playNote(622, 4); // E♭
      playNote(0, 2);   // Rest
      playNote(415, 1); // A♭
      playNote(466, 1); // B♭
      playNote(523, 1); // C
      playNote(466, 1); // B♭
      playNote(622, 3); // E♭
      playNote(622, 3); // E♭
      playNote(554, 4); // D♭
      playNote(0, 2);   // Rest
      playNote(415, 1); // A♭
      playNote(466, 1); // B♭
      playNote(554, 1); // D♭
      playNote(466, 1); // B♭
      playNote(554, 3); // D♭
      playNote(0, 1);   // Rest
      playNote(622, 2); // E♭
      playNote(523, 4); // C
      playNote(415, 3); // A♭
      playNote(0, 1);   // Rest
```

```
      playNote(415, 2); // A♭
      playNote(622, 4); // E♭
      playNote(554, 4); // D♭
      playNote(0, 4);   // Rest
}
```

So that you can verify your program is correct, here is a video of the song being performed.

To make your music sound more natural (if that is even possible with a piezo buzzer) try adding a very small delay at the end of your `playNote()` function so that sequential notes do not run together.

Additional XP: Instead of using the song above, write a program that plays a song of your choice. You will need to convert the musical notes into frequencies and beats. The following diagram is a little intense but lists the frequency of every note on a piano keyboard:



http://newt.phys.unsw.edu.au/jw/notes.html

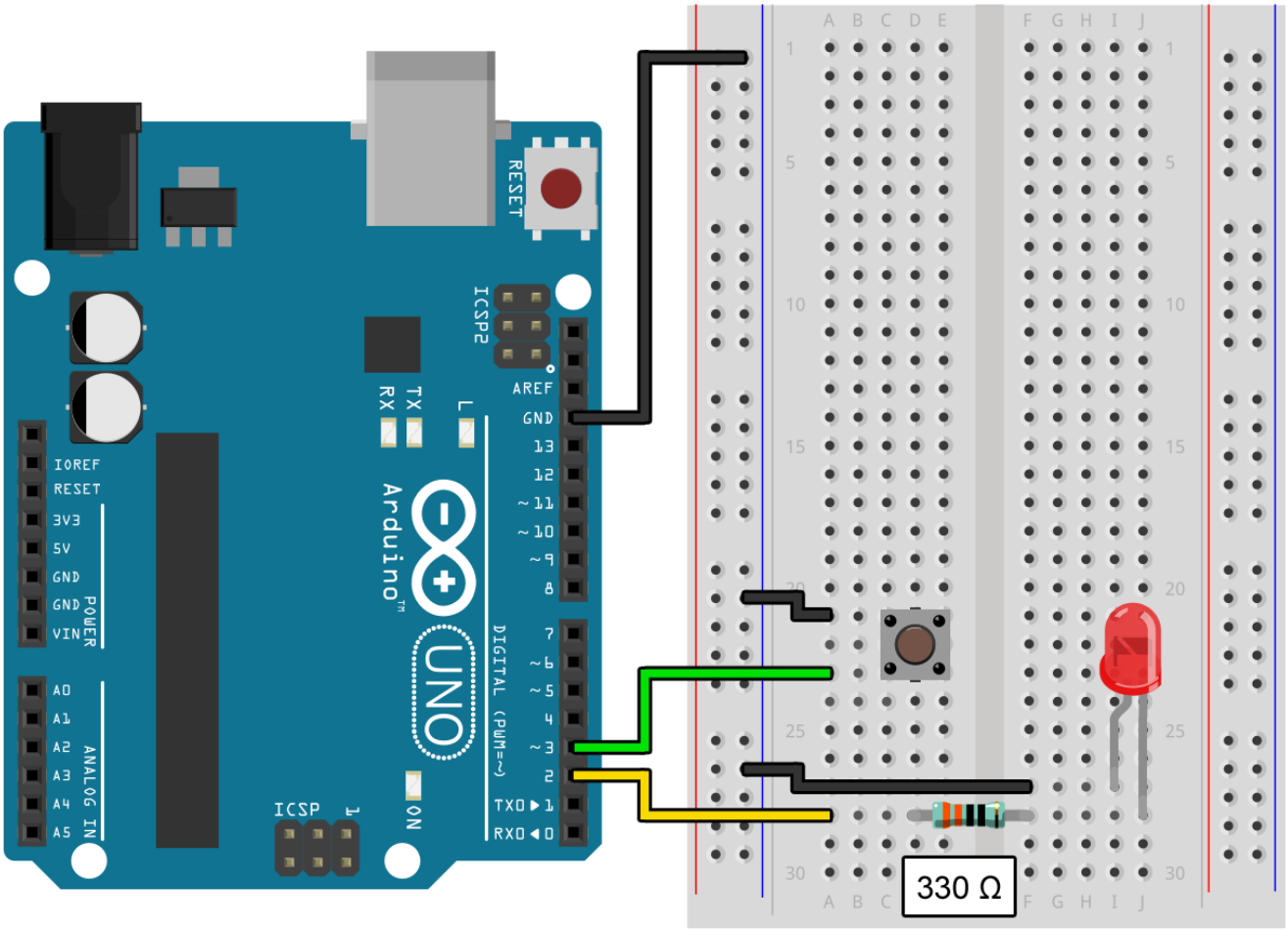
Doubling a note's frequency moves it up exactly one octave, while halving the frequency moves it down an octave. This is one of many mathematical relationships found in music theory.

Project 7: Buttons

Buttons, or *momentary switches*, have two pins that become electrically connected while you are holding down the button. They can be connected to your Arduino in the same way as a switch, allowing you to use them as an input to your program. The “click” you feel when you press them is called *haptic feedback*.

Example Program:

Assemble the following circuit and upload the program to your Arduino:



```
void setup()
{
      pinMode(2, OUTPUT);
      pinMode(3, INPUT_PULLUP);
}

void loop()
{
      if (digitalRead(3) == LOW)
      {
            // Turn on the LED when the button is pressed:
            digitalWrite(2, HIGH);
      }
      else
      {
            // Turn off the LED when the button is not pressed:
            digitalWrite(2, LOW);
      }
}
```

You should see the LED light up while you press the button and turn off when you release it.

- From an electrical standpoint a button is largely the same as a switch (Project 2). Pin 3 is configured as an `INPUT_PULLUP`, which means the internal pullup resistor is pulling the input `HIGH` by default. When the button is pressed it becomes connected to ground, which overrides the pullup resistor and causes the input to read as `LOW`.

Assignment:

Connect a piezo buzzer and at least 3 buttons to your Arduino and program it to play a different musical note on the buzzer while each button is held. If you pick the right notes (like C, D, and E) you can even play a simple song, like “Mary Had a Little Lamb” or “Hot Cross Buns”. Look at the end of the previous project for a list of the frequencies of different musical notes.

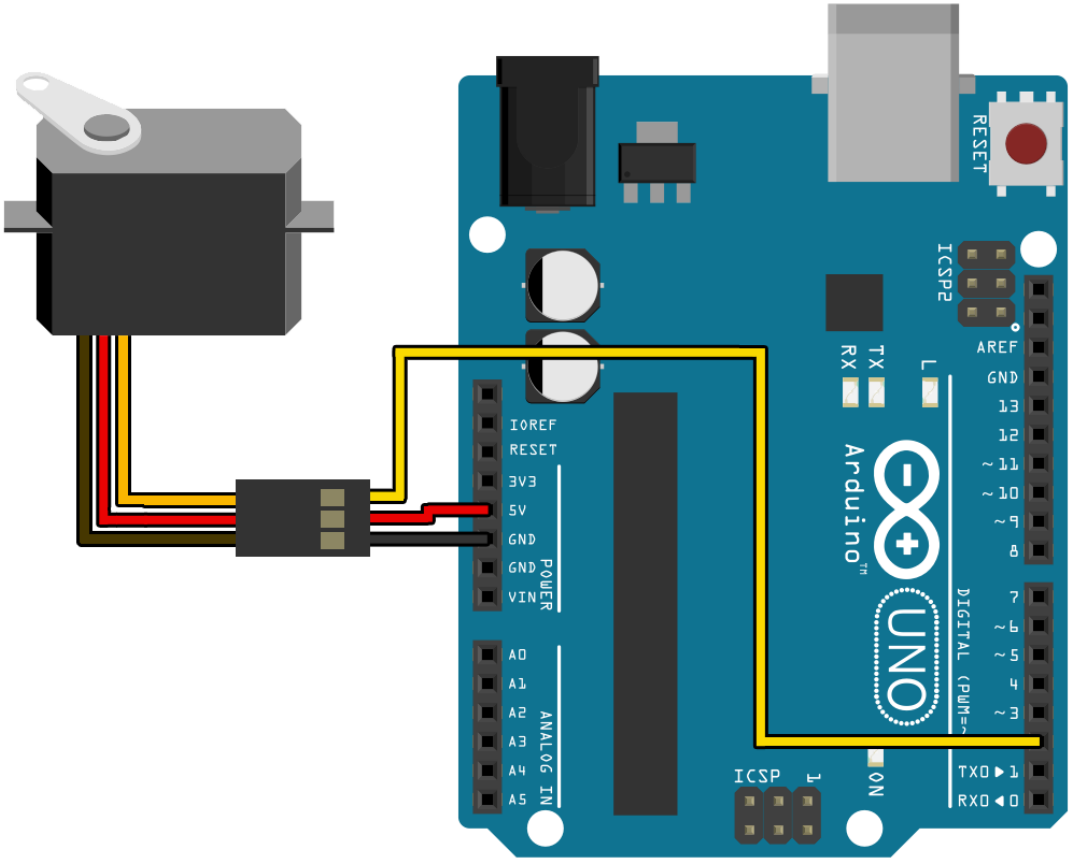
Additional XP: Instead of simply having each button correspond to a single note, program combinations of buttons to play different notes. You can play up to seven different notes with three buttons and up to 15 different notes with four buttons. With each button having two states (“pressed” or “not pressed”) this is just like a binary number. Look back at the deep dive after Project 3 for more information about how to write complex `if` / `else` statements.

Project 8: Servo Motors

Electric motors convert electrical power into angular motion. Most motors spin continuously, like a cordless drill or the wheels on an electric vehicle, with a speed dependent on how much power is applied. Servo motors are a special type of motor that, instead of spinning continuously, rotate to a specific angular position based on an electrical signal. They are particularly useful in robotics, where they are used for range-limited movements like opening and closing a claw or steering a car.

Example Program:

Assemble the following circuit and upload the program to your Arduino:



```
#include <Servo.h>

Servo myServo;

void setup()
{
      myServo.attach(2);
}

void loop()
{
      // Move the servo all the way to one end:
      myServo.write(0);
      delay(1000);

      // Move the servo all the way to the other end:
      myServo.write(180);
      delay(1000);
}
```

You should see the servo moving back and forth over an arc of about 180 degrees, which happens to be its full range of motion. It can be helpful to attach one of the plastic accessories that came with the servo motor to better visualize its motion; just be sure to do this while it is not turned on and moving!

- If you want to interface your Arduino with a servo motor the first thing you will need to do is add `#include <Servo.h>` to the top of your program like in the example above. This tells the C compiler to include the Servo *library*, which contains a bunch of code that you can use to control a servo motor without having to know the nitty-gritty details of how they work. One of the biggest advantages of using an Arduino is the huge number of libraries available for interfacing with other components.
- Then, for each servo motor that you want to control, you must create a `Servo` object at the start of your program (like `Servo myServo;`) and call its `attach()` method in `setup()` with the pin that the servo is connected to (here, pin 2). The specifics of *objects* and *methods* are beyond the scope of this project and AP CSP, but know that objects are a lot like variables in that you have to declare them ahead of time and each one needs a unique name, like `myServo` in the example above.
- Once we have declared and attached our `Servo` object we can call the `write()` method to set its position to an angle between 0 and 180, the full range of the servo. The motion of a servo is fast but not instantaneous, so if you want to wait until a servo reaches its destination you will need to call `delay()` after `write()`, like in our example above.

- When a servo motor is powered but sitting idle it will use about 10 mA of current. When the motor is moving from one position to another the current use dramatically increases to something closer to 250 mA. If a force is applied to resist the motor's movement it will *stall* and consume even more current, as high as 350 mA. Why does this matter? Any device hooked up to a PC using a USB port (like your Arduino) is only allowed to use up to 500 mA of current. If it exceeds this limit it will be disconnected. Be mindful of this limit when powering servo motor circuits over USB.

Assignment:

Build a "servo motor tester" using a servo motor and a potentiometer. As you turn the potentiometer knob over its full range of motion (0 to 1023) it should also move the servo motor over its full range of motion (0 to 180). While this is happening, your program should also print the servo motor position value in the serial monitor. This program will be useful if you decide to go on and do the "build an automaton" capstone project, as you can use it to directly control a servo motor and calibrate any servo motor position values you need to hardcode into the program.

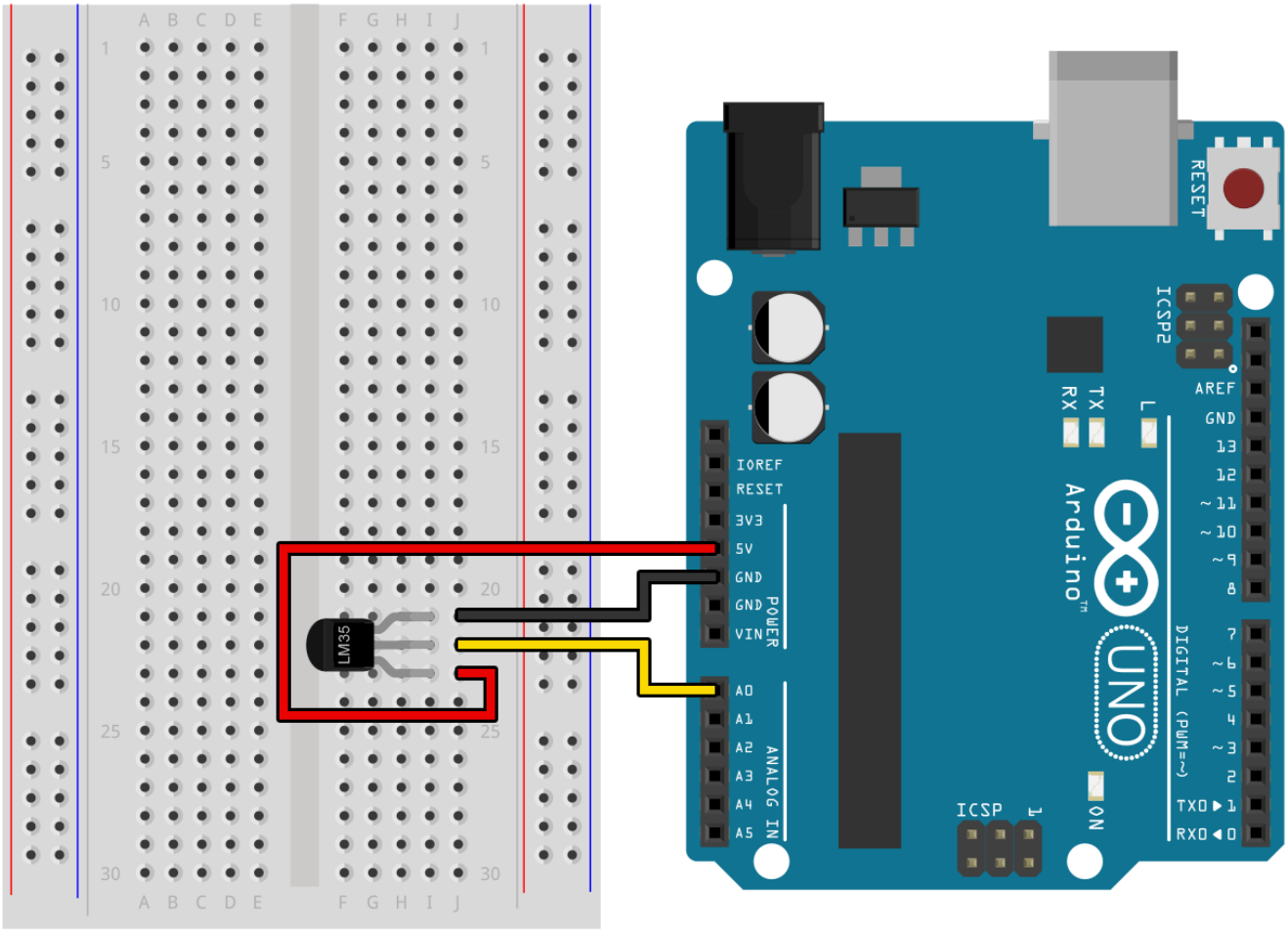
Do you remember the `map()` function from Project 5? It will be useful here too.

Project 9: Temperature Sensors

When correctly powered, this sensor generates a voltage that increases linearly with its temperature and can be read by a microcontroller. Although its pins are not labeled you must take care to hook it up correctly. **If you hook up the temperature sensor incorrectly it can get very hot and burn you.**

Example Program:

Assemble the following circuit and upload the program to your Arduino:



Pay close attention to this diagram. The flat side of the temperature sensor (with the part number barely visible) is facing **toward** the Arduino. The curved side of the temperature sensor (with no text) is facing **away from** the Arduino.

```
void setup()
{
      Serial.begin(9600);
      pinMode(A0, INPUT);
}

void loop()
{
      Serial.println(analogRead(A0));
}
```

If you open the serial monitor in the Arduino IDE you should see a stream of numbers being printed to it. These are the results of using `analogRead()` to read the sensor voltage, which encodes the temperature measured by the sensor.

Assignment:

Write a program that reads from the temperature sensor and prints the voltage and temperature (in both Celsius and Fahrenheit) to the serial monitor, creating a table of values that puts all three values on a single line of text and looks something like this:

```
0.24 V  |  23.93 °C  |  75.07 °F
0.24 V  |  23.93 °C  |  75.07 °F
0.24 V  |  24.41 °C  |  75.95 °F
0.24 V  |  24.41 °C  |  75.95 °F
```

To convert the values returned by `analogRead()` into a voltage, remember that `analogRead()` converts a voltage range (0 V to 5 V) into a 10-bit integer (0 to 1023). A value of 45 corresponds to a voltage of 0.22 V.

From there, we can consult the datasheet for the temperature sensor (part number LM35) to see that its voltage starts at 0 V and rises 10 mV (0.01 V) for every degree Celsius above zero. A voltage of 0.22 V corresponds to a temperature of 22 °C.

To convert from Celsius to Fahrenheit, use the following formula:

$$F = (C \times \frac{9}{5}) + 32$$

All this math is a perfect use for variables, which you learned about in the deep dive that followed Project 2. Because many of these values are not integers, however, you must switch from using `int` to `float` when declaring all your variables in this project. This will allow you to store decimal values like 0.22 and 0.01; they will be automatically rounded down to 0 if you use `int` instead of `float`, which is definitely not what you want!

Project 10: X/Y Joysticks

The X/Y joystick in your kit is similar to the analog sticks on video game controllers. From an electrical standpoint it is just two potentiometers (X and Y) and a button (for when the joystick is clicked).

Example Program:

Connect the X/Y joystick to your Arduino by making the following connections with the long M-to-F jumper wires included in your kit, and then upload the program to your Arduino:

X/Y joystick	GND	+5V	VRX	VRY	SW
Arduino pin	GND	5V	A0	A1	2

```
void setup()
{
      Serial.begin(9600);
      pinMode(A0, INPUT);            // X-axis
      pinMode(A1, INPUT);            // Y-axis
      pinMode(2, INPUT_PULLUP);      // Button
}

void loop()
{
      Serial.print("X=");
      Serial.print(analogRead(A0));
      Serial.print(" | Y=");
      Serial.print(analogRead(A1));
      Serial.print(" | B=");
      Serial.println(digitalRead(2));
}
```

If you open the serial monitor in the Arduino IDE you should see a stream of values being printed indicating the X and Y position of the joystick (from 0 to 1023) and whether or not the button is currently pressed down.

Assignment:

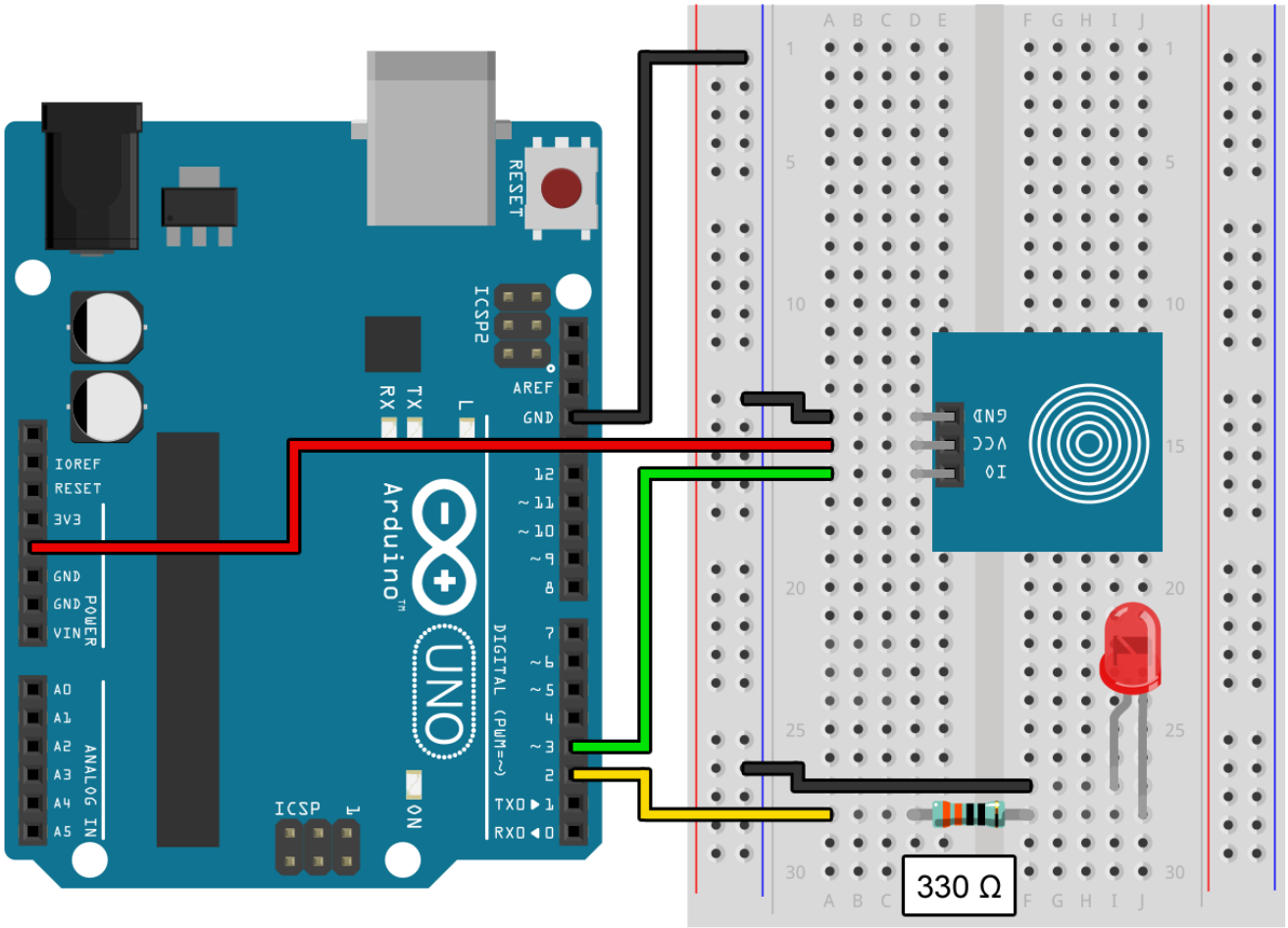
Wire up four LEDs that correspond to the +X, -X, +Y, and -Y directions and have them turn on when the joystick is pushed in that direction. When the joystick is pushed diagonally in two directions at once (like +X and +Y) both LEDs should turn on. When the joystick is centered the LEDs should all be off. When the joystick button is pushed the LEDs should all be turned on, regardless of whether the joystick is centered or being pushed in a direction.

Project 11: Touch Sensor

Although it looks nothing like a phone, this touch sensor uses the same general principle as a phone's touch screen: capacitive touch sensing. When you touch it you in essence become part of the circuit, not through normal conduction (because then it would be electrocuting you!) but by becoming part of the capacitor and changing its capacitance.

Example Program:

Assemble the following circuit and upload the program to your Arduino:



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

void loop()
{
      if (digitalRead(3) == HIGH)
      {
            // Turn on the LED when the touch sensor is touched:
            digitalWrite(2, HIGH);
      }
      else
      {
            // Turn off the LED when the touch sensor is not touched:
            digitalWrite(2, LOW);
      }
}
```

The LED should turn on when you touch the circular area on the touch sensor and turn off when you stop touching it.

- The touch sensor is a lot like a button or a switch, except that you should use `INPUT` instead of `INPUT_PULLUP` when configuring the pin. Unlike a button that connects to ground when pushed and is disconnected when not pushed, the touch sensor always puts out a `HIGH` or `LOW` signal for the Arduino to read. Thus, no need for the pullup.
- The touch sensor is also unlike a button or switch because `digitalRead()` will return `HIGH` when you are touching the sensor and `LOW` when you are not.

Assignment:

Using an LED connected to an `analogWrite()` compatible pin and a touch sensor connected to any other I/O pin, create a circuit for a “touch-sensitive lamp” that changes brightness when you touch it. Touching it once should turn the light on to about 33% brightness, touching it again should increase the brightness to 66%, touching it again should turn it all the way on, and touching it again should turn it off so that the process can be repeated.

Although the circuit for this assignment is easy, the code is not. In all the other projects with buttons and switches we only cared if a button was held down. In this assignment we care about when the touch sensor changes state, from released to pressed. In order to do this we will need to remember the previous state of the touch sensor using a variable. Fortunately, `HIGH` and `LOW` can be treated as integer values and stored in an `int`.

The following code, when combined with the example circuit in this project, blinks the LED whenever the touch sensor goes from released to pressed or pressed to released (that is, whenever the touch sensor changes state):

```
int previousState = LOW;

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

void loop()
{
      // Determine whether the touch state has changed:
      int currentState = digitalRead(2);
      if (currentState != previousState)
      {
            // Check against this new state from now on:
            previousState = currentState;

            // Quickly blink the LED on pin 3:
            digitalWrite(3, HIGH);
            delay(100);
            digitalWrite(3, LOW);
      }
}
```

You will most likely also need to use a variable to store the current brightness so that you can increment it when the button is pressed. An `int` will be sufficient for this too.

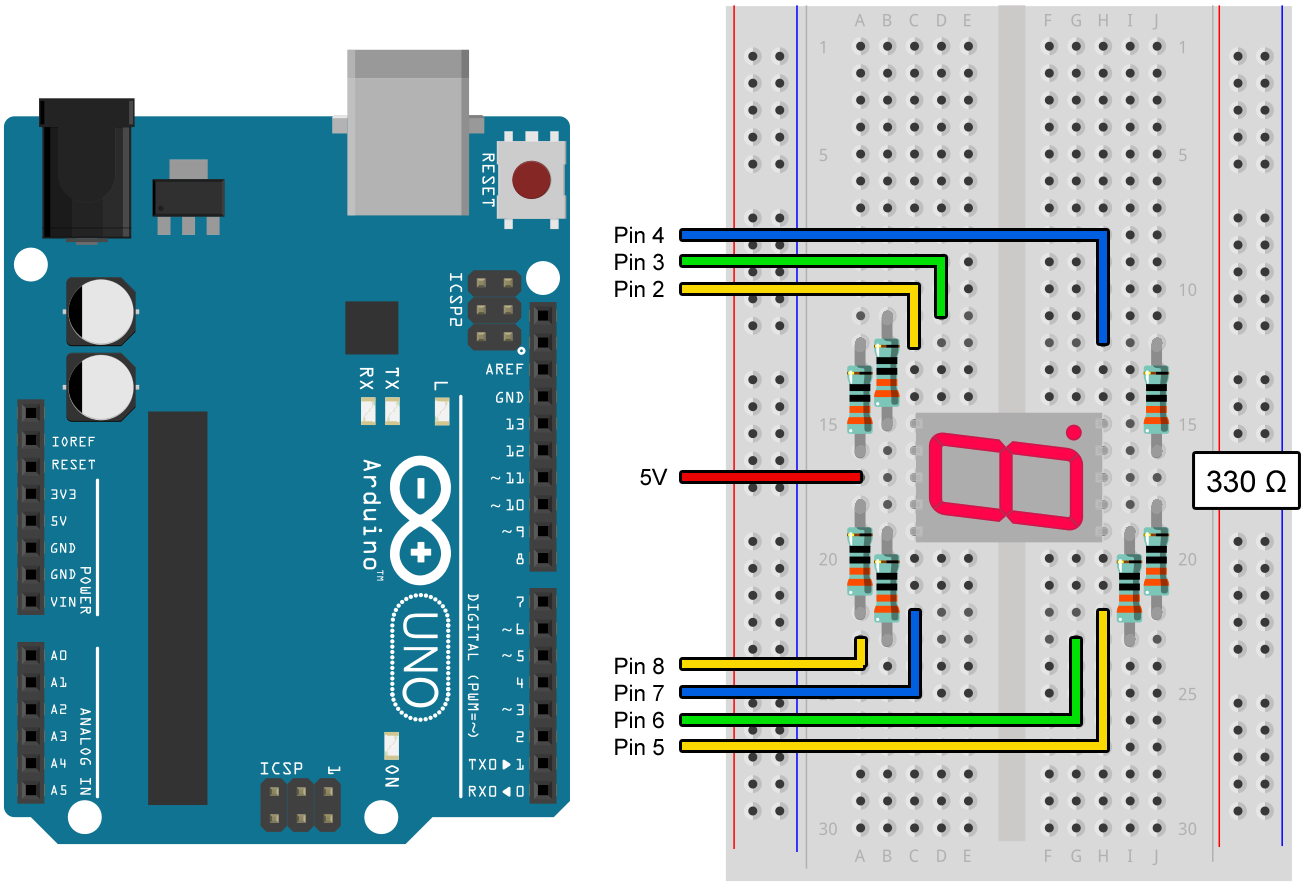
Project 12: 7-Segment Displays

Although they are a little outdated today, 7-segment displays were widely used in electronics of the last 50 years because they are essentially the simplest way to display numbers to a human (without requiring them to read binary, which is technically more “efficient”). If you are creative you can draw all 16 hexadecimal digits on a 7-segment display, and even some words if you are okay with ambiguous characters, but to draw the full English alphabet without ambiguity you would need a 14-segment or 16-segment display.

Similar to how the RGB LED actually contained three smaller LEDs (red, green, and blue), the 7-segment display contains eight LEDs: one for each segment of the digit, and one for a small dot that can be used as a decimal point. Just like with the RGB LED each internal LED needs its own resistor.

Example Program:

Assemble the following circuit and upload the program to your Arduino:



```
void setup()
{
	// Configure pins 2-8 as outputs:
	for (int i = 2; i < 9; i++)
	{
		pinMode(i, OUTPUT);
	}
}

void loop()
{
	// Turn off all the segments at once:
	for (int i = 2; i < 9; i++)
	{
		digitalWrite(i, HIGH);
	}
	delay(250);

	// Turn on the segments one at a time:
	for (int i = 2; i < 9; i++)
	{
		digitalWrite(i, LOW);
		delay(250);
	}
}
```

You should see the segments of the display repeatedly turn on in order and then all turn off.

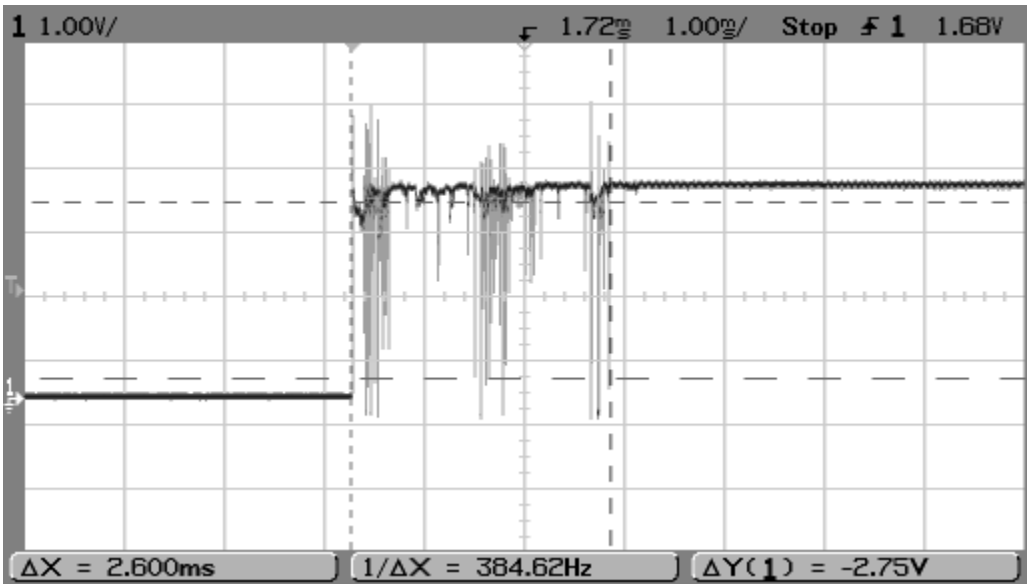
- In every other project that used an LED we connected the longer *positive* leg of the LED to an I/O pin (through a resistor) and the shorter *negative* leg of the LED to ground. This made it so that setting the I/O pin `HIGH` would turn the LED on and setting the I/O pin `LOW` would turn the LED off. The LEDs built into the 7-segment display in your kit are hooked up backward, which means that (1) we need to hook up the other end of the LED to 5 volts, not ground, and (2) we will turn the LED on by setting the I/O pin `LOW`, not `HIGH`. If you look closely at the calls to `digitalWrite()` in the example you can see this.

Assignment:

Using a 7-segment display and two buttons, create a counter that counts between 0 and 9, incrementing when one button is pushed and decrementing when the other button is pushed.

Although you will need to figure out how to implement it in code yourself, here is the common way that 7-segment displays are used to show the digits 0 through 9:

In the last project you learned how to detect a state change (or “edge”) in an input signal, like a touch sensor going from released to pressed. If you try to do something like this with a mechanical button, you will need to deal with a phenomenon called *contact bounce*, which is where the mechanical transition from released to pressed involves a bunch of rapid oscillations between both states as the metal contacts press together, causing the signal to fluctuate.



If you do not *debounce* your signal you will find that a single button press can easily trigger your state change detection code ten or more times. Adding a short delay (~50 ms) after a state change is detected is usually enough to prevent this from being a problem.

Project 13: Ultrasonic Ranger

The ultrasonic ranger in your kit uses ultrasonic sound (sound frequencies that are too high to be heard by humans) to measure the distance to the nearest obstacle and convert it into a signal that can be read by a microcontroller. This is similar to how bats use echolocation to “see” in the dark.

Example Program:

Connect the ultrasonic ranger to your Arduino by making the following connections with the long M-to-F jumper wires included in your kit, and then upload the program to your Arduino:

Ultrasonic ranger	Vcc	Trig	Echo	Gnd
Arduino pin	5V	2	3	GND

```
long readDistance(int trigPin, int echoPin)
{
      // Pulse the TRIG pin for 10 microseconds:
      digitalWrite(trigPin, HIGH);
      delayMicroseconds(10);
      digitalWrite(trigPin, LOW);

      // Measure the length of the pulse on the ECHO pin:
      return pulseIn(echoPin, HIGH);
}

void setup()
{
      Serial.begin(9600);
      pinMode(2, OUTPUT);
      pinMode(3, INPUT);
}

void loop()
{
      Serial.println(readDistance(2, 3));
}
```

If you open the serial monitor in the Arduino IDE you should see a stream of numbers being printed to the screen that get lower the closer you move an obstacle (like your hand or a book) to the front of the ultrasonic ranger.

- The ultrasonic ranger sends out an ultrasonic pulse when it receives a `HIGH` signal on its trigger ("TRIG") pin for at least 10 µs (microseconds). It then makes its "ECHO" pin go `HIGH` until the sound bounces back and is detected by the sensor, at which point it goes `LOW` and the measurement is complete. The function `readDistance()` in the example code above generates that 10 µs trigger pulse and then uses the built-in Arduino function `pulseIn()` to measure the length of the echo pulse (also in µs), which turns out to be directly proportional to the distance between the sensor and the nearest obstacle according to the speed of sound (343 m/s in dry, room-temperature air).

Assignment:

A theremin is an electronic musical instrument where pitch and volume are controlled by the distance between "antennas" on the instrument and the operator's hands. If you are not familiar with the instrument you can look on YouTube for examples of them being played (or visit carolinaeyck.com). The way that theremins work is much more like the capacitive touch sensor in Project 11 than the ultrasonic ranger, but nevertheless we can use it to make a musical instrument that operates like a theremin.

Create a circuit with the ultrasonic ranger and a piezo buzzer that varies the pitch of the buzzer as the distance to your hand changes, so that the pitch rises as your hand gets closer and stops making noise when you move your hand away completely. If you are having trouble getting the sensor to detect your hand, try using something large and flat like a book.

The readings from the ultrasonic ranger can be erratic. You may need to average together a few readings to get a "cleaner" signal. If you do, you will probably want to use variables of type `long` (32 bits) instead of `int` (16 bits) to avoid an *overflow* when adding together sensor readings (which can go as high as 38000).

Project 14: Rotary Encoders

The rotary encoder looks like a potentiometer but is actually digital instead of analog and can be rotated continuously instead of being limited to a small angular range. You can also press it like a button. They are not alike at all!

Example Program:

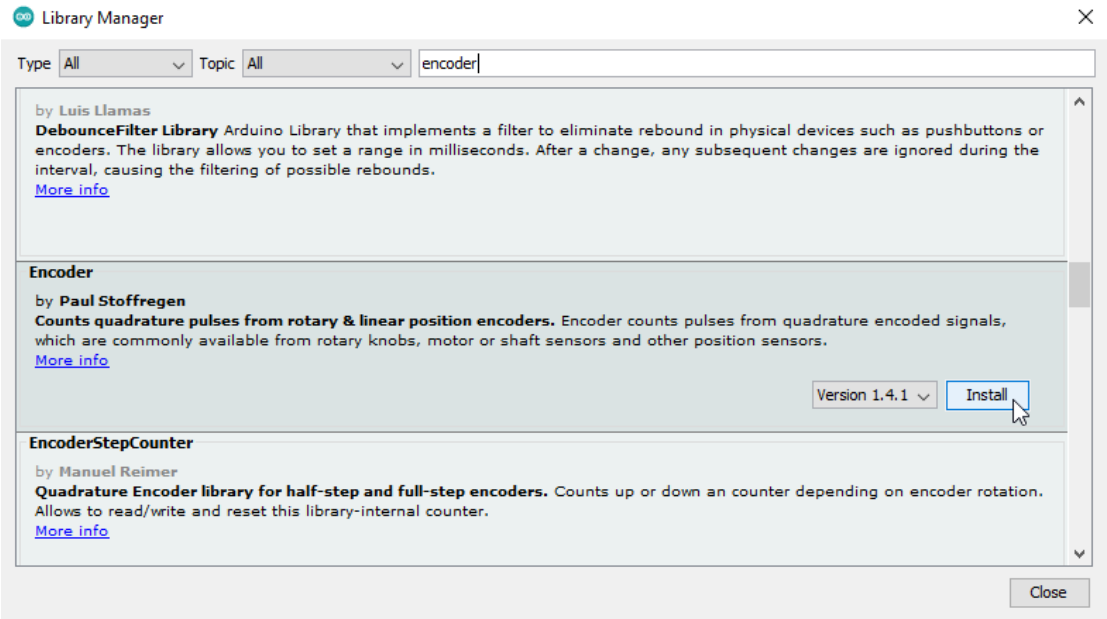
Connect the rotary encoder to your Arduino by making the following connections with the long M-to-F jumper wires included in your kit, and then upload the program to your Arduino:

Rotary encoder	CLK	DT	SW	+	GND
Arduino pin	3	2	4	5V	GND

```
#include <Encoder.h>

Encoder encoder(2, 3);

void setup()
{
      Serial.begin(9600);
      pinMode(4, INPUT_PULLUP);
}

void loop()
{
      // Reset the rotary encoder's logical position when clicked:
      if (digitalRead(4) == LOW)
      {
            encoder.write(0);
      }

      Serial.println(encoder.read());
}
```

Before you can compile and upload this program to your Arduino you will need to open the library manager in the Arduino IDE ("Tools > Manage Libraries…", or Ctrl+Shift+I), type "Encoder" into the search bar, and install the "Encoder" library by "Paul Stoffregen". (See screenshot on next page.)

Once you have installed the library and compiled and uploaded the program you should see a stream of numbers being printed to the serial monitor that increase when you turn the knob clockwise, decrease when you turn the knob counterclockwise, and reset to zero when you push the knob down so that it clicks like a button.



- Similar to the servo motors in Project 8, using the rotary encoder requires you to put `#include <Encoder.h>` at the top of your program and create an `Encoder` object, this time specifying the pins that it is connected to (2 and 3) when you declare it.
- By calling the encoder's `read()` function you can get its current position, which changes when it is turned in either direction.
- By calling the encoder's `write()` function you can reset the number that is being adjusted by the knob. Unfortunately, this does not also change the *physical* position of the knob.
- The rotary encoder conceals a button, much like the buttons in your kit, that connects the "SW" pin to ground when pressed. By connecting this to an I/O pin and configuring it for `INPUT_PULLUP` we can use it as an input, just like we did in Project 7.
- Although you can technically connect the "DT" and "CLK" pins of the rotary encoder to any two `digitalRead()` capable pins on the Arduino, pins 2 and 3 are special in that they support something called *hardware interrupts* that make the rotary encoder much more responsive and precise when you use those pins specifically. You do not need to understand why, you just need to know that it works!

Assignment:

Recreate the circuit from Project 3 using the rotary encoder instead of a potentiometer. At any given time only a single LED should be on; turning the knob should light up the previous or next LED in the sequence based on which direction the knob was turned. When the end of the sequence is reached, turning the knob further in that direction should "wrap around" and turn on the LED at the opposite end of the sequence, so that the knob can be continuously turned in either direction to repeatedly cycle through the LEDs. You can accomplish this by using the encoder's `write()` function to change the encoder's value when it goes "out of bounds" on either end.

Project 15: Character LCDs

There are many different types of *liquid-crystal displays* (or “LCDs”) that have been made over the years. The one in your kit is an old type that has been around for decades and is more like a digital watch than a modern phone screen (even though both are technically LCDs). They use a *backlight* to light up the entire screen, and then use an electrically controlled material to selectively block out the light and create darker colors.

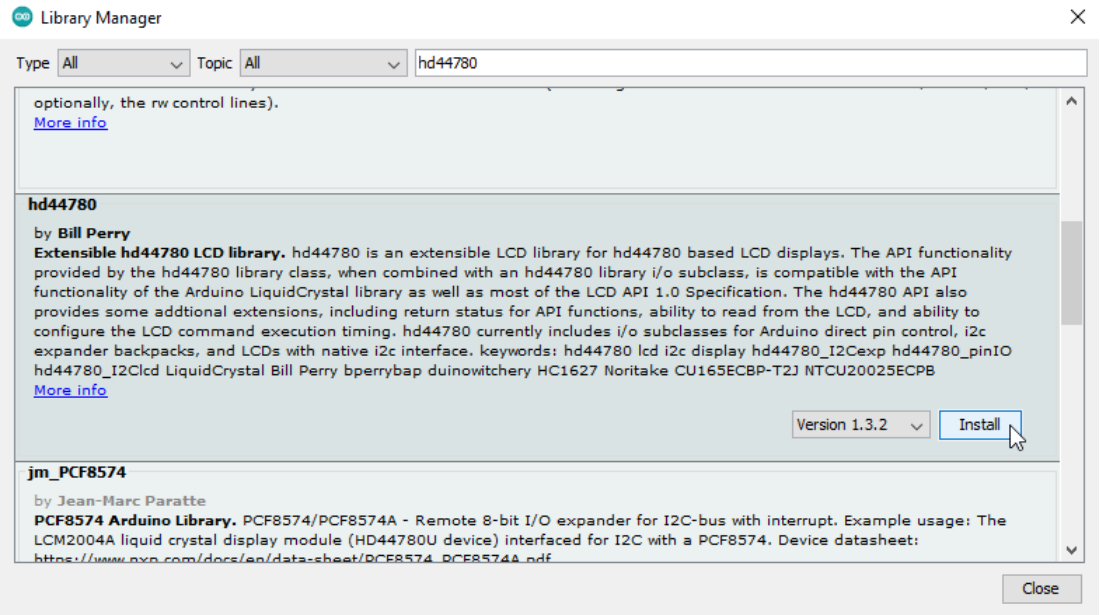
Example Program:

Connect the character LCD to your Arduino by making the following connections with the long M-to-F jumper wires included in your kit, and then upload the program to your Arduino:

Character LCD	GND	VCC	SDA	SCL
Arduino pin	GND	5V	A4	A5

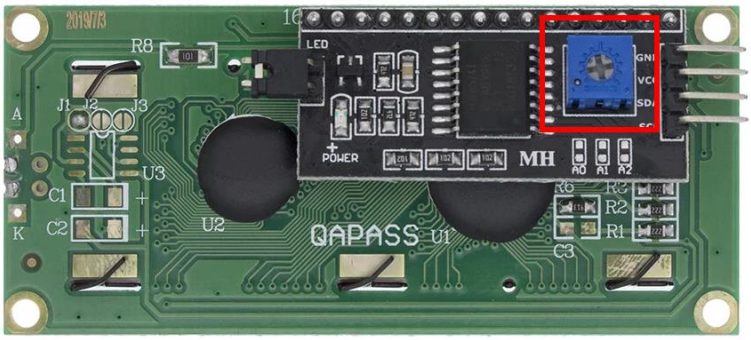
```
#include <Wire.h>
#include <hd44780.h>
#include <hd44780ioClass/hd44780_I2Cexp.h>

hd44780_I2Cexp lcd;

void setup()
{
      // Initialize the LCD:
      lcd.begin(16, 2);

      // Print an unchanging message on the top row of the LCD:
      lcd.print("I have been on");
}

void loop()
{
      // Re-print the changing message on the bottom row of the LCD:
      lcd.setCursor(0, 1);
      lcd.print("for ");
      lcd.print(millis() / 1000);
      lcd.print(" seconds.");
}
```

Before you can compile and upload this program to your Arduino you will need to open the library manager in the Arduino IDE (“Tools > Manage Libraries…”, or Ctrl+Shift+I), type “hd44780” in the search bar, and install the “hd44780” library by “Bill Perry”.



Once you have installed the library and compiled and uploaded the program you should see some text printed on the LCD screen. If you do not, quickly double-check your wiring. If everything is wired correctly and the backlight is on but you do not see any text, you probably need to adjust the contrast on the LCD screen.

Get a small Phillips head screwdriver and use it to adjust the blue potentiometer on the circuit board on the back of the LCD screen. If you turn it all the way clockwise you will see all of the pixels (even the ones that are off), and if you turn it all the way counterclockwise you will see nothing but the backlight. Find a happy medium where the text is clearly visible without the pixels that are off also being visible.

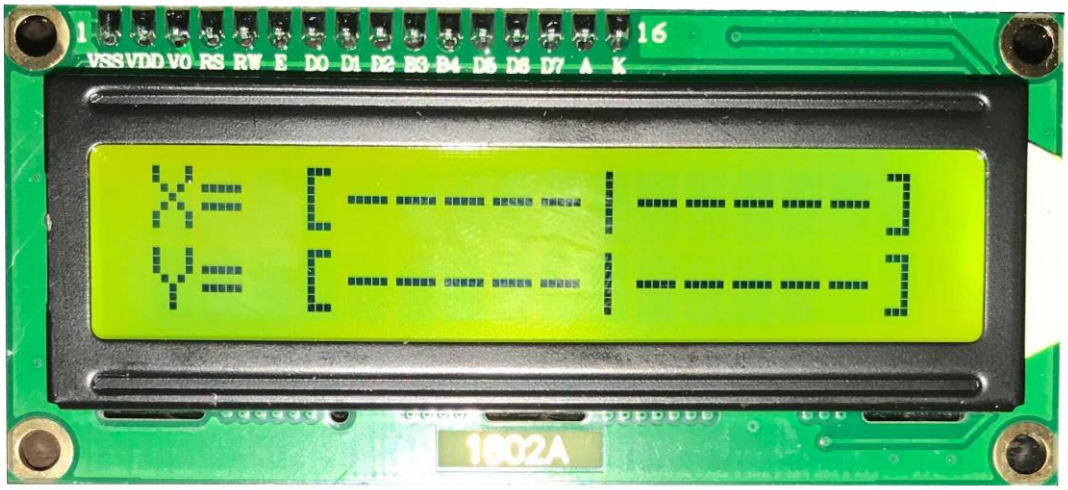


- Printing text to the LCD is a lot like printing text to the serial monitor, except that you have more control because you can move the *cursor* around and print new text over top of old text. This allows you to create things like user interfaces and animations. Check out the deep dive at the end of this project for a full list of functions supported by the hd44780 LCD library.

- The specific screen in your kit is called a dot-matrix display because it uses a grid of dots (or pixels) to create an image. Compare this with the 7-segment display from Project 12, which was only able to display digits.
- On second thought, that maybe was not entirely true; the specific screen in your kit is not *truly* a dot-matrix display because it cannot display arbitrary pixel patterns. Instead, it is grouped into two rows of 16 *characters*, each consisting of a 5x8 sub-grid of pixels that is used to display a single digit, letter, or symbol. These are commonly called *character LCDs* and are still much more versatile than 7-segment displays.

Assignment:

Use one of the sensors from a previous project (temperature sensor, X/Y joystick, ultrasonic ranger, photoresistor, potentiometer, etc.) and create a “dashboard” on the LCD that visually displays and updates the information from the sensor in real-time. You should do something more interesting than merely printing out a number, like this example below that shows the position of the X/Y joystick as two animated meters, one for each axis. Be creative!



Deep Dive: The hd44780 Library

In the example above we created an `hd44780_I2Cexp` object called `lcd` that was used to control the LCD screen (similar to how we created a `Servo` object to control each servo motor in Project 8). Here are some useful methods that can be used with an `hd44780_I2Cexp` object:

`begin(w, h)`	Initialize the screen, letting the library know what size it is (ours is 16x2).
`clear()`	Clear the screen.
`print()`	Print a number or text string to the screen.
`setCursor(x, y)`	Move the cursor (where text prints) to position (x, y). The position (0, 0) is the top-left corner of the screen and is where the cursor starts. In the example program, when we use `lcd.setCursor(0, 1)` at the beginning of `loop()`, we are moving the cursor to the first character in the second row so that we only need to update that row of the message.
`blink()`	Turn on the blinking cursor. (Initially off.)
`noBlink()`	Turn off the blinking cursor.
`backlight()`	Turn on the backlight. (Initially on.)
`noBacklight()`	Turn off the backlight.

The full documentation for the LCD library is available here. There are more functions available, although most of them are just for making the contents of the screen scroll as text is printed.

By using the `createChar()` function it is possible to create and display your own custom characters on the LCD, although you will have to consult the documentation and library examples to figure out how to use it in your projects.

Capstone Projects

For the same amount of XP as three projects (!) you may attempt a *capstone project* that combines together the skills you learned in these projects (or a previous Arduino-focused activity that you did outside of this class) to create something meaningful. There are a few guidelines:

1. In order to get three times the XP you must create something that is at least three times as complex as one of the projects. This means more code, or more components, or more complexity in some other way.

2. Before you are approved to start a capstone project you must write up a small *design document* that describes what you want to build so that your instructors can verify that it is not too easy or too complex for the scope of this class. Write complete sentences! Draw diagrams! Express yourself! It is not essential that you stick to your design document exactly, but you need to know where you are headed when you get started.

3. There is nothing wrong with hooking up components to an Arduino and making them blink and beep without purpose, but it is definitely a little… artificial. What you build for this project should *be something*. See below for examples of things you could make that have a little bit more *something* to them than the previous projects did.

Build a game

There were lots of handheld electronic games manufactured and sold in the 1970s and 1980s that were made with the same level of technology as the parts in your Arduino kits, like Mattel Football and Simon. For this project you should build your own game. It probably will not be incredibly fun, but it should be something that someone can play and either win or get better at. Here is an example game:

Four LEDs are hooked up to the Arduino, each with a corresponding button, along with the LCD and buzzer. After a short count-down, the Arduino randomly lights up one of the LEDs; if the player pushes the corresponding button they get a point and another LED lights up, but if they push the wrong button they instantly lose, which is announced by the buzzer. At the end of 10 seconds the game ends (if the player has not already lost), with the player's score being the number of buttons they managed to correctly push before running out of time. Both the time remaining and the current score are displayed on the LCD at all times.

Games often involve generating random numbers. You can do this with the `random()` function.

Build a gadget

Another possibility is to build a “gadget” that solves a real-world problem, even if it is silly, contrived, or already solved. Although a lot of things in this category would depend on specialized hardware that is not in your kit, there are lots of “toy” versions of real-world solutions you could build, like:

- a programmable thermostat (temperature sensor, LCD, buttons, red and blue LEDs)
- a clock, stopwatch, or kitchen timer (LCD, buttons, switches)
- a configurable metronome (LCD, rotary encoder, piezo buzzer)

Build an automaton

The word *automaton* generally refers to a machine that operates under its own control, but also specifically refers to a type of mechanical puppet that moves on its own and performs parlor tricks, like shooting an arrow or writing on paper. They were very popular with rich people hundreds of years ago. There was even a famous one called the Mechanical Turk that played chess… and was only able to do so because it hid a person inside.

This capstone project requires you to build an automaton that looks and acts like something specific. In addition to the parts in your kit, you will need some craft supplies, like colored pencils or markers, cardstock, and tape. Be creative! Some examples include:

- a cartoon head that opens its mouth to sing (servo motor, piezo buzzer)
- a pair of arms that use flag semaphore to spell out a message (both servo motors)
- a video game character who jumps when you push a button (servo motor, buzzer, button)

If you decide to use your second servo motor you will need to create a second `Servo` object to control it. Also, take another look at the note about servo motor power usage in Project 8; you may run into strange glitches (like your Arduino unexpectedly restarting) when using both motors at once.

Build something artistic

If you are feeling exceptionally creative, you could also create an art project using the components in your kit and whatever other supplies you have on hand. Although it is impossible to pin down what is and is not art, the Arduino needs to add some kind of complexity or interactivity so that the programming has substance.

If you decide to go in a musical direction, consider using the Tone library, which makes it possible to play up to three notes at once, each on a separate piezo buzzer. Another trick for constrained musical environments (like old video game consoles) is to use *arpeggios* and play the notes of a chord in rapid sequence, which only uses a single sound at a time but still kind of sounds like a chord. Although it is a far cry from a proper synthesizer, it could be an interesting project to program a multi-instrument song into an Arduino and have it synchronize lights to the music.