Introduction to Arduino

ENGR 1624

Introduction

Arduino is a common tool for introducing students to programming and prototyping with microcontrollers. With Arduino, the barriers to entry are very low, but the system can be made to do very powerful things. We don't have time to cover much, but it will be useful for everyone on your team to be familiar with how to load and run code and the basics of programming.

In this exercise, you'll build some circuits and learn some of the basic Arduino functions – the building blocks, if you will – of the Arduino environment. We'll provide you with some skeleton code (i.e., code that is largely already written but still needs some meat on its bones) and ask you to change parameters and logical elements. For those that are interested in going further, sign up for the "Power to Control: Advanced Arduino" superpower.

Objectives

Upon successful completion of this lab, a student will be able to:

- Build introductory circuits on a breadboard,
- Edit and upload code to an Arduino,
- Use basic Arduino routines to read sensors and control outputs,
- Use the Serial Monitor to output data to a computer, and
- Use an Arduino library, in this case the `Servo` library.

Preparation

Background materials

Review the following materials:

- Read up on circuits basics.
- Read about how to use a breadboard.
- Download and unzip the sample codes needed for this lab as follows:

1. Download the `arduino.basic` codes from github here. Click on the "Clone or Download" button and download them as a .zip file.
2. If you haven't already, install the Arduino IDE. See instructions online.
3. Open the Arduino IDE find the "Preferences" under one of the menus. Find the location of your Sketchbook under Preferences and unzip the file you downloaded above into that location. Do not just move the .zip file there – you may need to "extract" that file first if your operating system does not do that automatically (Windows typically does not).
4. Open the Arduino IDE and verify that the codes can be found under File → Sketchbook → arduino.basic.

You need to show up on Tuesday with the code ready to go!

If you're interested in learning more about some of the topics presented herein:

- SparkFun has a decent tutorial about PWM.
- As well as a technical discussion of analog vs. digital.
- Way more than you'd ever want to know about LEDs.

I/O Fundamentals

Physically, input and output with an Arduino happen through *pins*, which aren't actually pins but the sockets along the sides of the Arduino.[1] Each of the pins is labelled (e.g., `8` or `A2` or `GND`. Some of the pins have special functionality (you might have noticed a `~` next to some of them), and we'll explain some of that functionality in the tutorials (you must be on the edge of your seat...).

Digital output: Hello, World!

As you saw in the background reading above, you can connect a pin on an Arduino to a circuit using a *jumper wire*, a short piece of wire that has convenient, exposed ends that can be pushed into the Arduino or the breadboard. Here, you'll first blink the built-in LED on the Arduino, and then you'll build a circuit with a separate LED on your breadboard.

Procedure

1. Connect the Arduino to your computer with a USB cable and open the Arduino IDE.
2. Navigate to Tools → Board and select "Arduino/Genuino Uno" if it is not already selected.
3. Navigate to Tools → Port and select the port that corresponds to your device. On a Windows machine, it will likely show up as "COM #". On a Mac, it will likely have "Arduino" or "redboard" in the name. If it's not obvious, just try one or call over an instructor. Better yet, unplug the Arduino, note which ports are in the list, plug the Arduino back in, and look for the new port.

[1]They're called pins because the microcontroller chip – the brains of the Arduino – uses pins to communicate with the outside world.

4. Open the `blink` example by selecting File → Sketchbook → arduino.basic → blink.
5. Upload the program by clicking the right facing arrow near the top-left of the IDE. The IDE will compile and upload the program, which starts running as soon as the upload is done. You should see the on-board LED on the Arduino blink on and off every second.

Congratulations, you've now completed the "Hello, World!" equivalent of Arduino! Of course, blinking an on-board LED isn't terribly useful, so let's do something slightly less useless and build a circuit on a breadboard.

Procedure

1. Disconnect the Arduino from your computer and build the circuit shown in Figure **??**.
   - It is always a good idea to disconnect the Arduino when building a new circuit to avoid any possibility of a short circuit.
   - Note that the long leg of the LED should be connected to pin 13. This is the positive side of the LED. The negative side is marked not only by the shorter leg but also by the flat edge on the LED casing.
   - For this circuit, you'll need the 330Ω resistor, which is marked with orange-orange-brown-gold stripes. For more on resistor sizing and markings (on your own time), SparkFun has a good description.
2. Reconnect the Arduino. Since the code you loaded before lives in persistent memory, it will start running as soon as you connect the Arduino. You should see the on-board LED start to blink again, and if you did everything correctly, the LED on your breadboard will also blink.

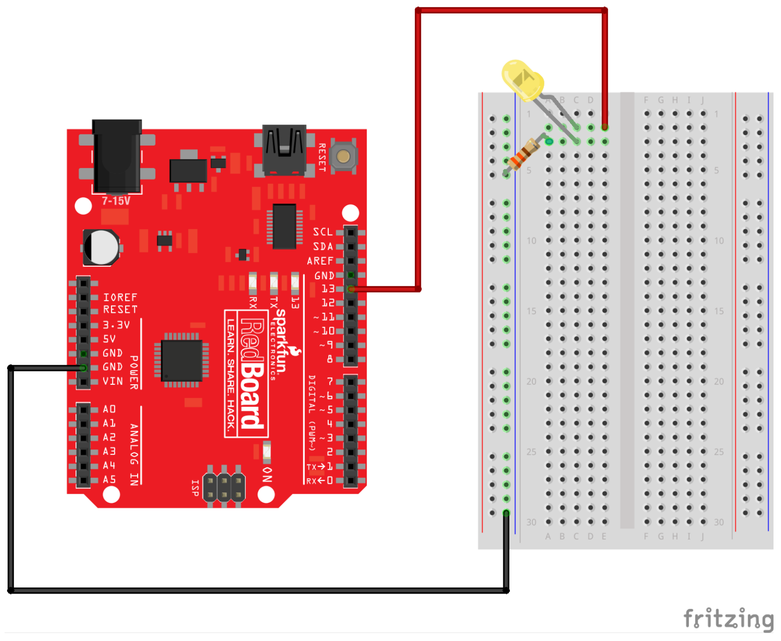

Figure 1: Fritzing diagram for the LED breadboard circuit.

In a moment, you'll make edits to the code, but first, let's explain what the various lines of code are doing. Here is the complete program:

```
const int ledPin = 13;

// the setup routine runs once at the start of the program:
void setup()
{
  pinMode(ledPin, OUTPUT);         // initialize the digital pin as an output.
}

// the loop routine runs over and over:
void loop()
{
  digitalWrite(ledPin, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                  // wait for a second
  digitalWrite(ledPin, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);                  // wait for a second
}
```

The first line declares a variable, `ledPin`, that will be used to tell the program which pin to use to control the LED. Capitalization matters, such that `ledPin` is not the same as `ledpin`. To be able to control the pin, we must first set it to be an `OUTPUT`, which typically happens in the `setup()` routine. When you write a program using the Arduino IDE, everything in the `setup()` routine will run *once* when you first start the program. Everything in the `loop()` routine will then run over and over, forever.

For example, in the `loop()` routine above, the first line turns the LED on using the `digitalWrite()` function. By writing `HIGH` to `ledPin`, the Arduino will make the pin output 5V and current will flow through the LED, turning it on. The code then waits 1000 *milliseconds* using the `delay()` command, after which it turns the LED off by setting the pin `LOW`, followed by another `delay()`. Because it's in the `loop()` routine, the code will be called over and over again. Lather, rinse, repeat.

A few more syntax notes for those that are curious:

- The double forward-leaning slashes `//` are used to denote comments. Anything after the double forward-leaning slahses are just comments that the computer does not read.
- The semicolon after each line tells the computer that it is at the end of a statement.
- The curly brackets before `{` and after `}` the `setup()` routine and the `loop()` routine tell the computer where the routines start and end.

Now you'll edit the code to change the length of time the LED is on and off.

Procedure

1. Edit the code so that the LED will remain on for 2 seconds and turn off for half a second.
2. Upload the code and see if your code works as expected.

Show your working system to an instructor.

Digital input: Reading a button

There are lots of things you can do with digital output: command motors to spin, make sounds, light LEDs, and a whole lot more. Of course, it's also useful to *input* signals into your microcontroller, for example, to read a button or (as you did in the sensor lab) a touch sensor. Here, you'll declare a pin as an input and use it to detect the state of a button. When the button is down, the LED will light; when the button is up, it'll turn off.

Figure ?? shows two buttons. You only need to build a circuit with one of those two buttons - your pick!!

Procedure

1. Disconnect the power to your Arduino and add one of the buttons as shown in the circuit shown in Figure **??**.
2. Open the `button` example code under Sketchbook → arduino.basic → button. Note the following:
   - The button pin is declared as an `INPUT`. This means it will respond to external signals that are applied to it.
   - When the button is not pressed, the pins of the button are not connected electrically. When it is pressed, the pins are connected electrically.
   - The button is connected to a $10k\Omega$ resistor, which is marked with black-brown-orange-gold stripes. It is being uses as a *pull-up resistor*, which guarantees that the button pin reads `HIGH` when the button is *not* pressed.
   - When pressed, the pin is connected electrically to ground so it will read `LOW`. This is an example of *inverted logic* because pressing the button makes the pin connected to the button `LOW`.

     You could set this up with a *pull-down resistor* if you put the resistor between ground and the button (instead of between 5V and the button), in which case an unpressed button would read `LOW` and a pressed button would read `HIGH`. We'll only have you set it up as a *pull-up resistor* during class today - but the concept is the same regardless. If you are interested in learning more (outside of class), you can pull up or pull down a great description here.
   - The pin is read using `digitalRead(<pin>)`, which reads the state of the specified pin and returns a `1` if the pin detects a `HIGH` voltage and `0` if it reads `LOW`.
   - The result is then tested in an `if..else` statement and the LED is set accordingly.
3. Make sure that `buttonPin` corresponds to the physical pin on the Arduino that you connected to your button.
4. Upload the code and press the button to verify that the behavior is correct.

5. Edit the code so that the behavior is reversed: the light comes on when the button is up and goes off when pressed. Upload and test your code.
6. Now edit the code so that the program waits for 2 seconds after the button is pressed before turning that LED off. What function will you use? Upload and test your code.

Show your working system to an instructor.

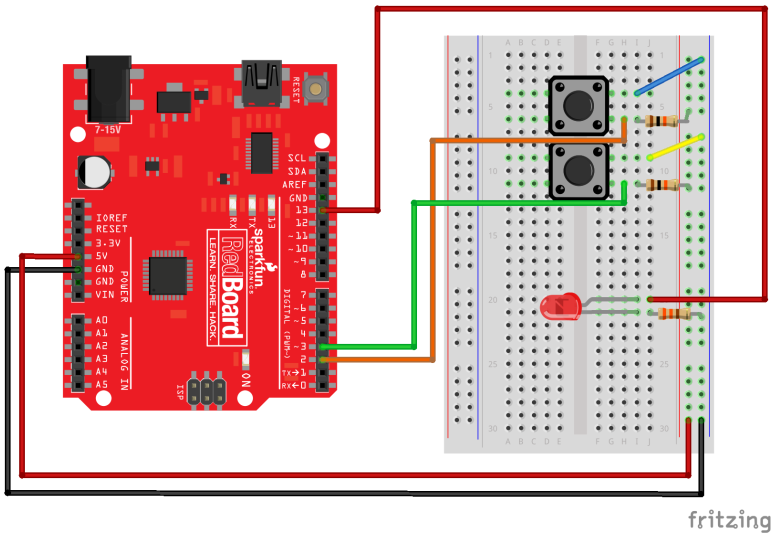

Figure 2: Fritzing diagram for the button breadboard circuit. You only need to build the circuit for one of the buttons.

Analog input and output

Digital I/O, as demonstrated above, is extremely useful. However, there are times when you want to produce or read a signal that isn't just `HIGH` or `LOW` – some things are in between! For example, if you were controlling a motor, you might want to change the *speed* of the motor, as opposed to just turning it on and off. Or perhaps you want to find the temperature, which is more than just "hot" or "cold", but somewhere in between. For those functions, you need to use the *analog* input and output functions.

Analog output: Dimming an LED

Here, you will change the brightness of the LED using the `analogWrite()` function. Just like `digitalWrite()`, the function takes a `pin` as an argument, but instead of `HIGH` (5 V) or `LOW` (0 V), you pass it a `value` between 0 - 255, where 0 is off and 255 is fully on. Contrary to what the name suggests, however, the pin doesn't actually vary the voltage of the pin directly. What it does is toggle the pin *really fast* in a way that the *average* voltage approximates an intermediate voltage. This apparent voltage is given by the formula:[2]

[2]If you're paying attention, you'll notice that this formula won't produce 0 V when you pass a 0 to analogWrite(). Arduino has special code to deal with this situation.

$$apparent\ voltage = \frac{value + 1}{256} \cdot 5V \qquad (1)$$

Under the hood, the Arduino changes the *duty cycle* – the amount of time that the pin is on for each on-off cycle – by varying the width of each 5 volt pulse. Figure **??** shows an example of three different duty cycles. Because of this, the method of varying the apparent voltage is called *pulse width modulation*, or just PWM. Note that the maximum value you can pass `analogWrite()` is 255 – values above that will have unexpected behavior.

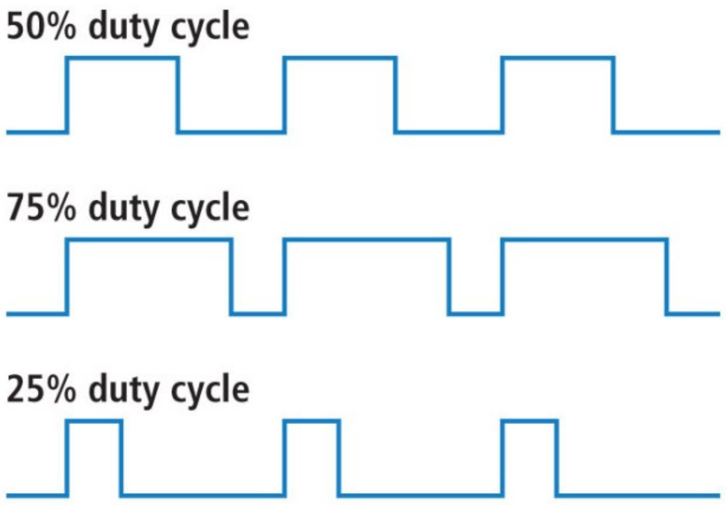

Figure 3: Three different PWM duty cycles. Notice that the period of each cycle remains the same, but the width of the pulse is varied.

Procedure

1. Navigate to File → Save As... and save the current file (which should be `button`) as `dimmer`. While there are better ways to do code management, you don't want to clobber a working code that you're going to change, since you might need to go back to it to troubleshoot.
2. Move the wire that powers the LED from pin 13 to pin 11. Only the pins that have a ˜ next to them can perform PWM (the mystery of the ˜ is revealed!).
3. Change the line of code near the top to indicate that the led is on pin 11.
4. Upload the code to be sure that everything still works as before. *It's generally a good idea to change one thing at a time and test it, as opposed to changing a bunch of things all at once.*
5. Now change the `digitalWrite()` statements to `analogWrite()`. When the button is pressed, have the code pass a PWM value of 150 to `analogWrite()`. Pass 75 when the button is up.
6. Upload the code. Press the button and notice what happens to the LED – it should vary in brightness, but still be lit in both cases.

Show your working system to an instructor.

Analog input: Reading a potentiometer

A potentiometer (which, in its rotary form as we use here, is a fancy name for a "volume knob from a radio") is a useful component when you want to have a variable control input. Unlike a button or a switch, which are either open or closed, the output of a potentiometer varies continuously. When connected as shown in Figure **??**, the voltage produced on the middle pin, called the *wiper*, will vary from 0 V to 5 V as the wiper moves back and forth along the resistive element.

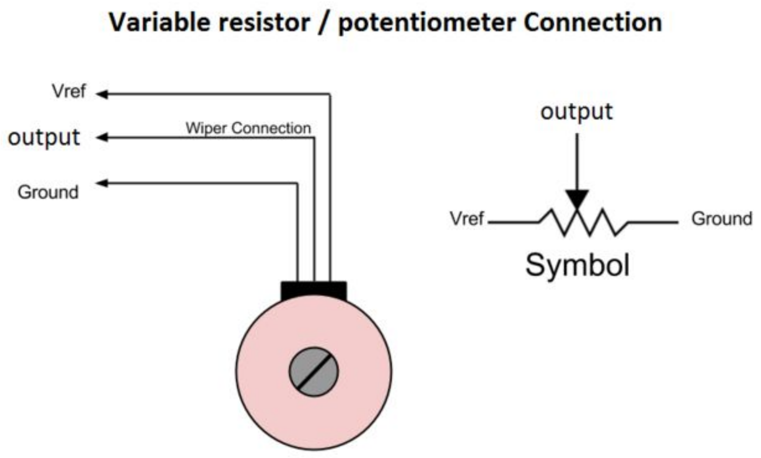

Figure 4: Physical representation and schematic symbol for a potentiometer. For an Arduino, $Vref$ is typically 5 V.

To read this voltage, you'll use something called an *analog-to-digital converter*, or simply an *ADC*, which converts a voltage into a value that can be used by the microprocessor. For an Arduino Uno, the ADC value is given by the formula:

$$ADC\ value = \frac{input\ voltage}{5\ V} \cdot 1024 \tag{2}$$

where the ADC value is rounded down to the nearest integer and constrained to be between 0 - 1023. For example, if the ADC were to sample 2 V, it would record a value of

$$ADC\ value = \frac{2\ V}{5\ V} \cdot 1024 = 409.6 \rightarrow 409 \tag{3}$$

Procedure

1. Unplug your Arduino and build the circuit shown in Figure **??**. The six analog pins, `A0..A5` are used for reading analog voltages using the function `analogRead(<pin>)`
2. Open the program `potentiometer` from the same sketchbook location as before.
3. Upload and run the program. As you turn the potentiometer, the LED should grow brighter or dimmer, depending on which way you turn it.

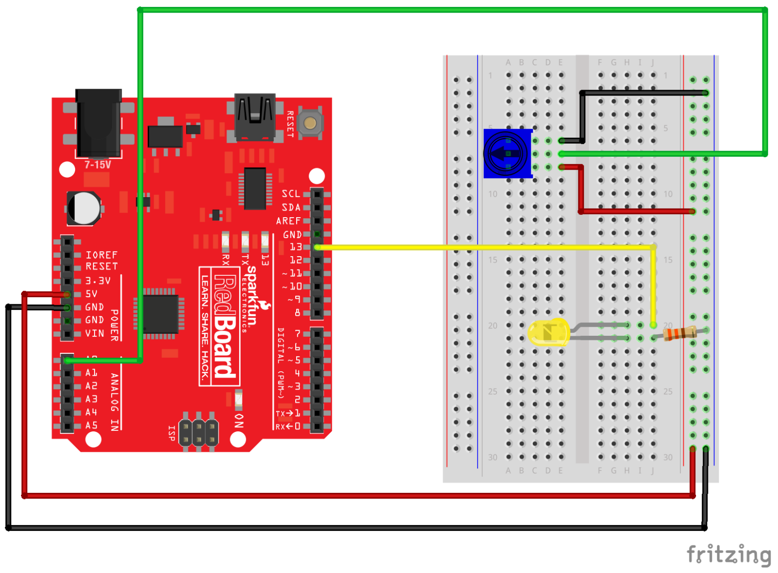

Figure 5: Fritzing diagram for adding a potentiometer. Be sure to move the LED wire to pin 11.

Note that the result of `analogRead()` is divided by 4 before the value is sent to the LED pin. While `analogRead()` returns a value from 0 - 1023, as noted above, `analogWrite()` only takes arguments from 0 - 255.

Analog input: Reading a temperature sensor

Here you will determine the temperature in the room using a temperature sensor that outputs a voltage that varies depending on the temperature. Specifically, you will use the `TMP36` temperature sensor, which outputs a voltage, V, according to the formula:

$$V = 0.5 + 0.01 \cdot T \tag{4}$$

where T is the temperature in Celsius and V is in volts.

In this program, you'll make use of the Serial Monitor, which is a convenient way to send information from your Arduino to your computer. To use the Serial Monitor, you must call

```
Serial.begin(<baud rate>);
```

in your `setup()` routine, where `baud rate` is the speed that the Arduino will send data, in bits per second. Common choices are 9600 (the default) and 115200 (high speed). For these programs, it doesn't matter which you use, *just be sure to choose the same baud rate in the Serial Monitor when you open it*, otherwise you'll get gibberish. You write to the Serial Monitor using

```
Serial.print(<data>);
```

and

```
Serial.println(<data>);
```

where the second version will send a newline character after it sends the data, causing the output to scroll. `data` can be text or numbers, where the former must be enclosed in quotes, e.g.:

```
float temperatureA = 22.5;
float temperatureB = 30.1;

Serial.print("Temperature A = ");
Serial.println(temperatureA);
Serial.print("Temperature B = ");
Serial.println(temperatureB);
```

would output:

```
Temperature A = 22.5
Temperature B = 30.1
```

The `float` denotes that a variable can have decimal places – in computer-speak, it's referred to as a *floating-point* number.

Procedure

1. Unplug your Arduino and add the temperature sensor as shown in Figure **??**, *except connect the middle leg of the sensor to pin* `A1` (not pin `A0` as shown in the figure). Leave the potentiometer in place for a later tutorial. **NOTE**: The TMP 36 has a flat side and a rounded side on the black plastic housing. The flat side is facing to the right in Figure **??**.
2. Open the program `tmp36`.
3. Upload and run the program.
4. Start the Serial Monitor by clicking the magnifying glass in the upper-right of the IDE. The Serial Monitor will print out the value read by the ADC every second.
5. Using the formulas above, figure out the formulas to convert first from ADC value to voltage and then voltage to temperature.
6. Edit the lines of code that say `float voltage = 0;` and `float temperature = 0;` to compute the voltage and temperature using the formulas you just found. **Word of warning**: When coding the equations, use decimal points, e.g., `adcValue / 1024.0`. This will tell the compiler to use floating-point math – otherwise, the program will "round" all the numbers and you'll end up with crazy results.
7. Add code to print the voltage and the temperature to the Serial Monitor, in addition to the raw ADC value that is already printed. Print each set of readings on one line, separated by a space. Add a delay of 500 ms after you print so that you don't overwhelm the Serial Monitor.
8. Upload the program and verify that your code works.

Show your working system to an instructor.

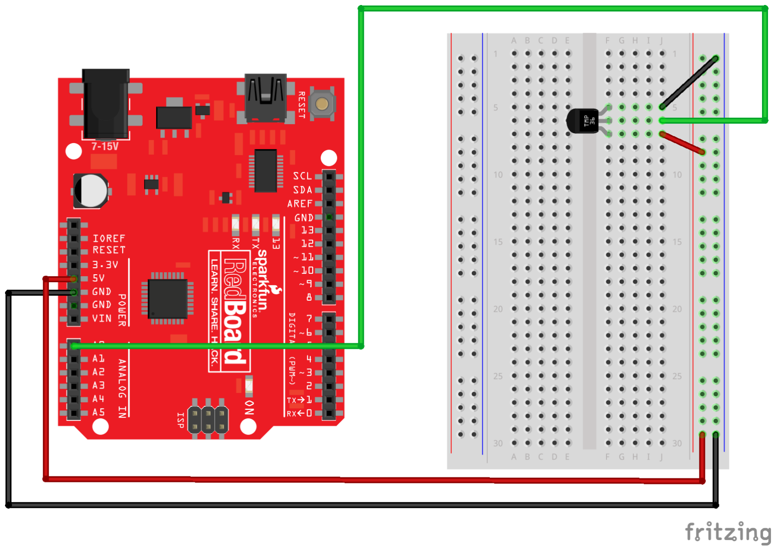

Figure 6: Fritzing diagram for adding the TMP36 tempearture sensor. NOTE: the TMP 36 has a flat side and a rounded side on the black plastic housing. The flat side is facing to the right in

Using libraries: Controlling an RC servo motor

Kind people have written many libraries for Arduino to accomplish many common tasks: communicating with certain chips, connecting with WiFi or ethernet, running timers, and many, many more. The advantages of using existing libraries are obvious: no need to recreate code and flexibility topping the list. Be aware, however, that there are some disadvantages, as well; the most common one is that some libraries don't "play nice" with others, which can lead to headaches. Here, we won't concentrate on the pitfalls, but show how to implement and use a library, in this case the `Servo` library. You'll make a "dial thermometer" that shows the temperature by moving an RC servo motor.[3]

Procedure

1. Disconnect your Arduino from your computer and build the circuit in Figure **??**.
2. Open the example file `knob`. This program reads the position of the potentiometer using the `analogRead()` command and moves the servo in response to changes.
3. Near the bottom, there's a `delay` statement. Depending on when you downloaded the code, this may be 15 ms or 200 ms. If your system doesn't work well at first, the first thing you should try is making the delay longer – as much as 400 ms.
4. Connect the Arduino to your computer and upload the program. If all went well, you should be able to make the servo sweep back and forth in response to the turning of the potentiometer.

Before continuing, let's explain what's going on. There is some administrative work setting up the servo in `setup()`. In the `loop()`, the program reads the position of the potentiometer using

[3]It's referred to as an "RC servo" motor because this variety was made popular in remote control vehicle applications.

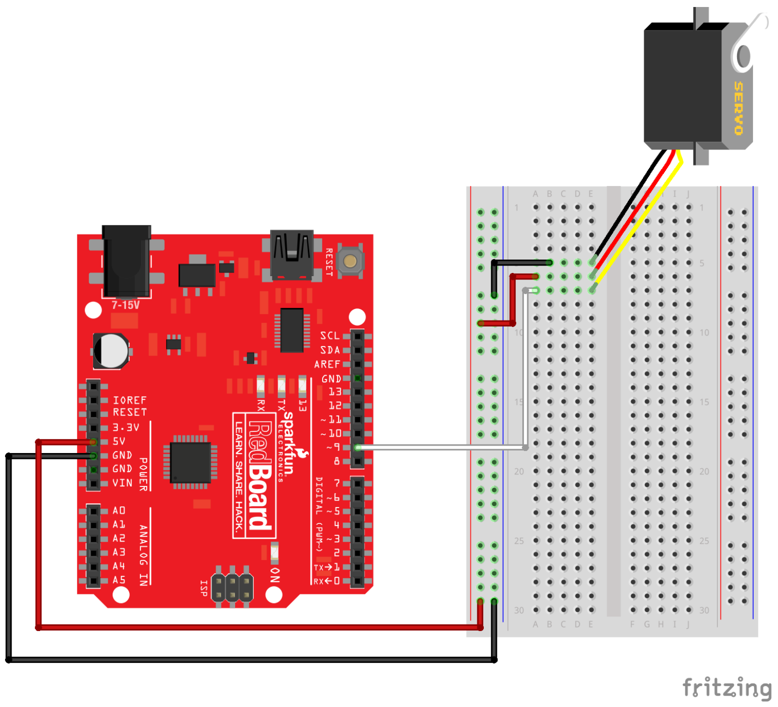

Figure 7: Fritzing diagram for adding the RC servo motor. The potentiometer should already be there from earlier.

`analogRead()`. This value – what is the range again? – is scaled to a value of 0 - 180 using `map()`, a convenient helper function, and the new value is used to command the servo using the `Servo.write()` function, which takes an *angle* (in degrees from 0 to 180) as an argument.

The Servo library has many useful functions. Let's look at some details:

- Near the top of the code is a line that reads

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

  which tells the Arduino compiler to make use of the library. Other libraries are "`#include`-d" similarly.

- The command, `Servo.attach(servoPin);` tells the Arduino which pin to use to talk to the servo motor (in this case, we set `servoPin = 9` near the top of the program. A similar function, `Servo.detach()` disconnects the servo.

- One issue that is glossed over is the amount of power to drive the servo motor. Driving the servo from just the USB cable probably isn't going to give the servo enough power. If your servo jumps around a lot, you might want to use a separate power supply – just ask and we'll find one for you. It helps to be a little cynical when it comes to Arduino tutorials – easy is nice, but often there is more to the system than meets the eye. If you end up using a servo in your project, talk to us and we'll help you get set up with a separate power source for the servo.

Now, you'll make your servo motor react to changes in temperature.

Procedure

1. Start a new file under File → New. Call it `thermometer`.
2. Add/edit code (you're welcome to cut-and-paste from the previous tutorials) so that your code has the following functionality:
   - It reads the temperature sensor and calculates the temperature in Celsius,
   - It maps a temperature range of 20 - 35 C to an angle of 0 - 180,
   - It commands the servo motor to move to that angle,
   - It prints the raw ADC value, the temperature in C, and the position of the servo in degrees to the Serial Monitor. Print each record on one line, separated by spaces.
3. Upload and run your program. Place your fingers on the temperature sensor to show that warming the sensor will cause the servo to move in the expected way.

Show your working system to an instructor.