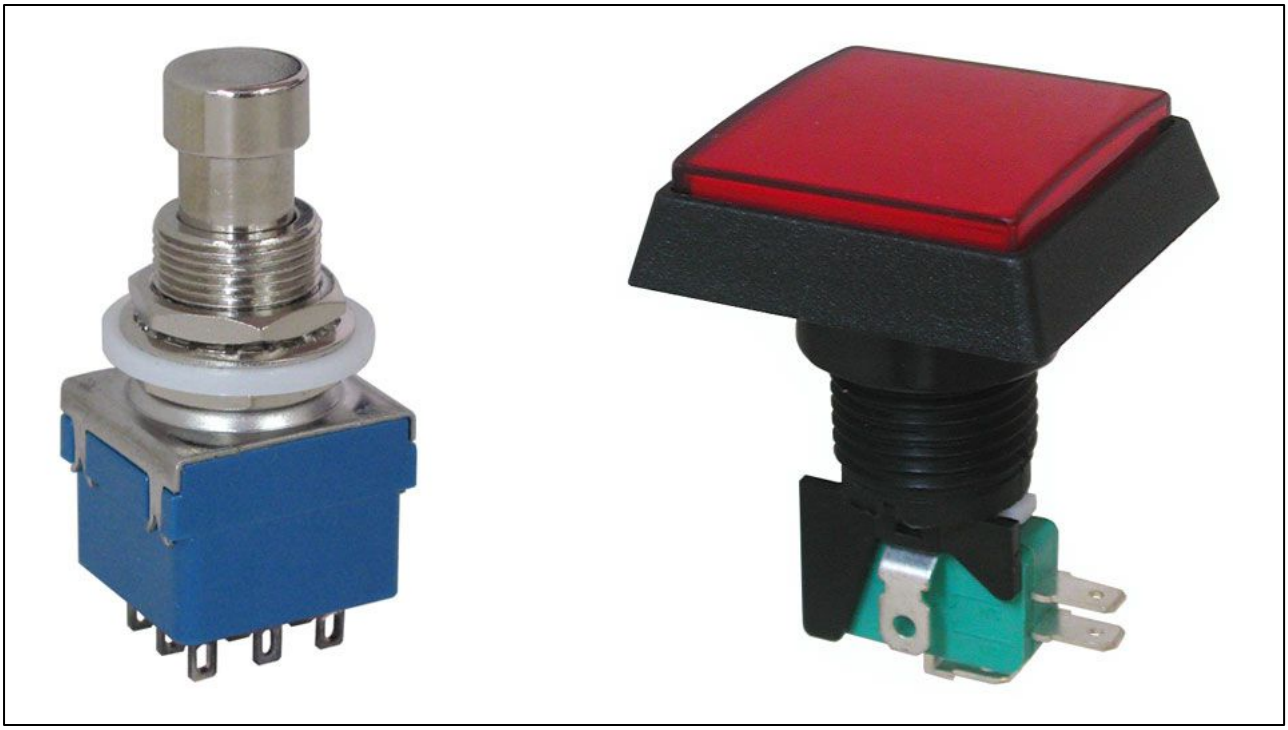
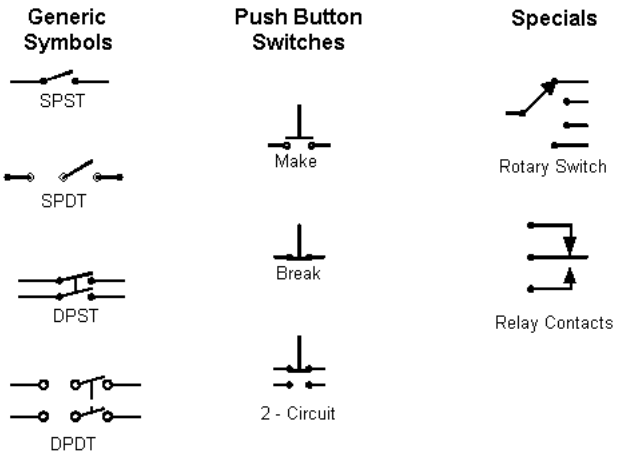
Digital Input

Digital input is all or nothing, true or false. There are no shades of gray here. Either a pin detects it is in an ON state, or it is in an OFF state. This can be a very useful way for an Arduino to sense the world.



Let’s start with the simplest form of input: a button. There are lots of cool options here. The switch on the left can be pressed with your foot. The one on the right lights up and is pleasingly enormous. Timball has some more switches handy, I think. However complicated they look, they are really pretty simple on the inside. Wiring up a big awesome switch in place of a puny one is an easy, low-risk electronics project (provided it isn’t a power switch, in which case you should be careful!).



We're going to use a very humble kind of switch: a pushbutton. These are very cheap and handy (there's one built into the Arduino!). Ours have four legs, which you should think of as two pairs. Each pair is electrically unified. When the button is pressed, the two pairs are connected to each other for as long as the button is held down.

These pushbuttons are what's known as an SPST switch: **single pole/single throw**. Either of those S's can become a D, for double.

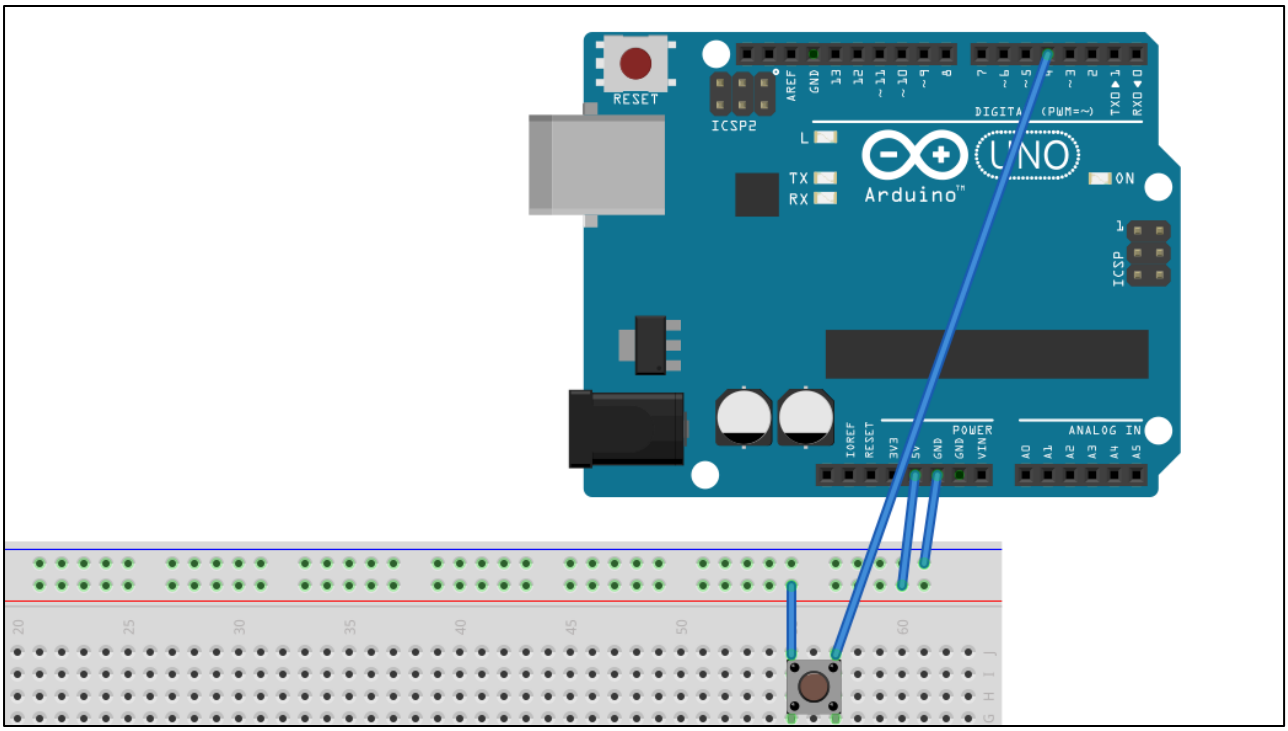
Pole refers to the number of electrically distinct switch circuits in the module. Think of it like the north and south pole -- they are as separate as can be.

Throw refers to the number of paths by which each circuit can allow electricity to exit. Think of it as the way the switch *throws* energy out into the world. In an SPST normally-open switch, when the button is not pressed, the input energy doesn't go anywhere. When it is pressed, it goes out the second set of contacts. By contrast, in a SPDT switch, when the button is not pressed energy flows out of one set of contacts, and flows out of a different set of contacts when it is pressed.

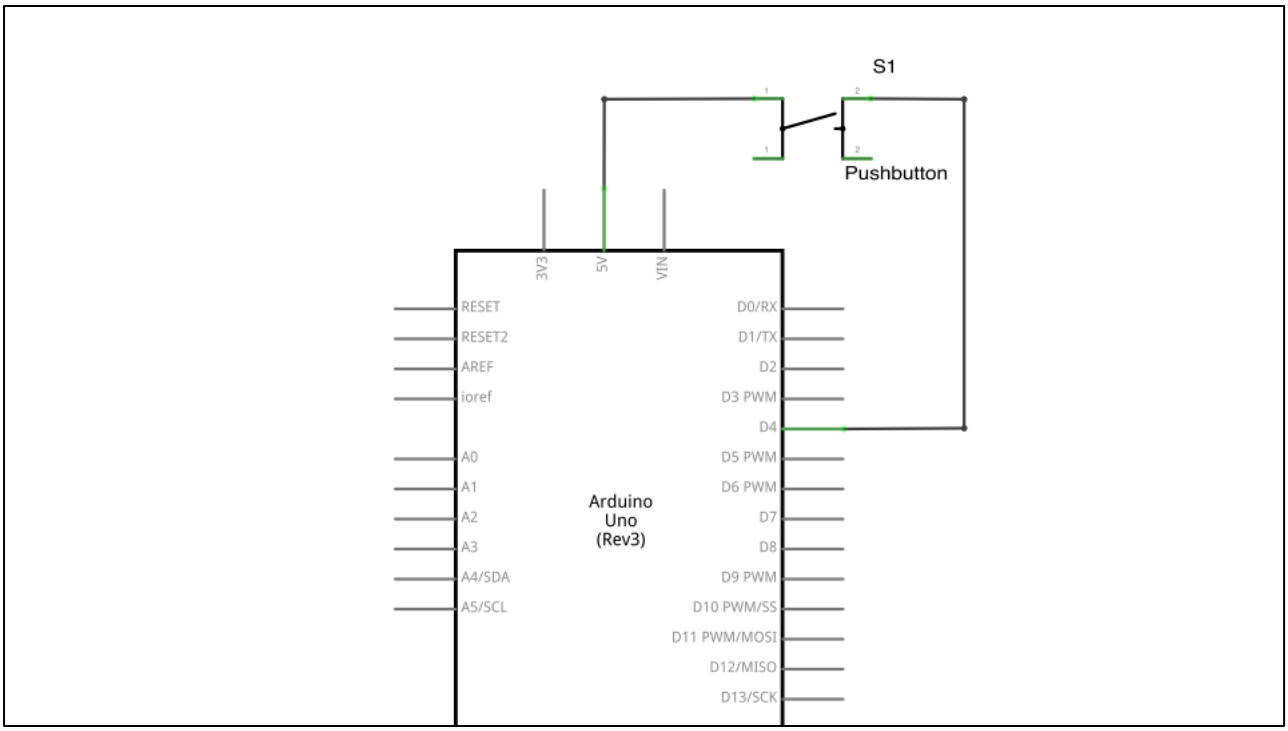
Pushbuttons are also **momentary** switches: they're only closed for as long as the button is pressed. Obviously some other kinds of switches, like light switches, behave differently.

In practice you can usually substitute one type of switch for another if you have some creativity or extra switches. Don't worry about these distinctions too much -- I'm

mentioning them just so you can go “ohhhh yeeaaaaahhhh” if you ever find yourself switch-shopping.



So how do you wire up a switch? Let's start by looking at the most obvious case and explaining why it's not ideal. Here we connect one of the switch terminals to 5 volts/Vcc, and the other to pin 4, which we will use to measure voltage. Go ahead and wire this up on your breadboard. If you still have your RGB LED in place, leave it there.



Here is the same circuit displayed as a schematic. I'm showing this mainly to get you used to looking at these, because it's how circuits are usually represented.

Note that the Arduino doesn't look very Arduino-like. And in fact, the pins aren't spatially organized in the same way that they exist in real life. This is very common, I' m afraid.

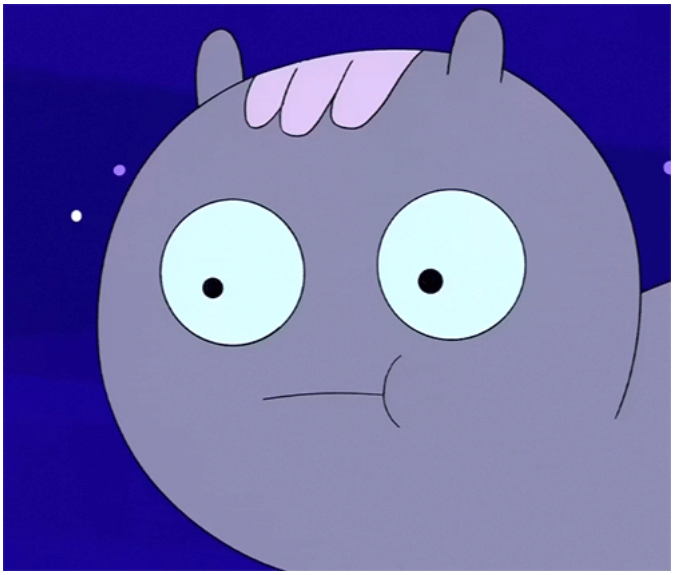
```
int buttonPin = 4;
int ledPin = 13;

void setup() {
  // set pin 4 for input, pin 13 for output
  pinMode(buttonPin, INPUT);
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // read input
  int reading = digitalRead(buttonPin);

  // turn LED on or off, depending
  if (reading==HIGH){
    digitalWrite(ledPin, HIGH);
  }
  else {
    digitalWrite(ledPin, LOW);
  }
}
```

Here's some very simple code that initializes pin 4 to INPUT mode and our old friend, LED-enabled pin 13, to OUTPUT mode. Then in the loop, the value of pin 4 is checked. digitalRead() will return HIGH if the voltage on pin 4 is at or above 3 volts. It will return LOW if it is at or below 2 volts.

Depending on the value, we then turn pin 13 (and its LED) on or off, just like we did last week.

You can download this code from https://github.com/sbma44/arduino-class/blob/master/lesson-2/simple_digital_input.ino rather than typing it in, if you'd like.

pinMode(), INPUT and OUTPUT



So what does INPUT mode mean, anyway? Well, it adjusts the *impedance* of the pin -- basically, how big the resistor attached to it is. In OUTPUT mode, the impedance is made very low so that a (comparatively) lot of current can flow into or out of the pin. In INPUT mode, the impedance is made high, so that just the barest trickly of current can pass -- but the voltage can still be observed.

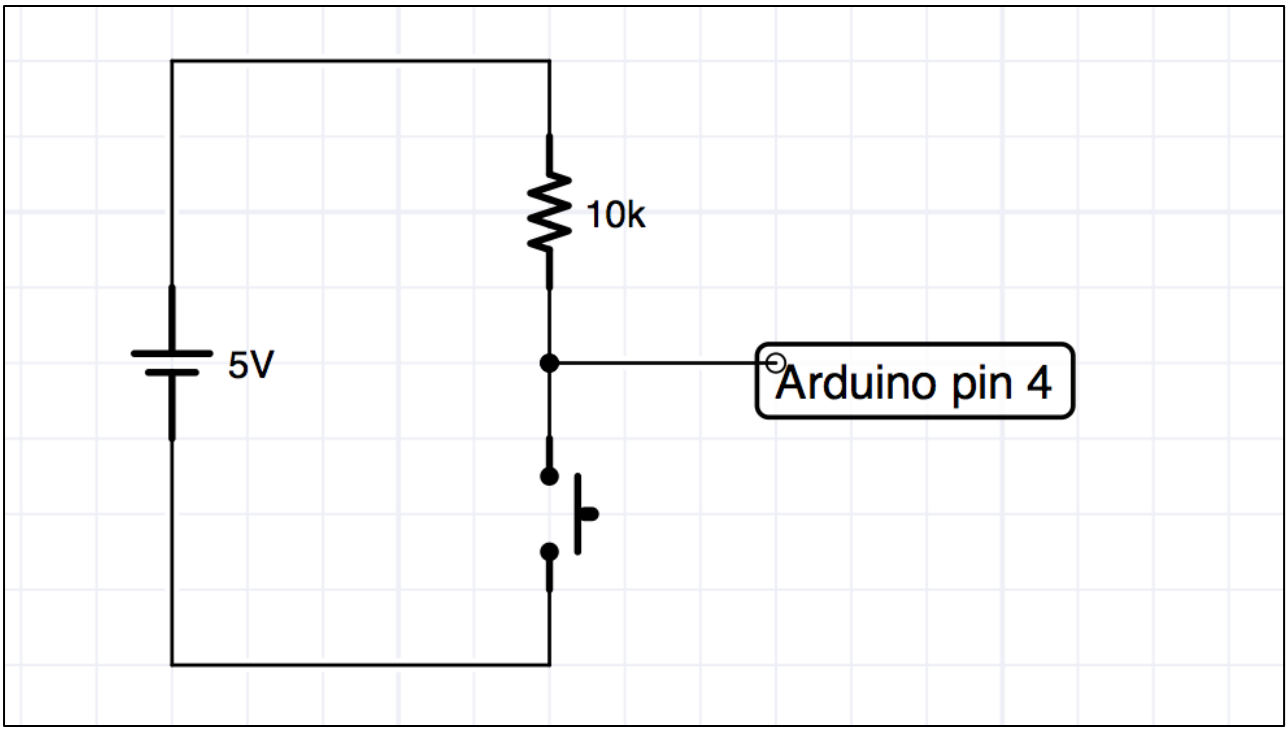
(This horse's eyes were the most compelling GIF for "dilation" that I could find. Think of how much light the pupils let through as being analogous to how much energy a resistor lets through -- high impedence is when the pupil is small, low impedence is when the opening is wide.)

FLOATING IS CONFUSING

So why is the code no good? Because of what happens when the button *isn't* pressed. When it is pressed and the circuit is closed, pin 4 is connected to Vcc and will read HIGH thanks to the incoming 5 volts.

But when it isn't pressed, it isn't connected to anything -- neither Vcc nor ground. So what value will it be at? It turns out that the impedance can be so high that it's impossible to say. It can carry a static electric charge, so that it stays high enough to be HIGH. Or it can fade away, pulling it LOW. This is the kind of thing that can change just by a cell phone or other wires or cosmic rays or even your statically-charged finger being nearby! It's very hard to debug.

So we try to avoid floating pins. We want to be sure that everything we're observing is always connected to *something*.



But how can you be connected to both Vcc and ground without causing a short circuit? The answer is actually pretty simple: we use a resistor.

This configuration is what's known as a *pull-up resistor.* It pulls the value of pin 4 up to Vcc (5 volts) when the switch is open. But when the switch is closed, pin 4 is connected to ground and, because there's so much less resistance in that direction, pin 4 will go to zero volts.

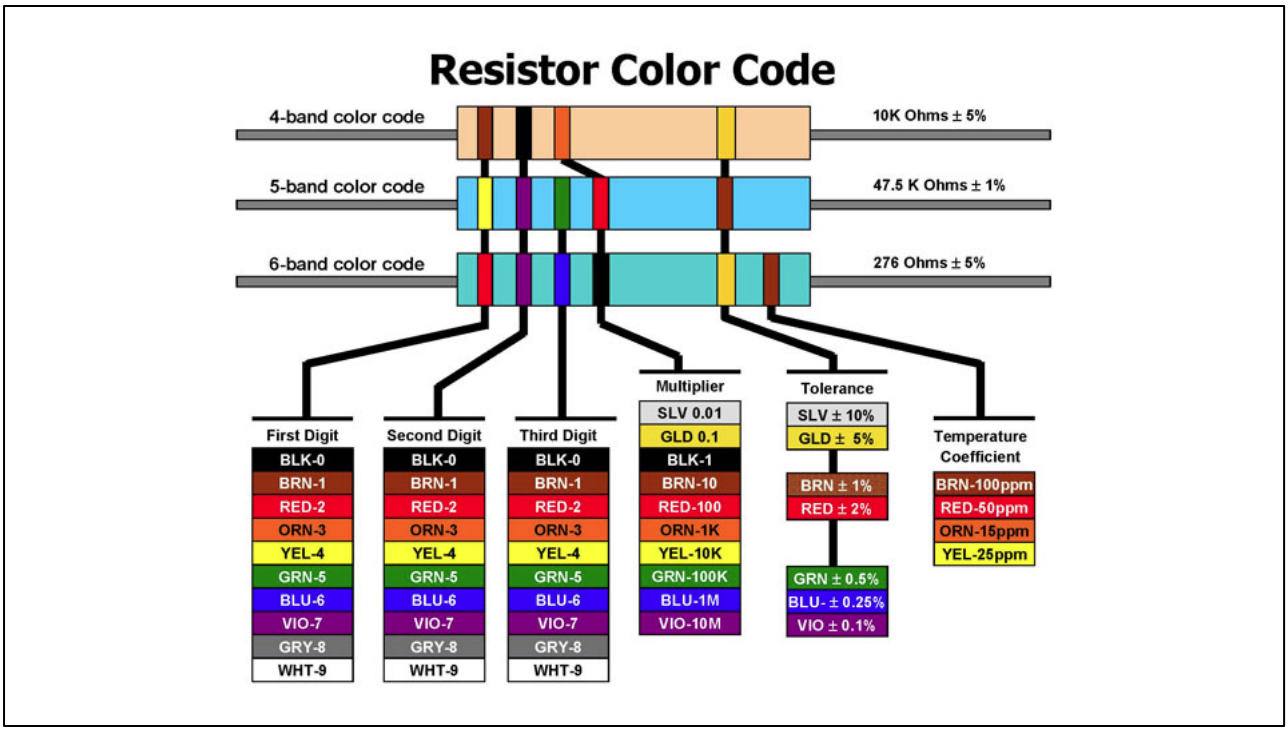
You can have *pull-down resistors*, too, of course (in this diagram, you would switch the position of the resistor and the switch). They just work in the opposite way, keeping the pin normally at 0V, and bottling up the electricity from escaping to ground too quickly when the switch is closed and it floods into the pin.

Note the value of the resistor in this diagram -- it's 10,000 ohms (often abbreviated 10k). This is *way* more than the 150 ohm resistors we're using. This is for efficiency's sake: you can use a very high resistance if you just need to sense voltage. We picked these resistors because they let through the right amount of current to make LEDs light up. It will be inefficient to use these for pull-up resistors -- they'll be letting through a lot more juice than is necessary -- but our computers' USB buses can handle it.

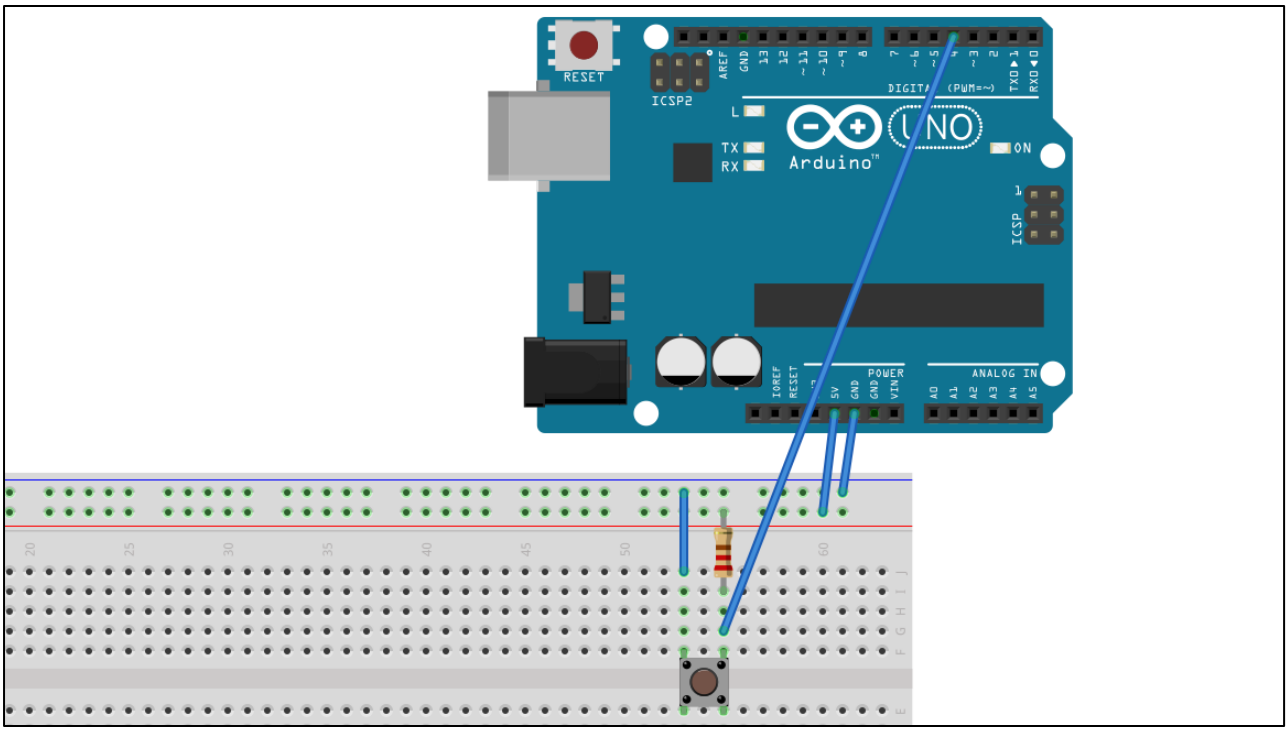
To figure out how much current they'll be wasting, why not try http://www.ohmslawcalculator.com/ohms_law_calculator.php ? You should just enter the voltage and resistance, then click "Calculate" (you could also just do this on paper -- it's about

the simplest algebra problem you could imagine). Try it with the theoretical 10K resistor and with the resistor value we actually have, and compare the number of watts that will be going through. By way of comparison, a AA battery contains about 2250 milliwatts of power. Even with this wasteful configuration, this isn't a huge deal, since the large current draw will only happen when the switch is closed. If we were using a different kind of switch that stayed closed after activation it would matter more.

Try making this circuit on your breadboard. This is a great chance to practice converting circuit schematics into breadboard layouts!



(This seems like a good time to remind you how to read resistor values. In practice, I just use a multimeter because it's wayyyy easier. But you probably don't own one yet, so here you go!)



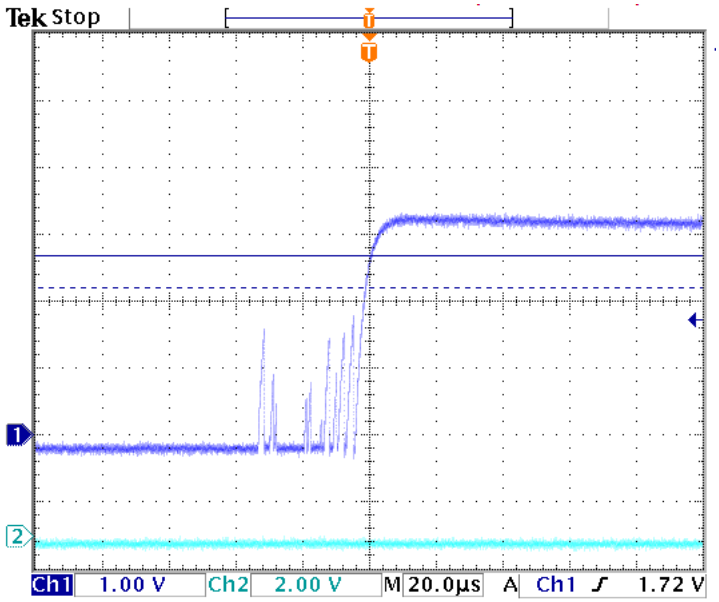
OK, if you had trouble, this is what it should look like. Run it with the code from slide 6!

https://github.com/sbma44/arduino-class/blob/master/lesson-2/simple_digital_input_modes.ino

So far our program is pretty pointless: we can turn an LED on and off with ten cents' worth of LEDs, pushbuttons and wires. No need for a microcontroller!

So let's do something more ambitious. This program expects your RGB LED to be hooked up, like at the end of last lesson. Each time you press the switch, it will advance through three modes, looping back to the first after the third. In each mode it turns one of three LED pins to LOW and the other two to HIGH. Try it!

Debouncing

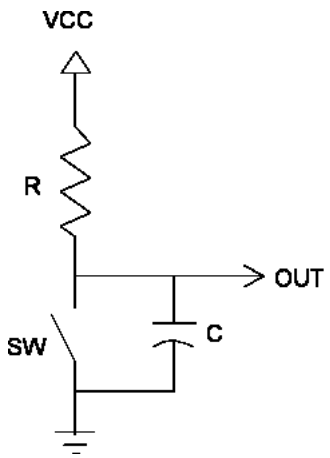


Some of you might be noticing that the modes don't switch as evenly as you expected. That's not unexpected! It's because we failed to *debounce* the switch.

Switch bounce happens because of tiny imperfections within the switch. As the two conducting plates approach each other, individual atoms of the uneven surface make and break contact, leading to a less-than-perfectly-clean transition from one state to another. This is a slide of an oscilloscope graph of this happening.

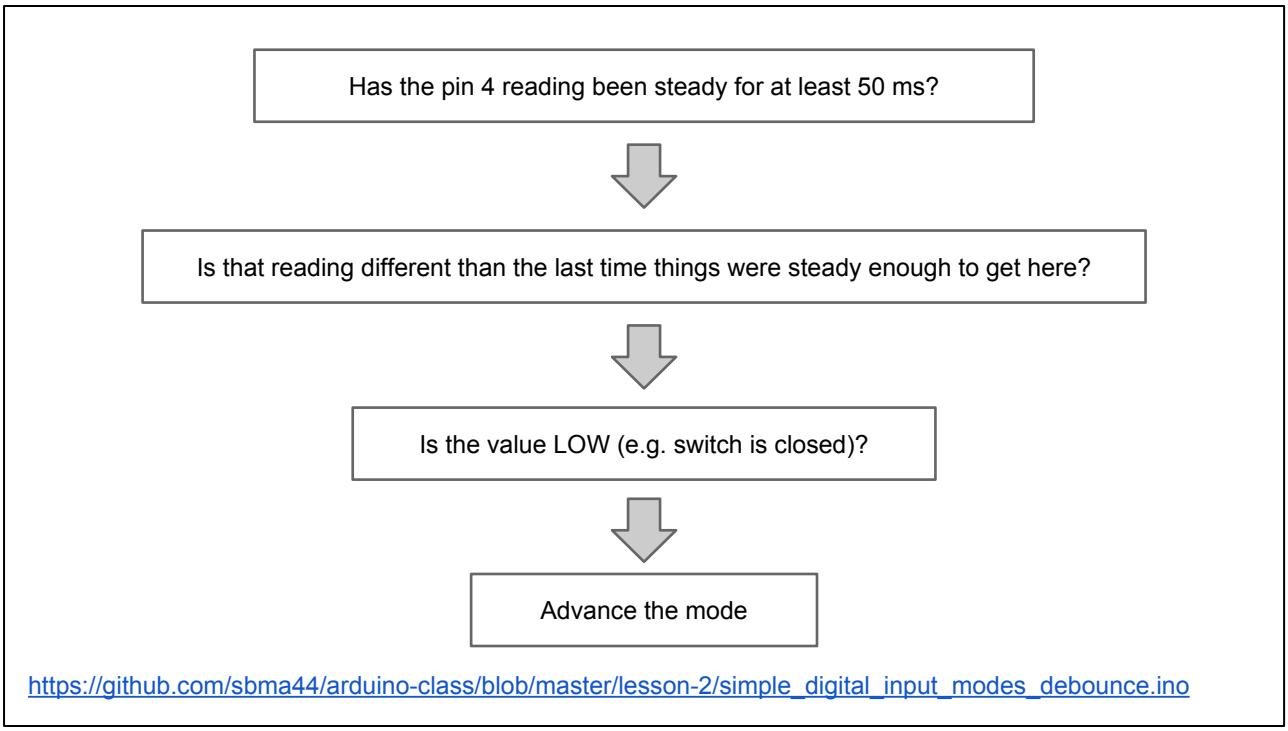
Depending on the specifics, the Arduino can notice this -- it's so fast that it it observes the state of the circuit several times, even as you're making what seems like an instantaneous action! It's nuts.

Hardware Debouncing



We can “debounce” this scenario in a couple of ways. The most elegant method uses hardware, by adding a capacitor. You can think of capacitors as energy storage tanks -- they’re like rechargeable batteries, but they store much much less energy and discharge and recharge that energy millions of times faster.

Here, the capacitor gets charged up by the current coming in via the pull-up resistor. When the switch is closed, energy starts to drain out of the part of the circuit below *R*. But the capacitor *C* has to empty out before everything goes to ground and the Arduino (connected to *OUT*) can feel the difference in the voltage level. If the capacitor is sized correctly, the bounces will be smoothed out by this. I’m sure there’s a formula for figuring out the ideal capacitor size, but in practice I would just use trial and error (there is a decent amount of fudge room with most component values, and this is especially true for caps).



https://github.com/sbma44/arduino-class/blob/master/lesson-2/simple_digital_input_modes_debounce.ino

But capacitors cost money, and software doesn't. So most debouncing is now done in software, and that's what we're doing to do, too.

Here's the code adapted to include debounce code. Let's walk through it together -- it's a little confusing. But the chart above gets at the heart of what it's trying to do.

The basic idea is to take multiple readings over some span of time, and to only count the switch as closed if it seems to be in a steady state. This introduces delay, but by human standards the button's responsiveness will still seem pretty snappy. Here, we'll be waiting for the reading to remain steady for 50 milliseconds, or 1/20th of a second. You can adjust this in code to improve responsiveness at the cost of lessening your system's resistance to bounciness.

Assignments! Pick any of these.

- Change the order/values of the color pattern
- Add more steps to the color pattern
- Make the color pattern auto-advance, and turn the button into “pause”
- Add a “reverse” button
- Make the sequence about blink rate, not color

That covers input and output. Now it’s time to improvise! Here are a few ideas for how you could expand your circuit and code. Pick one, or come up with something yourself. I’d like to be sure that you can manipulate these tools, not just download and run them. Don’t be shy about asking for help if you get stuck.