HackerBox Basics Workshop

By HackerBoxes in CircuitsElectronics

Introduction: HackerBox Basics Workshop

The HackerBox Basics Workshop provides an enlightening introduction to electronics suitable for ages 10-110. No soldering required. The electronic components and modules were carefully selected to work along with the included solderless breadboard using jumper wire connections. The HackerBox Basics Workshop is perfect for self-study or classroom use in schools, business, scouts, makerspaces, or other training scenarios. Accordingly, the workshop is available for purchase here in single units or class packs of ten.

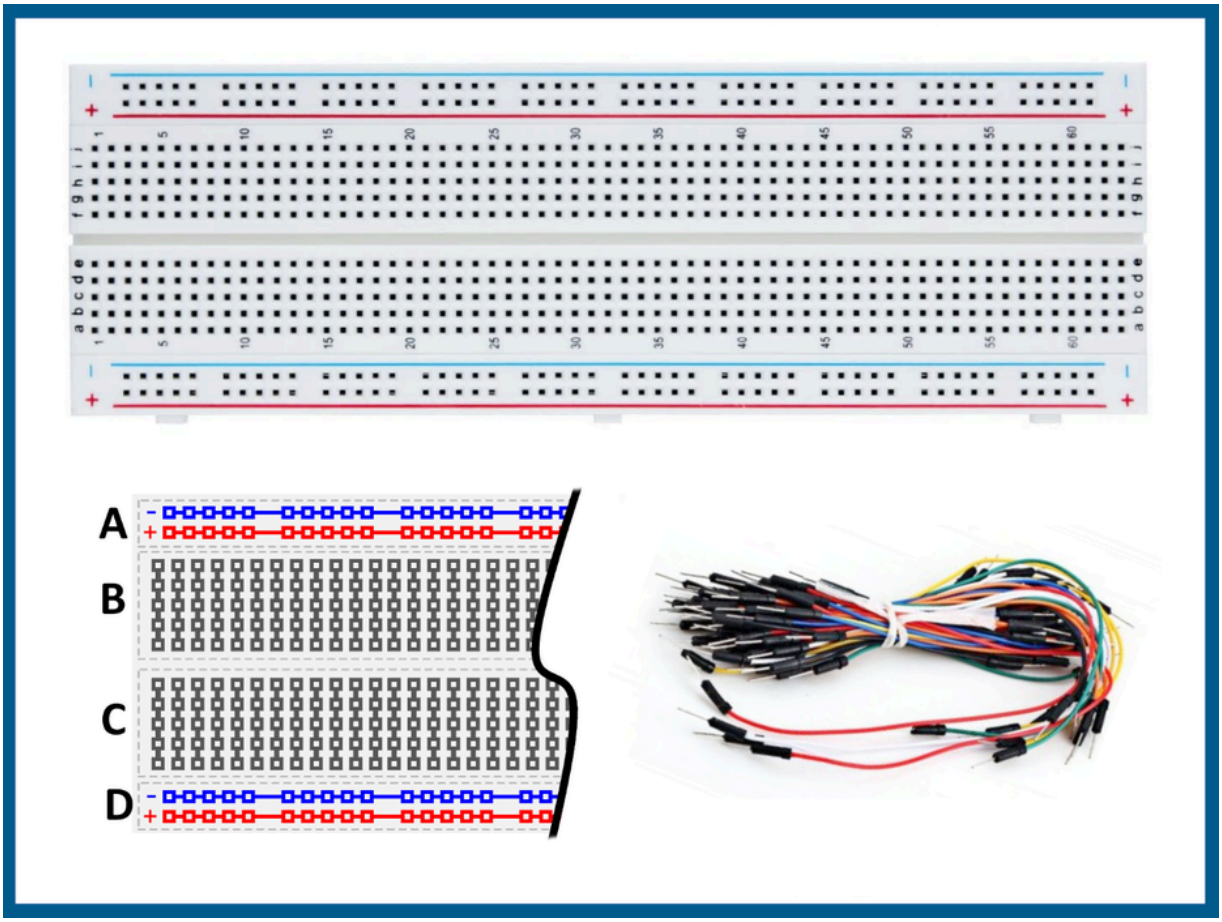
Supplies

This material covers introductory electronics in fifty topics and hands-on experiments. This journey, which we call *Electrons to A.I.*, starts with fundamental electricity, visits semiconductor transistors, digital logic, data storage, sensors, controllers, computer programs, and arrives at embedded computing devices capable of machine learning and artificial intelligence. The *Electrons to A.I.* educational programming spans the following subject matter:

1. Solderless Breadboards
2. Electron Flow
3. Control Electron Flow with a Switch
4. Control Electron Flow with a Pushbutton
5. Microcontrollers
6. Set Up The Arduino Nano
7. Control Electron Flow with Program Code
8. Looping and Timers
9. Program Output to Serial Monitor
10. Program Input from a Pushbutton
11. Digital Versus Analog
12. Analog Input from a Potentiometer
13. Measuring Voltage
14. Voltage Dividers
15. Resistor Structures
16. Ohm's Law
17. Adjusting Light Brightness
18. Light Sensors
19. Temperature Sensors
20. Program Control Flow
21. Storing Data in Arrays
22. Generating Sound
23. Measure Distance
24. Electromechanical Motion
25. Controlling Servo Motors
26. Displaying Graphics and Text
27. Full Color LEDs
28. Serial Addressable LEDs
29. Measuring Capacitance
30. Capacitor Structures
31. Electron Flow through Diodes
32. Transistors
33. Transistors as Switches
34. Digital Logic
35. Logic Gates from Transistors
36. Integrated Logic Chips
37. XOR Gates from NAND Gates

38. Combining Logic Gates
39. Storing Information
40. NAND Gate Flip-Flops
41. D Flip-Flop Integrated Circuit
42. Binary Counter
43. Computer Architecture
44. Assembly Language and Machine Code
45. Instruction Cycle
46. Algorithms and Heuristics
47. Machine Learning
48. Artificial Neural Networks
49. Embedded Neural Networks
50. Artificial Intelligence

Step 1: Solderless Breadboards



Solderless breadboards are the fastest and easiest way to prototype and experiment with electronic circuits and systems.

The drawing illustrates the four areas of the typical solderless breadboard.

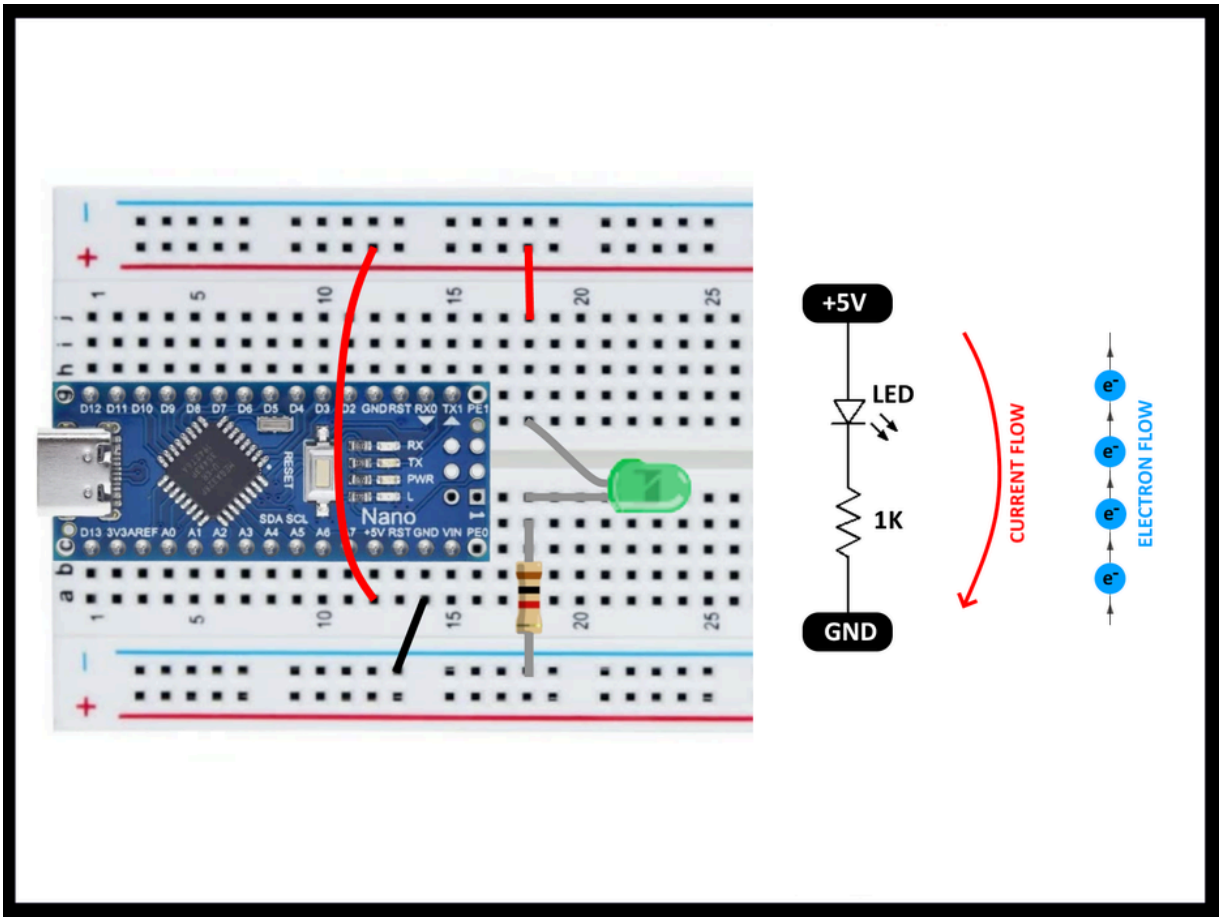
Areas A and D are "power rails" generally used to conduct the plus and minus voltages for your power supply across the entire length of the board. For example, the blue lines might be grounded (ZERO VOLTS) and the red lines might be connected to the +5V power supply line.

Areas B and C are "terminal strips" generally used to connect the pins of various circuit components. Each vertical column of five pins are connected together in the same way that the long horizontal rows of the power rails are connected together. Note that Area B is separated from Area C by a gap of exactly the distance between the two rows of pins on a standard DIP Chip. Accordingly, the terminal strips of Area B and those of Area C are not connected to one another.

Jumper Wires having pins or stripped wires on each end may be used to connect between the terminal strips as needed. The jumpers may also be used to connect the power rails into the terminal strip areas where needed.

For additional background on solderless breadboards, have a look at this tutorial from Sparkfun.

Step 2: Electron Flow



The Arduino Nano module is a microcontroller device with a USB-C connector. There is a lot of interesting circuitry on the Arduino Nano, which we will get to in due time. For now, we are only using the Arduino Nano as a mechanism for connecting 5V power from USB to our breadboard.

Once the Arduino Nano is inserted into the solderless breadboard as shown, the white USB-C to USB-C cable can be connected to the Arduino Nano. The other end of the cable can be connected to your PC (or USB hub) assuming there is an open USB-C port. If instead there is only a USB-A port available, there is a black/silver USB-C to USB-A adapter inside the plastic box of components.

Once power, a tiny red LED on the Arduino Nano will flash. We can ignore that for now.

Disconnect the USB power cable while assembling the circuit shown.

Let's examine the components and connections of the circuit...

The red "+" power rail across the top of the breadboard is connected to the +5V pin of the Arduino Nano by the longer red jumper wire.

The blue "-" power rail across the bottoms of the breadboard is connected to a GND pin of the Arduino Nano by the short black jumper wire.

The shorter red jumper wire connects the +5V power rail to the green LED. Note that the LED has a long pin and a short pin. The long pin must connect to the +5V power rail.

The short pin of the green led is connected to a 1K Ohm resistor.

The other end of the resistor is connected to the GND power rail.

These connections implement the circuit shown in the schematic to the right of the solderless breadboard.

One pin of the green LED is connected to the +5V power rail. The other pin of the green LED is connected (via the 1K resistor) to the GND power rail. Accordingly, there will be a potential difference between +5V rail and the GND rail across the LED. This difference will attract electrons from the GND rail to the +5V rail. Opposites attract, so the negative electrons (all electrons are negatively charged) are pulled towards the +5V direction.

Why exactly are we talking about electrons? Atoms make up everything and those atoms have electrons floating around them. Some of those electrons can be made to move. The electrons move more easily from metal atoms than from atoms of insulating material. Thus metals can conduct electricity (electrons). What that means is when a voltage (also called a potential difference) is applied across a conductor (like metal wire), some electrons in the conductor are drawn from the negative side of the voltage to the positive side of the voltage.

Electrons get sucked from the ground power rail, through the green LED, and towards the +5V rail. When the electrons flow through the LED, it glows with light.

Current Affairs

It is worth committing to memory that while electric current is the flow of electrons, the convention for specifying the direction of current flow is in the opposite direction of the flow of the electrons. Electrons flow negative to positive but "Conventional Current" flows positive to negative. Just accept it, or read under the heading "conventions" on the Wikipedia page for electric current to learn more.

Identifying Resistors

The Basics Workshop includes 1K and 10K resistors. How can we tell them apart?

The 1K resistors can be:

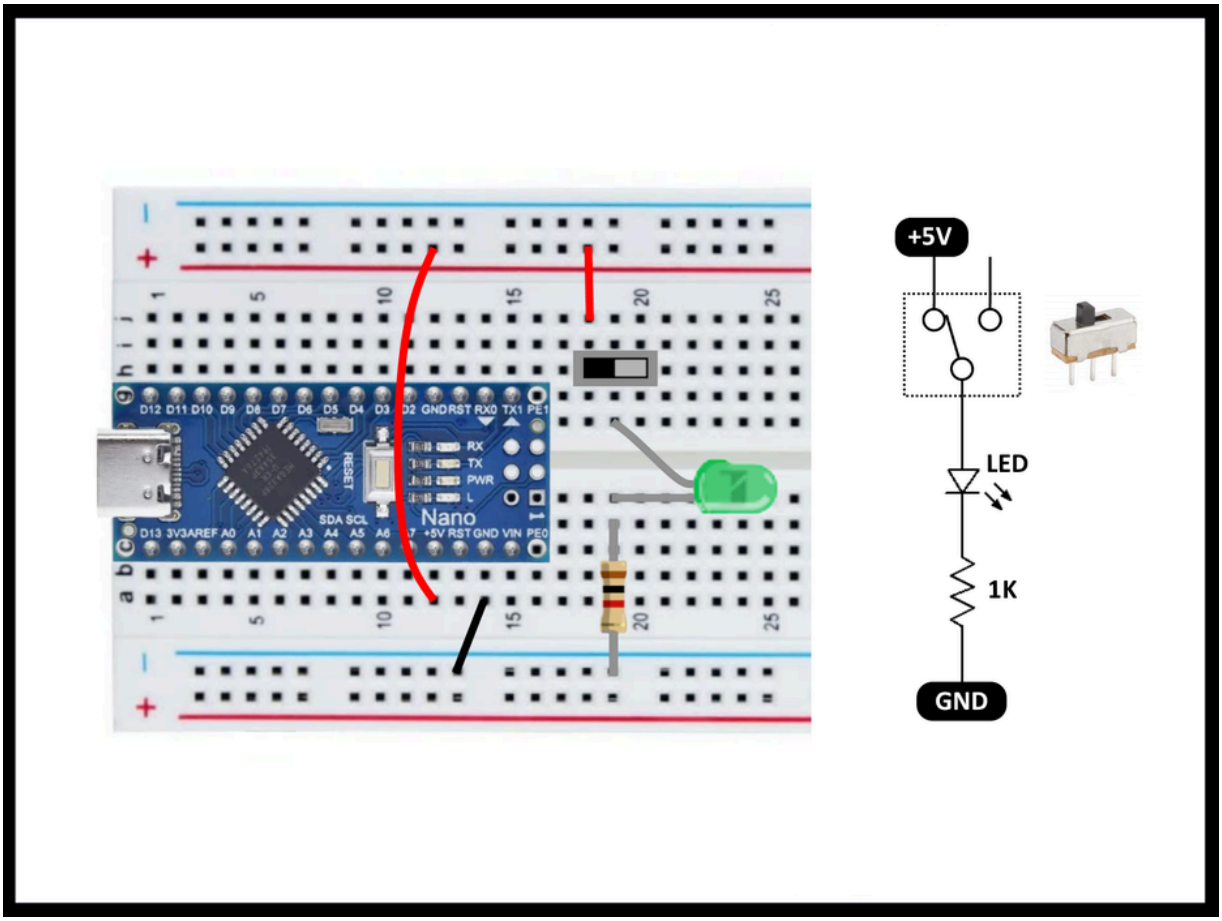
1. Beige with stripes colored: brown, black, red (1_0_00 = 1K Ohms), or
2. Blue with stripes colored: brown, black, black, brown (1_0_0_0 = 1K ohms)

The 10K resistors can be:

1. Beige with stripes colored: brown, black, orange (1_0_000 = 10K Ohms), or
2. Blue with stripes colored: brown, black, black, red (1_0_0_00 = 10K ohms)

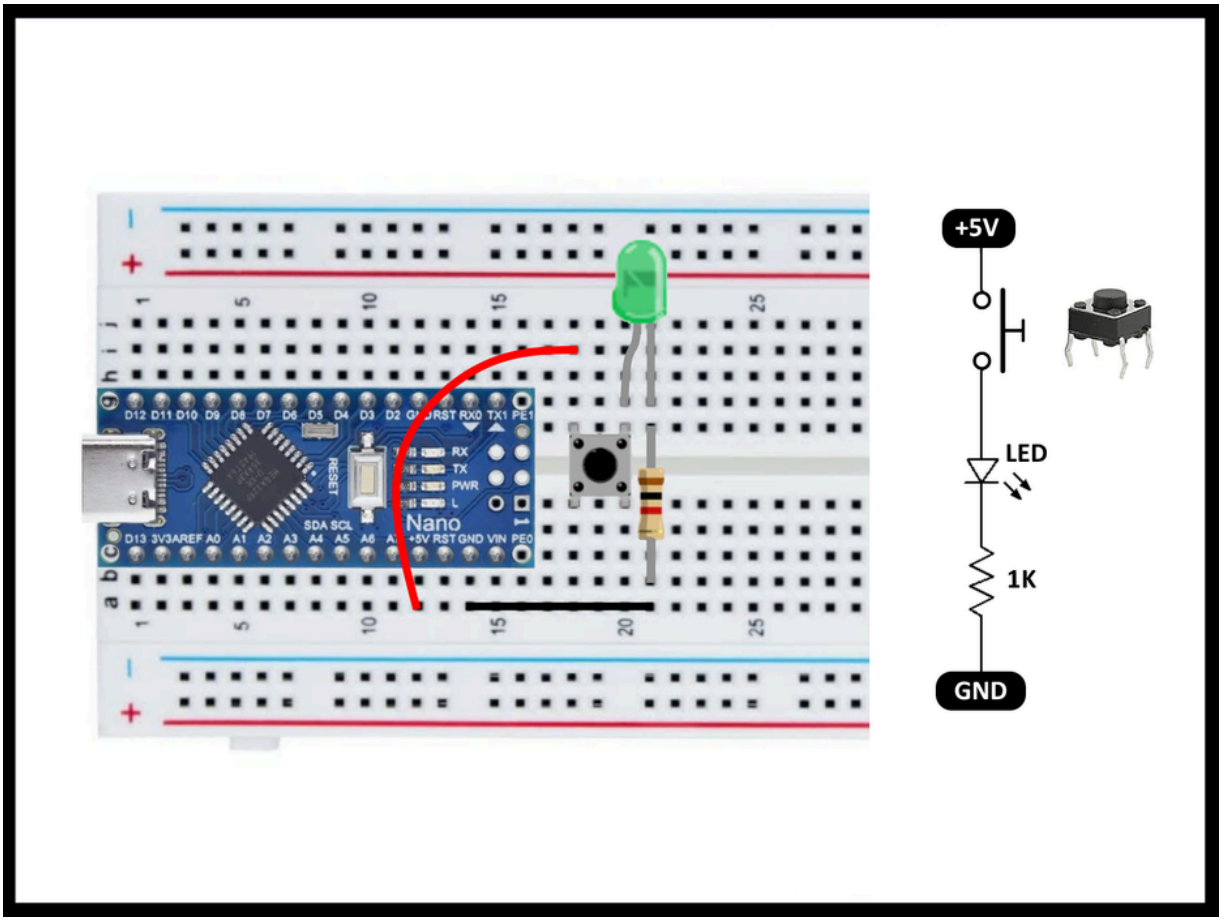
Resistor color code calculator.

Step 3: Control Electron Flow With a Switch



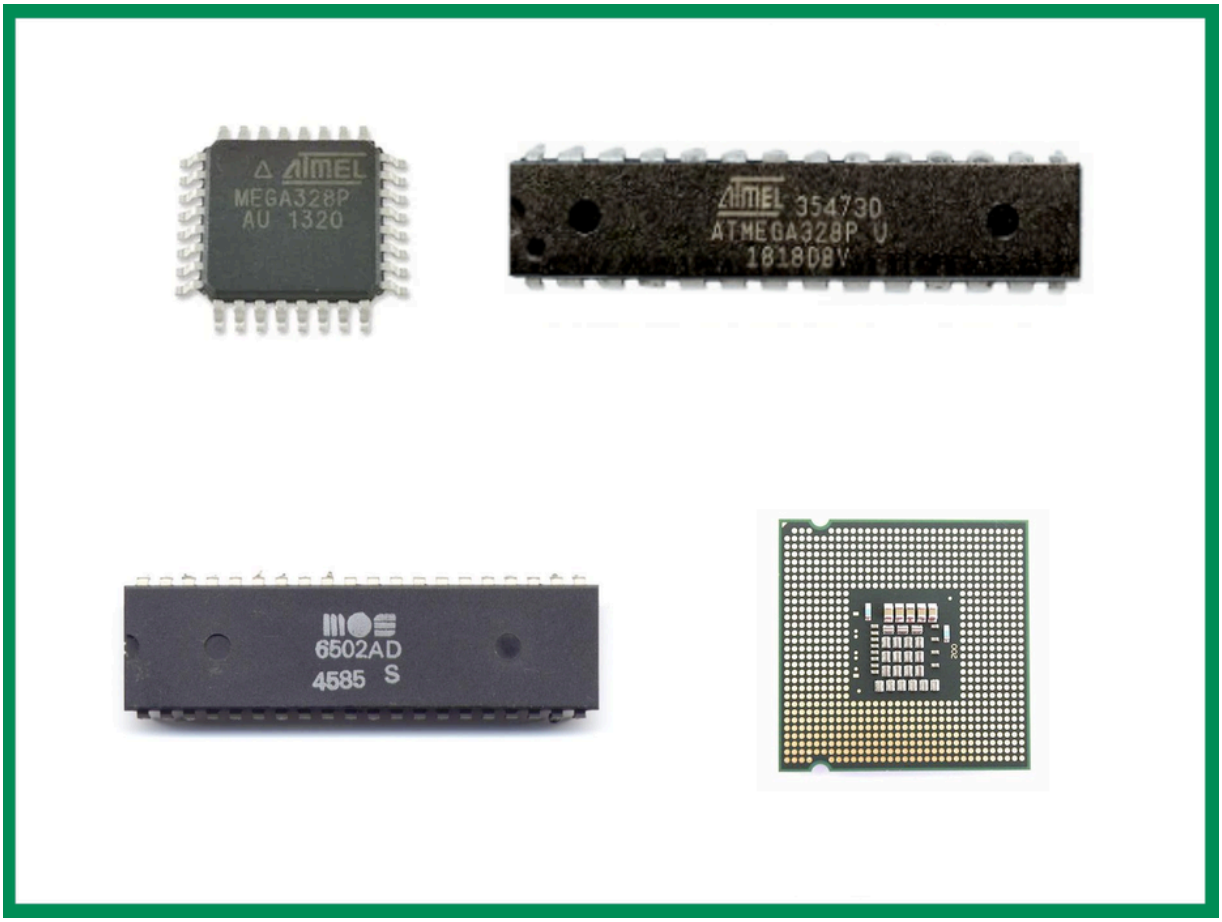
Our circuit can be updated with the addition of an ON-OFF slide switch. The switch can turn the flow of electrons on and off so that the LED is illuminated or not illuminated.

Step 4: Control Electron Flow With a Pushbutton



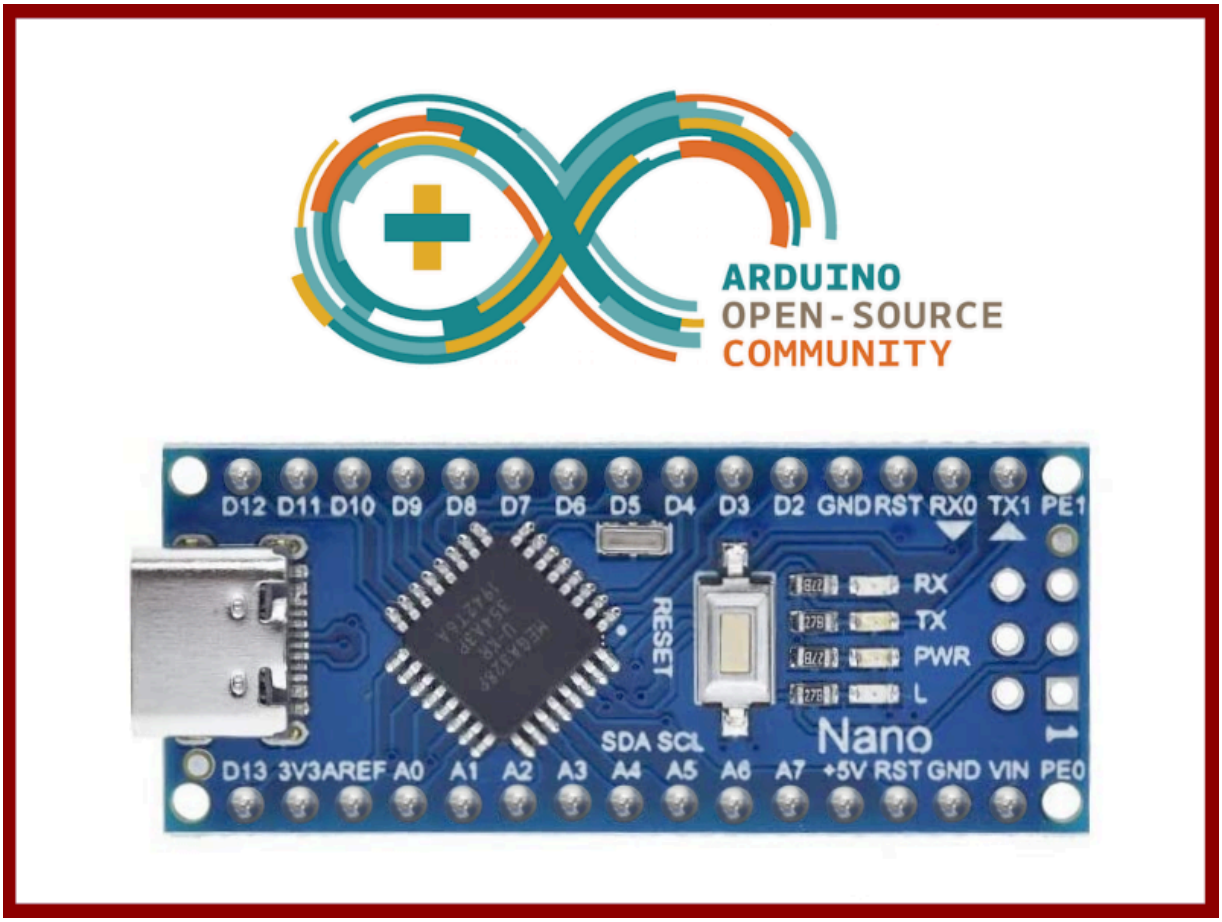
Replacing the slide switch with a momentary pushbutton allow the flow of electronics to occur when the pushbutton is pressed. The flow of electronics is blocked when the button is released opening the circuit.

Step 5: Microcontrollers



A microcontroller or microcontroller unit (MCU) is a small computer on a single integrated circuit (IC) chip. A microcontroller contains one or more CPUs (processor cores) along with memory and programmable input/output peripherals. Program memory in the form of flash memory and/or ROM is also often included on chip, as well as a small amount of RAM. Microcontrollers are designed for embedded applications, in contrast to the microprocessors used in personal computers or other general purpose applications consisting of various discrete chips. Modern MCUs often integrate one or more advanced peripheral blocks such as graphics processing units (GPU), Wi-Fi modules, or coprocessors. (wikipedia)

Step 6: Set Up the Arduino Nano



The microcontroller we'll be working with here is the ATmega328P, which is part of the Arduino Nano module that we've already placed on the solderless breadboard.

The software we will use to program and interface with the Arduino Nano is called the Arduino IDE. Let's download and install it now.

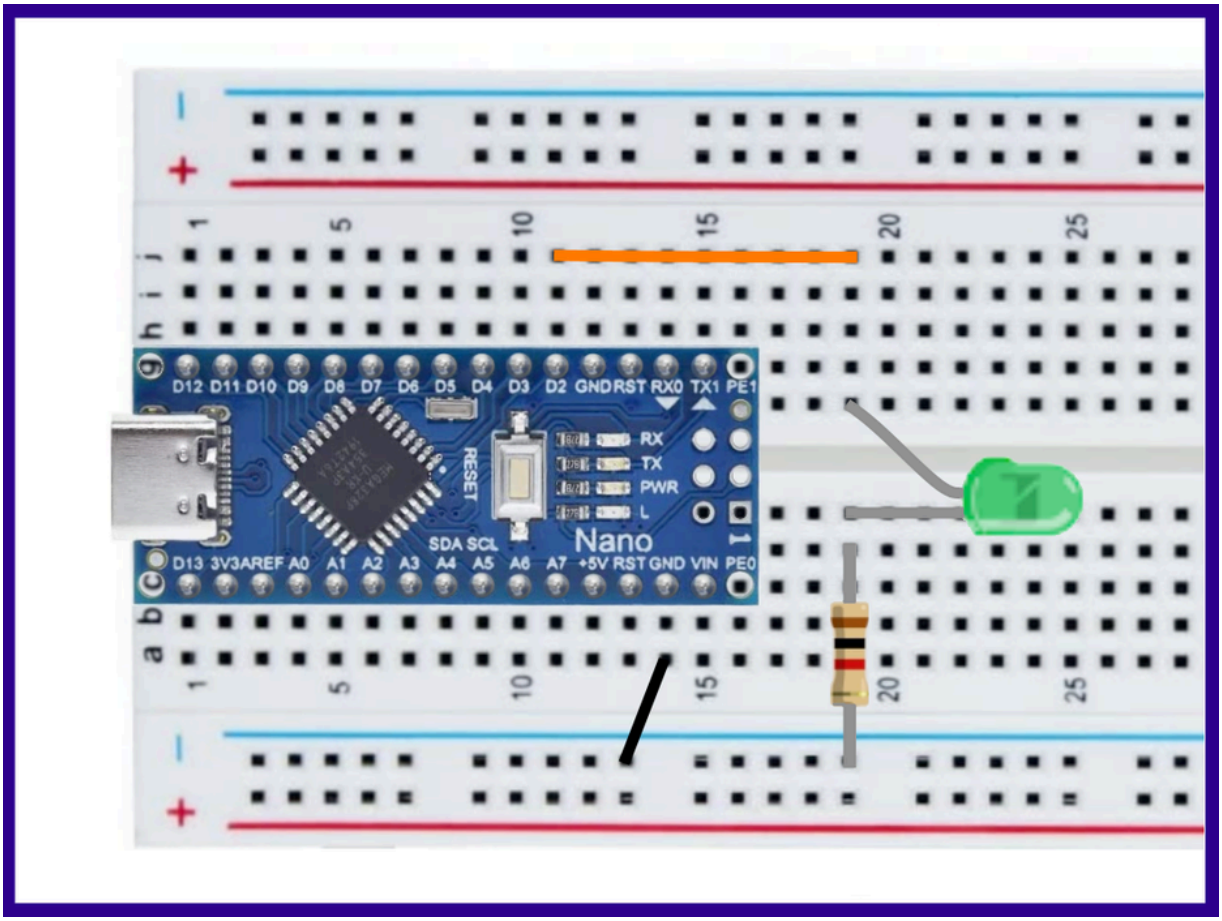
1. Connect the Arduino Nano to a USB port of your computer
2. Run the Arduino IDE
3. In the IDE, select Tools > Board > Arduino Nano
4. Also select Tools > Processor > ATmega328P (Old Bootloader)
5. Also select Tools > Port > (the USB port connected to the Nano)

Troubleshooting:

If there are multiple USB ports to select from, you can do a little test. One of the ports will disappear from the list when you unplug the USB cable from the Nano and then navigate back into the Tools dropdown menu again. That one that disappears is the port connected to the Nano.

If there are no USB ports to select (or at least there is no port that disappears when unplugged), you may need to install a driver for the USB chip on the Nano module. This chip is the CH340 which have a driver included in most modern operating systems, but there is more information here if you need it.

Step 7: Control Electron Flow With Program Code



Disconnect the power from the Arduino Nano (unplug the USB cable) and wire up the circuit shown here. This circuit is exactly like the one used in Step 2 with one important distinction. The wire connecting to the long pin of the green LED connects to the MCU I/O pin D2 instead of connecting to +5V.

Pin D2 is an INPUT/OUTPUT (I/O) pin which means that the MCU can input or output signals through it. In order to operate the LED, the pin will be used as an output. More specifically, a digital output. A digital output can only be set to HIGH (5V) or LOW (GND).

You can probably guess that when Pin D2 is set to HIGH (5V) the green LED will be illuminated as it was in Step 2. When Pin D2 is set to LOW (GND) the green LED will not be illuminated. This is a lot like using a switch but instead of having to flip the switch on or off, the LED will now be under program control.

So let's write a program. Select File > New in the IDE. In a new, empty sketch there are two empty functions: setup() and loop().

Inside setup() type:

pinMode(2, OUTPUT);

This tells the chip that we will use ARDUINO PIN 2 as an output from the chip.

Inside loop() type:

digitalWrite(2, HIGH);

This tells the chip to output a HIGH value (5V) to ARDUINO PIN 2.

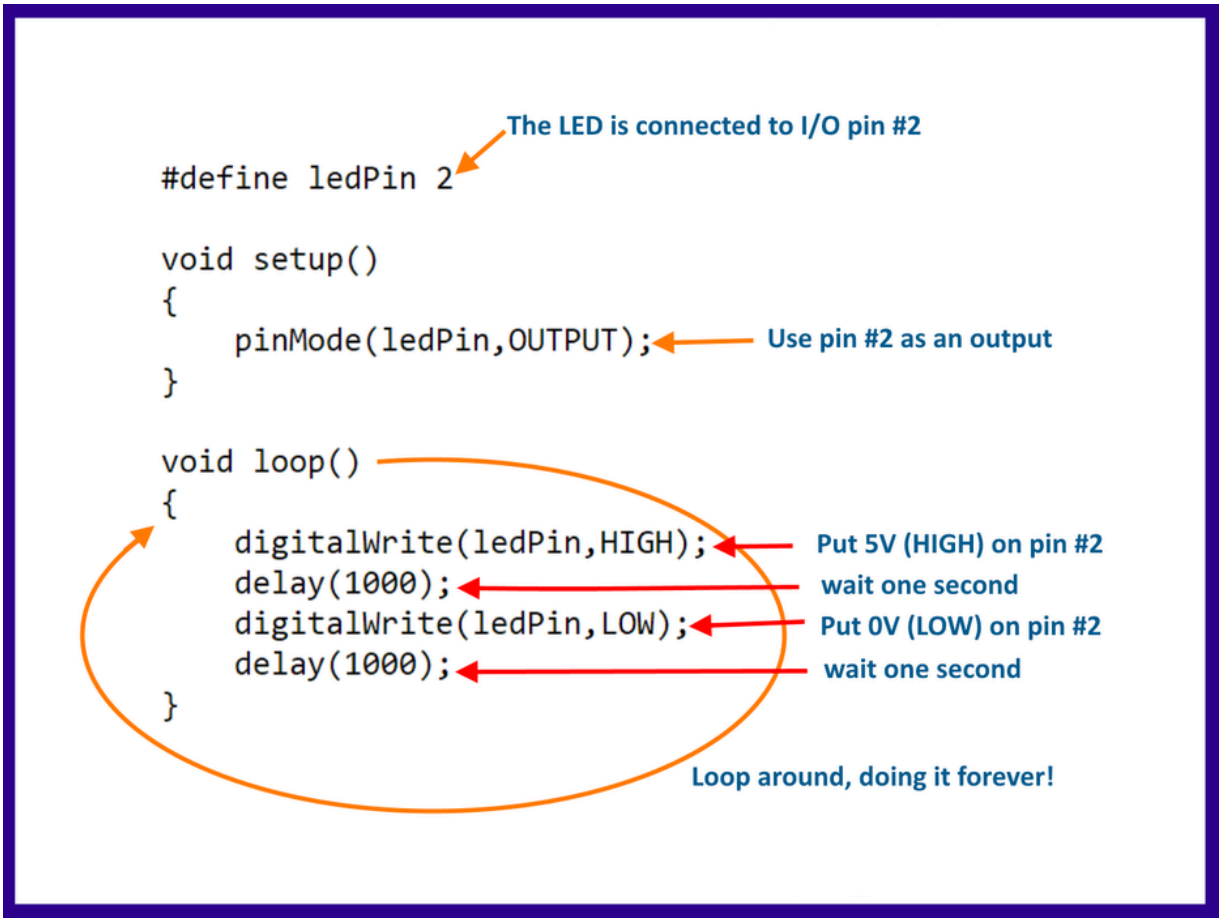
Click the arrow above the code window to compile the code and upload the program into the Arduino Nano board. The first time you compile a new program, the IDE will ask you to select the folder you want it saved to and also to give it a file name.

During download, the small LEDs on the Arduino Nano module will flicker briefly. Finally the green LED, which you have wired to the D2 pin will light up and glow steady. Congratulations! You just wrote and uploaded your first microcontroller program.

You may have noticed that the tiny red LED on the Arduino Nano has stopped flashing. That is because the program that was flashing the LED has been replaced with your new program that turns on pin D2 and lights up the LED attached thereto.

Change the word HIGH in your program to LOW and then upload the program again. As may seem obvious, this will turn the LED off.

Step 8: Looping and Timers



Let's try a more complicated program. Clean up the program from the last step to look exactly like the one in the image here.

First, we'll define a macro ledPin as 2 to represent the I/O pin 2 (aka D2) that the green LED is wired up to.

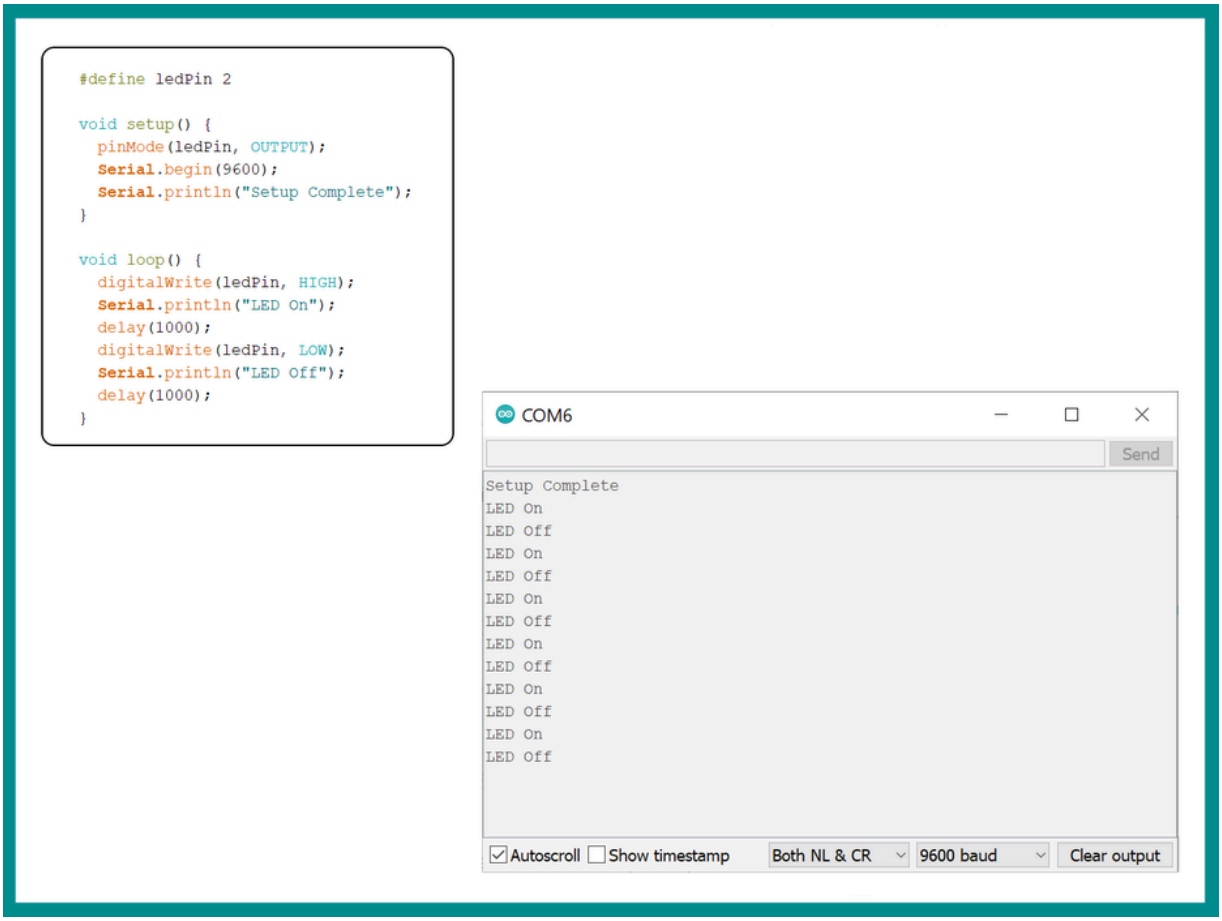
The loop() function loops around forever and ever. In each pass, the LED is turned on, we wait for a delay of 1s (1,000 milliseconds), the LED is turned off, we wait for a delay of 1s, and then we loop around and do it again.

Once you test out this code on the Nano, play around with changing the delay parameters from 1,000 to 100, or 500, or 2,000. Remember these delays are a number of milliseconds.

Note that the two delay numbers do not need to be the same. Try setting the LED on for 200 and then off for 2,000. Does the LED flash pattern match what you expect from your program.

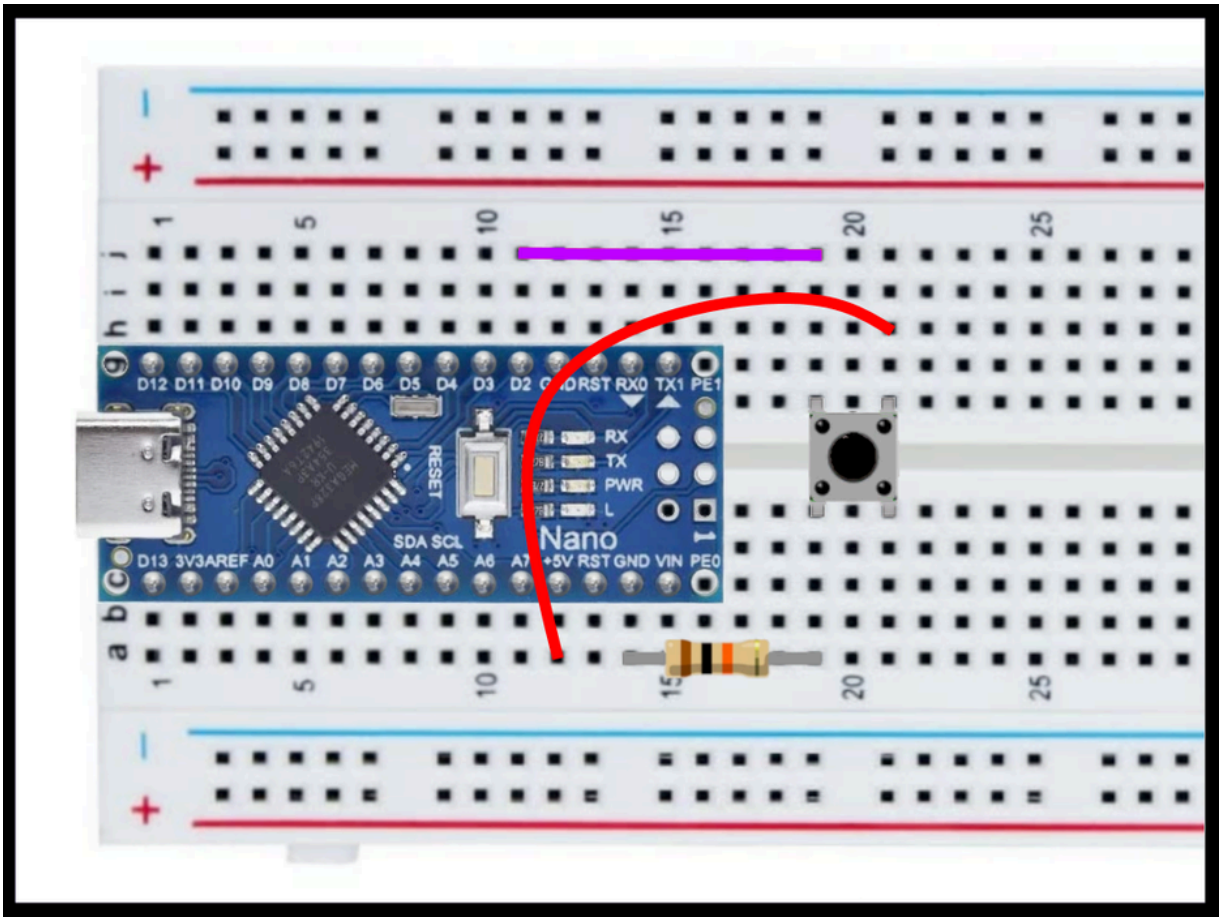
Change both delays back to 1,000 and then also change the ledPin value from 2 to 13. The tiny red LED on the Arduino that was originally flashing when we first powered the module is attached to pin D13, so this final change returns the Arduino Nano to how we found it - with the tiny red LED slowly flashing on and off.

Step 9: Program Output to Serial Monitor



Update the code to reflect what is shown in the image. Note that the setup and LED state change are now also output to the serial port. Program the Nano and then open Tools > Serial Monitor to see the output over the MCU's serial port. Printing output to the serial port can be useful for simple program debugging.

Step 10: Read Program Input From a Pushbutton



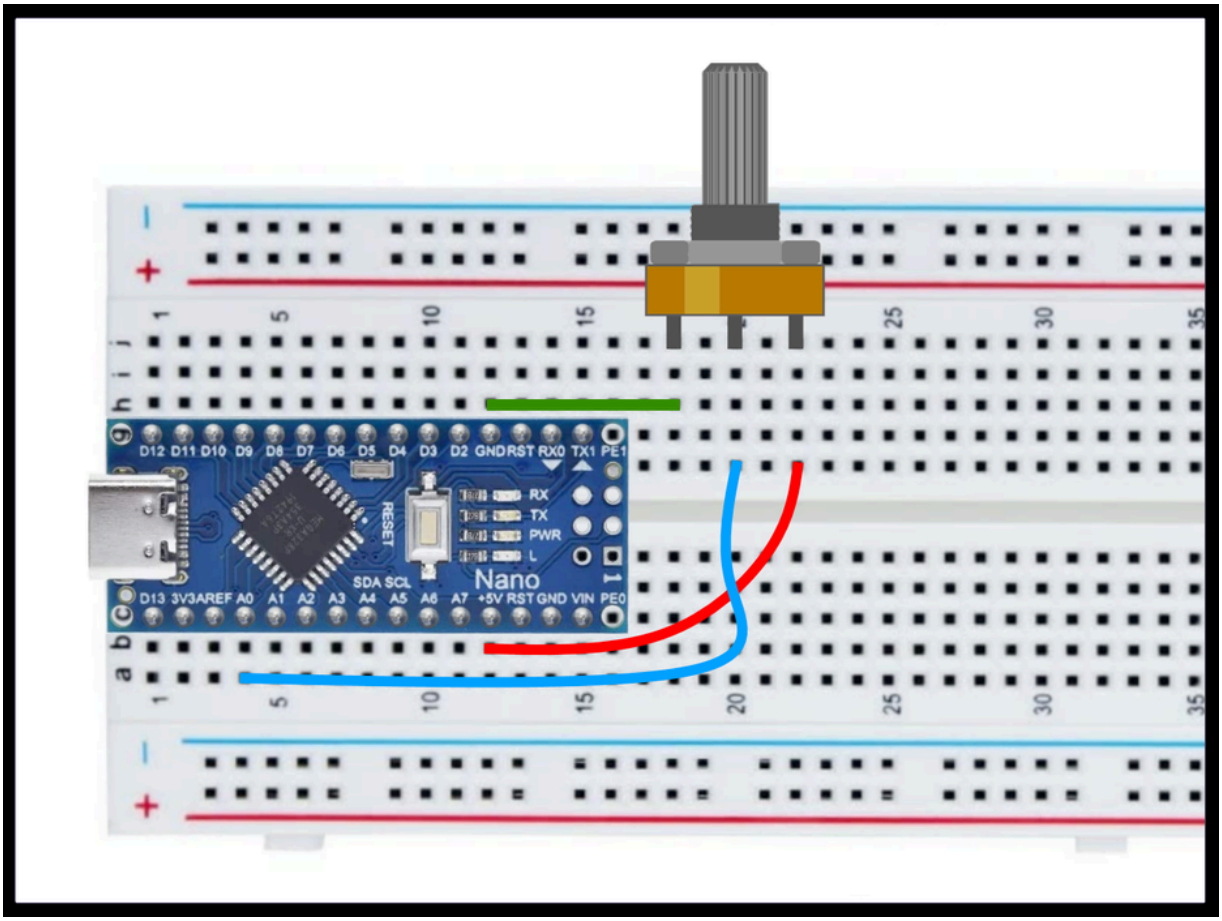
Wire the button and a 10K resistor to the Nano as shown. Download the attached *ReadButton.ino* sketch file and program it to the Arduino Nano. Again open the Serial Monitor to view the output printed over the Nano's serial port.

Step 11: Digital Vs. Analog

On and off signals - switch on and off - button pressed or not - LED on or off are DIGITAL they are only on or off - one or zero.

Analog signals (like the state of a light dimmer knob) can take on many values besides simply on and off - one and zero. Of course, values in a computer are only digital (ones ad zeros) so analog values from the real world are still stored and processed within a computer as digital numbers. Those numbers just have a wider range of values other than just high and low (one and zero).

Step 12: Reading Analog Input From a Potentiometer



Wire the potentiometer (viable resistor) to the Nano as shown. Download the attached *ReadAnalog.ino* sketch file and program it to the Arduino Nano.

Open Tools > Serial Monitor

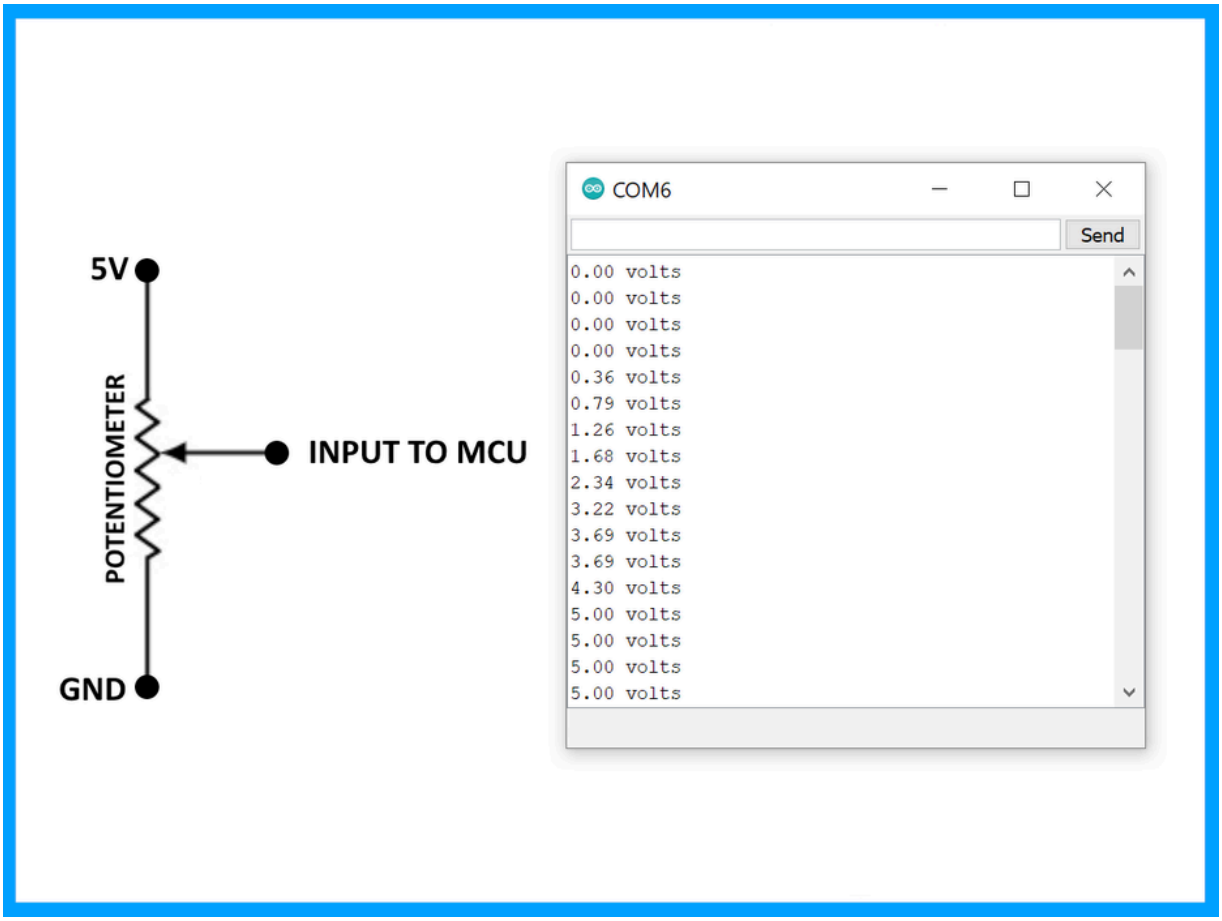
Slowly turn the potentiometer to see the values change in the serial monitor

Close the serial monitor

Open Tools > Serial Plotter

Slowly turn the potentiometer to see the plot trace change in the serial plotter

Step 13: Measuring Voltage



Reading analog values is quite interesting because it is how we get real world data into the MCU. Reading when a button is open or closed (one or zero) is one thing, but reading a range of different values allows our program to "know" much more interesting signals than simply on and off. For example, these interesting signals may represent sounds, light, images, radio, and so forth.

Note that the values from the previous *ReadAnalog.ino* sketch range from 0 at one end of the potentiometer to 1023 when the potentiometer is turned all the way to the other end. What do these values mean?

The analog values from the potentiometer enter the MCU through an analog-to-digital converter (ADC). The ADC is actually reading the voltage at the input pin. The ADC represents the voltage using ten bits. Ten bits can hold two to the power of ten (1024) different values. Accordingly, the ADC represents the input voltage as a number between 0 and 1023.

The ADC value 0 (lowest value) represents 0 volts at the input pin. The ADC value 1023 (highest value) represents 5V at the input pin. Generalizing this conversion

scale to any ADC value, we can see that the ADC value may be multiplied by (5/1023) to convert the ADC value to voltage. That scaling factor of (5/1023) maps the 0-1023 input values to 0-5 volts.

Download the attached *ReadVoltage.ino* sketch file and program it to the Arduino Nano.

Open Tools > Serial Monitor

Slowly turn the potentiometer to see the values change in the serial monitor. Note that the range of displayed values is now 0.00 to 5.00 volts.

Let's look more closely at the sketch. It is very close to our last sketch with a couple of interesting changes...

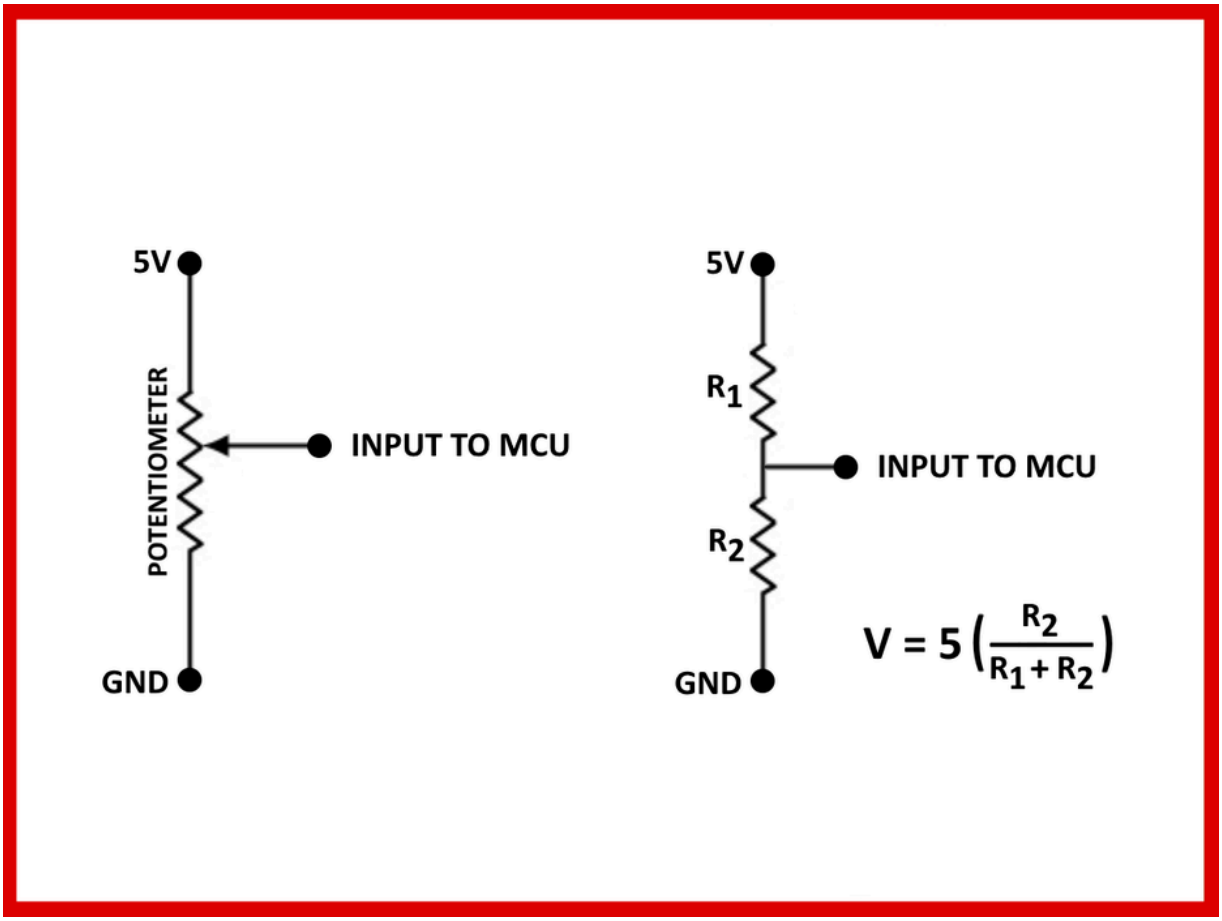
The variable *analogValue* is still declared as type *int*, or an integer number. It will only take on values of whole numbers between 0 and 1023. In other words, *analogValue* will never be 1.5 or 2.7.

In contrast, the variable *voltage* is declared as type *float*, or a floating point number. It can take on decimal values. This is important because the integer read into the *analogValue* variable will not necessarily remain a whole number once it is multiplied by the scaling factor of (5/1023).

Look at the function call to output the numerical value *voltage* to the serial monitor. It is not *println* (print a line) like last time. Instead it is *print* (print a string). Then, the following line *Serial.println(" volts")* is used to append a space and the word volts after the number. Since the number output did not end the line (it was *print*, not *println*) the space and the word *volts* will be on the same line as the number. However, since the function call to *println* to outputs the space-volts string, it does end the line. This allows the next number printed to begin on its own new line.

The subtleties of output formatting in computer programs are simple but complicated. They show us the multitude of actions that we take for granted when writing or typing. Our brains have just learned to do many things automatically, such as moving the pencil to the next line or hitting return at the end of a line. We must be more explicit about such things when writing computer programs.

Step 14: Voltage Dividers



The potentiometer we've been using has a total resistance of 10K ohms (10,000 ohms). As the schematic symbol implies, the potentiometer is actually a long 10K resistor connected between the two outer pins of the potentiometer. The center pin of the potentiometer connects to a wiper that sweeps across the length of the resistor as the shaft is rotated. The wiper effectively splits the resistor into two resistor portions that are connected in the middle at the center pin such that R1 + R2 = 10K. The allocation of the total 10K resistance between R1 and R2 is changed by rotating the shaft of the potentiometer.

If the potentiometer shaft is adjusted to its center point, R1 = R2 = 5K. This will equally divide the total voltage. The total 5V will be divided in half and the ADC will see 2.5V at the input pin.

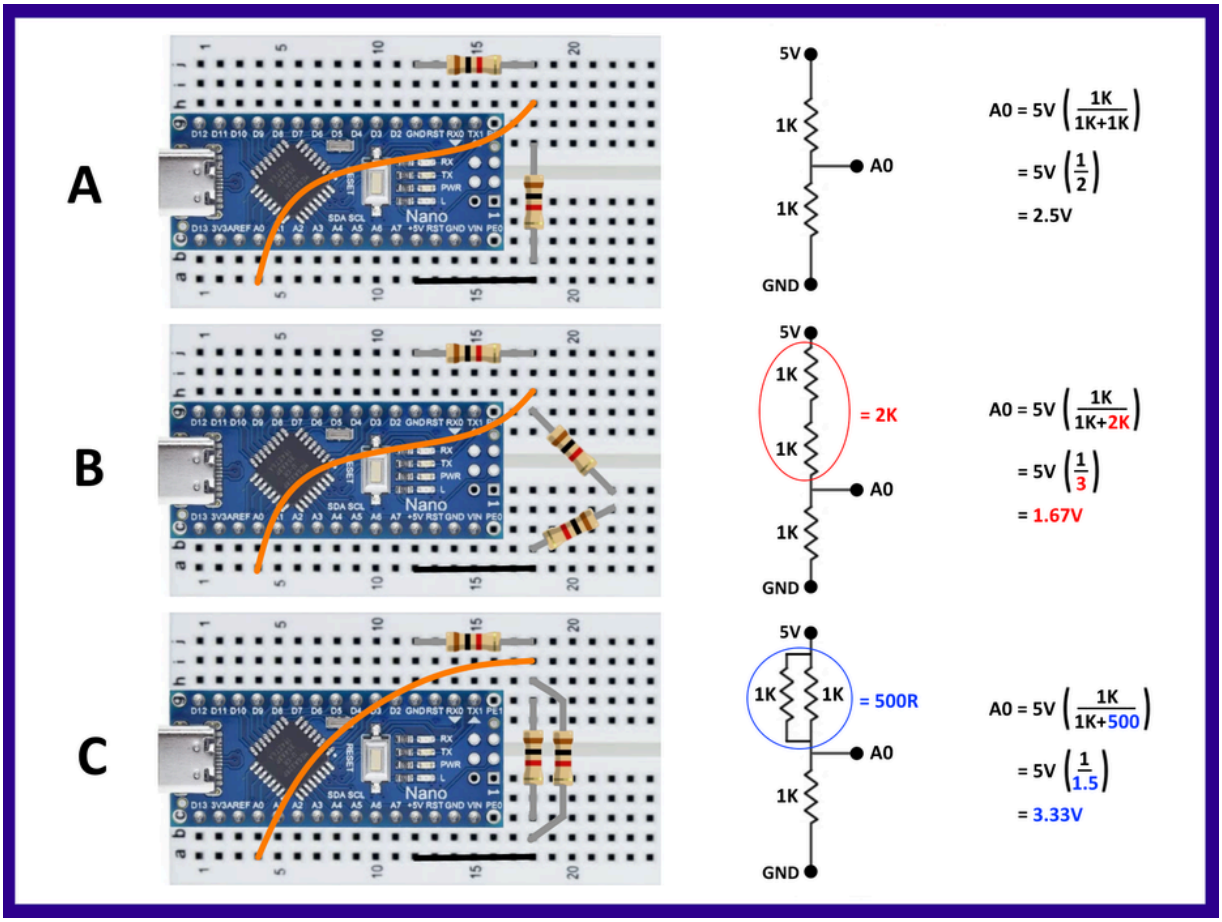
This structure of two resistances with an output tapped between the resistances is common and very useful. It even has a special name: **voltage divider**. The two resistances of a voltage divider can be the two portions of a potentiometer, two separate resistor components, or various other resistive loads. An example voltage

divider we will use later will have one resistor and one sensor that changes its resistance based on whatever it is sensing.

The formula shown in the image illustrates how the values of the two resistances R1 and R2 determine how the total voltage (5V in this case) gets "divided" to create the output voltage being measured. Don't worry too much about all of the math for now, you will encounter this structure again and again and it will eventually click into place.

Useful trick: The value of an unknown resistor can be determined by creating a voltage divider including the unknown resistor and a known resistor. The ADC is then used to measure the voltage output from the voltage divider. The measured voltage is plugged into the formula allowing us to calculate the unknown resistance.

Step 15: Resistor Structures



Case A: If we remove the potentiometer and replace it with two equal 1K resistors R1 and R2, the division of total voltage will be exactly half (2.5V) just as when we set the potentiometer to its midpoint creating two equal R1 and R2 resistive portions. Set this up on the breadboard and try it out. The *ReadVoltage.ino* sketch file is still useful for this experiment.

Case B: Replace the 1K resistor R1 from Case A (that's the resistor between A0 and 5V) with a 2K resistor to establish the illustrated voltage divider ratio. But wait, we don't have a 2K resistor! Luckily, resistances add up in series, so two 1K resistors placed in series are equivalent to one 2K resistor. Set this up on the breadboard and try it out.

Case C: Replace the 1K resistor R1 from Case A (that's the resistor between A0 and 5V) with a 0.5K (500 ohm) resistor to establish the illustrated voltage divider ratio. But wait, we don't have a 500 ohm resistor! Luckily, two equal resistors become half of the original resistance when arranged in parallel. Two 1K resistors placed in parallel are equivalent to one 500 ohm resistor. This "parallel halving" can be conceptualized as

twice the current passing through two parallel pipes (or resistors) than would pass through only one of them. The effective doubling of pipe width in the parallel structure cuts the resistance in half.

Techniques using various resistor configurations (also called resistor networks) are useful to modify voltage levels and bend electricity to our will.

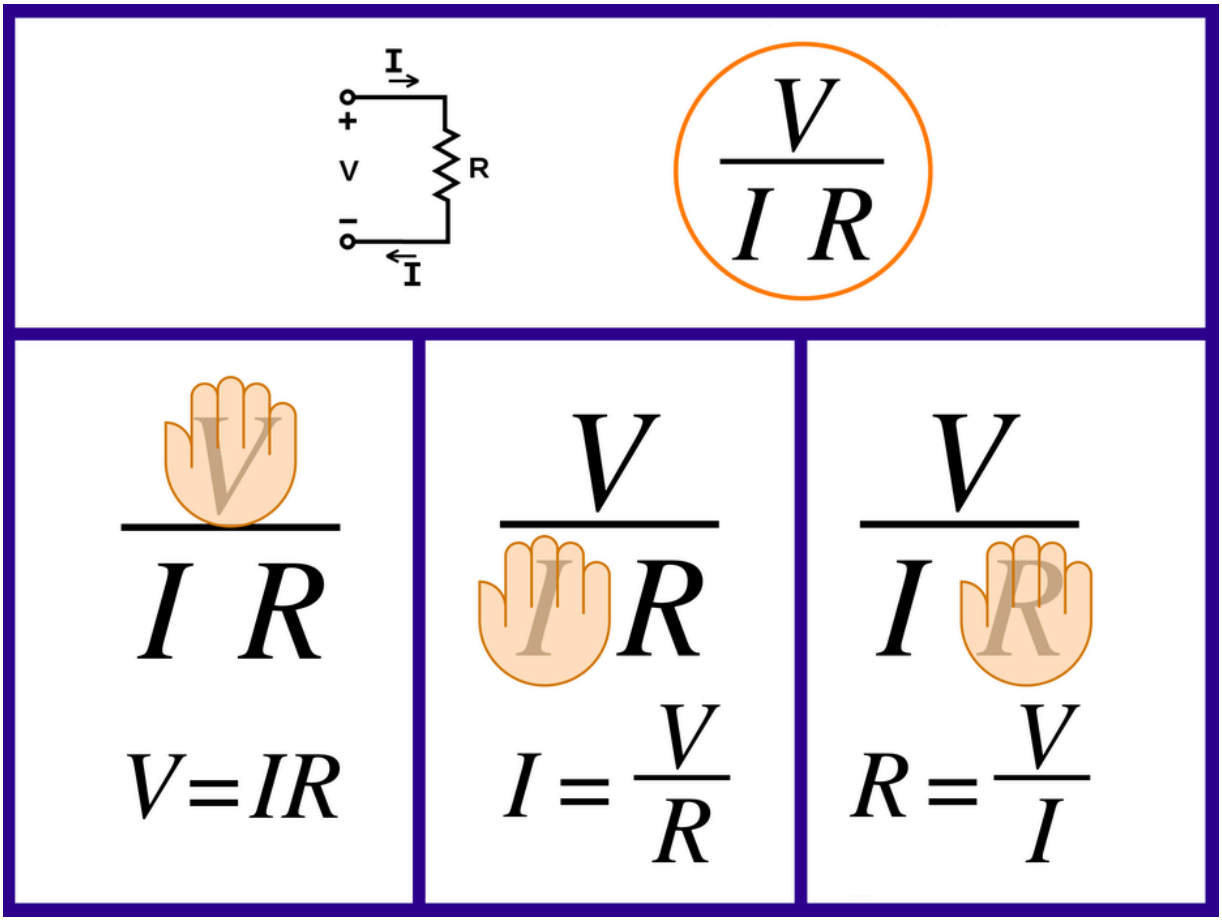
Some questions to ponder:

What is the equivalent resistance of THREE parallel 1K resistors?

What is the equivalent resistance of a 1M (one million) resistor in series with a 0.1 ohm resistor? Are both resistors really necessary? Would such a structure ever be found in a commercial product?

What is the equivalent resistance of a 1K resistor in parallel with a 10K resistor? Warning: This one is tricky. It will probably require a little research if you want to attempt the challenge, but the payoff is that you will also discover the theory behind why the two parallel 1K resistors in Case C above combine to form a 500 ohm resistor.

Step 16: Ohm's Law



A lot of attention is paid to voltage, current, and resistance because the three quantities are related in a simple, reliable way. Let's dig deeper...

Voltage (V) is the difference in electric potential energy between two points. To make an analogy, when talking about mass, gravitational potential can be thought of as how high something has been raised off the ground and thus how much energy it has to release when it falls to the ground. Similarly, a charge at 5V potential has more energy to expend getting to 0V (ground) than does a charge at only 2V. Voltage is sometimes called "electrical pressure" because it is a bit like water pressure. To give the tap water in your house pressure as it flows out of the faucet, water is often pumped uphill to a water tower. The higher the tower (gravitational potential above the ground), the more pressure or the harder water can push through the pipe. Similarly, the more voltage (electrical potential raised above ground), the harder the electrons can push through the wires.

Electrical **Current (I)** is very similar to the notion of water current in a river or a pipe. Current is how much stuff (electrons in this case) flow through per unit time. For

example, gallons-per-minute of water or charges-per-second of electricity.

Resistance (R) can be thought of as the “tightness” of the pipe. The narrower the pipe is (higher resistance), the less current flows through for a given potential (voltage). You can make more current flow through a pipe by pushing it harder (higher voltage) or opening the pipe up (less resistance).

The relationship between these three qualities is formalized as **Ohm's Law:**

V = I x R

where voltage (V in volts) equals current (I in amperes) times resistance (R in ohms).

If you are not a fan of algebra, a useful mnemonic tool is illustrated here. Starting with the "V over IR" expression in the orange circle, we can simply over the parameter we'd like to know and the remaining two parameters display the necessary calculation to find the desired parameter.

How much current flowed through our 10K potentiometer when it was connected between 5V and Ground?

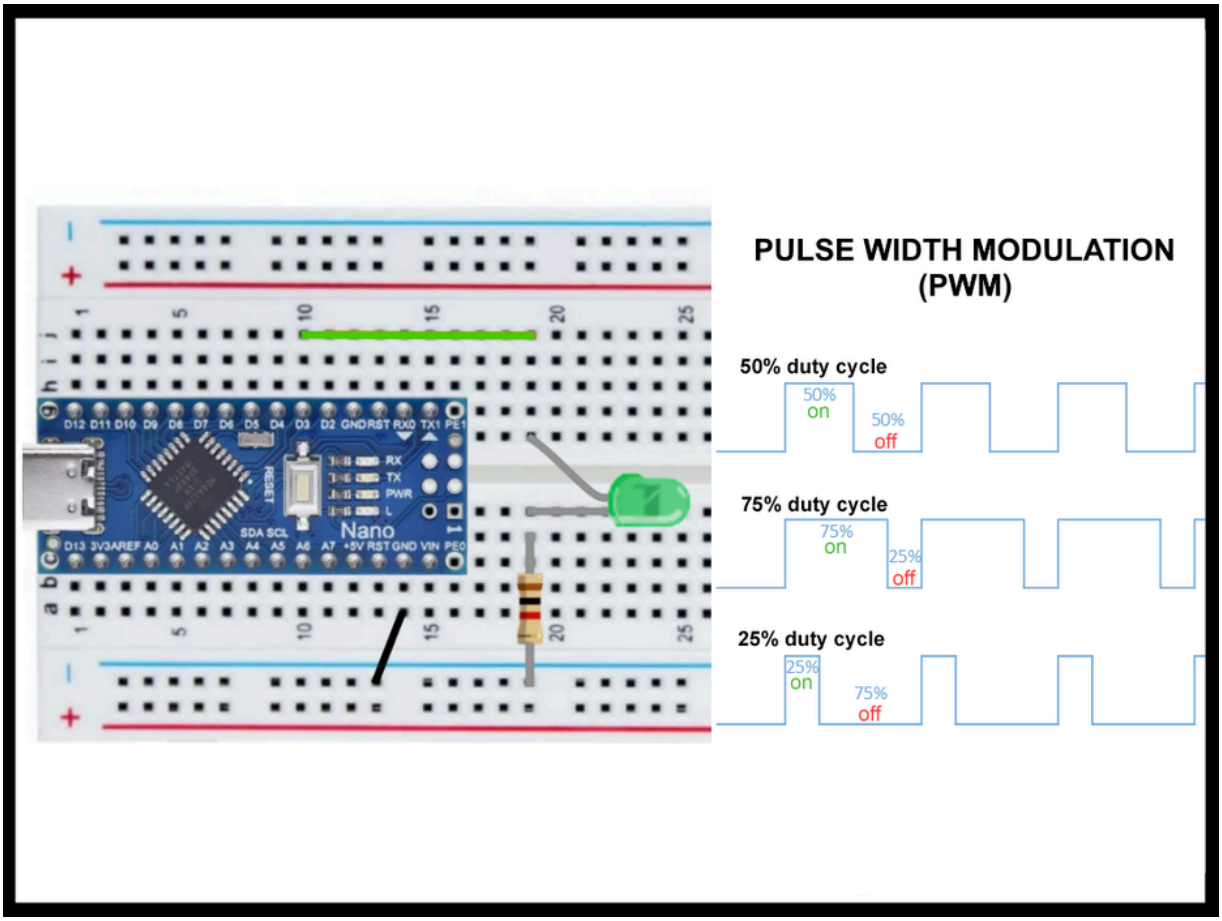
5V = Current x 10,000 ohms

Current (I) = 5 / 10,000 = 0.0005 A = 0.5 mA

This may seem very small, but keep in mind that the pins of a digital silicon chip (like our MCU) really do not like to supply (or sink) a lot of current. This is part of why we've been placing a "current limiting resistor" in series whenever connecting an LED to the MCU.

Note that the same amount of current flowing through the long 10K resistor of the potentiometer also flows through each of the potentiometer's resistive portions R1 and R2. This is due to a generalization of the law of conservation of charge: "current in" generally equals "current out". The more water you drink, the more you will probably need to visit the restroom.

Step 17: Adjusting Light Brightness



Build this circuit. It's the same one we used in Steps 7, 8, and 9 with one important difference. The MCU output is now set to pin number 3 instead of pin number 2. Why is that? Pin 2 does not support PWM but pin 3 does. We'll get to why that's important for this experiment.

You might think that we can just dim a light (or other wise adjust its brightness) but changing the voltage on it. Well, that might by true in some cases, but LEDs conduct exponentially, so we can think of them as only being "all on" or "all off". Furthermore, our simple MCU does not have a DAC (digital to analog converter). Many MCUs do have DACs, but this one does not. Without a DAC, the MCU cannot actually make an analog value, but only on (5V) and off (0V).

If you download the *LEDdimmer.ino* sketch and have a look, you will notice the use of a function called *analogWrite()* even though there is no analog output on this MCU.

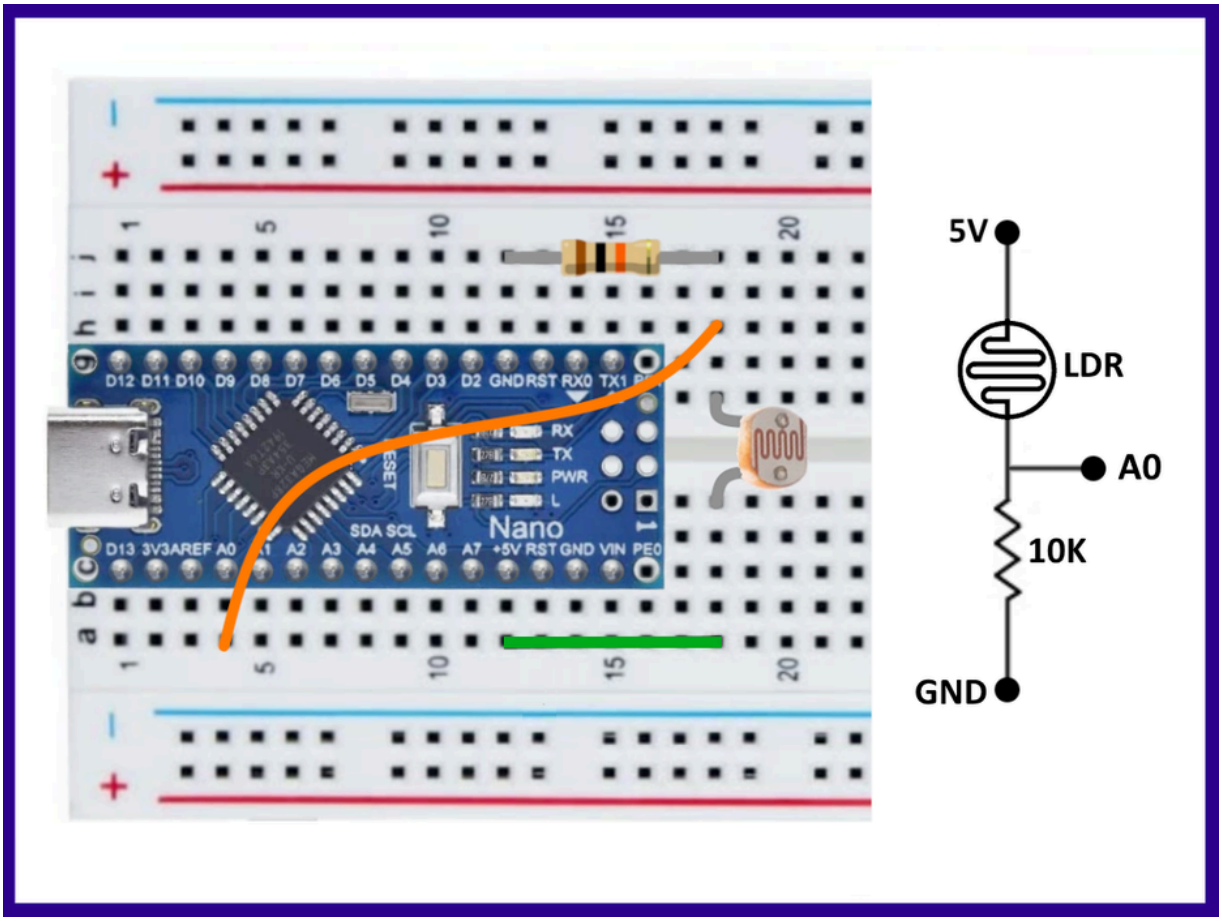
What is going on when we call *analogWrite()* is actually a trick that involves pulsing the output pin rapidly on and off. The duty cycle of this pulsing is what is adjusted to create what seems sort of like an analog signal. The output signal is only ever 0V or

5V (never actually analog) but the duty cycle specifies what percentage of time that the pulsing signal is high versus low. This technique is called **PWM (pulse width modulation)**. Example PWM waveforms are illustrated in the image.

The *analogWrite()* function can be called with the value 0 (pulsing always low or 0V), the value 255 (pulsing always high or 5V), or any value between to specify what amount of the time the pulsing is high. The *LEDdimmer.ino* code uses the values 50, 150, and 250 to generate three different levels of brightness for our green LED. Even though we cannot see it, the PWM is actually pulsing the LED on and off very rapidly.

Feel free to try out other values or patterns or values. How about wiring up the potentiometer, reading the potentiometer from pin A0, scaling the value read to the range of 0-255, and then using that scaled value to drive the LED IO pin 3. Go ahead and give it a shot!

Step 18: Light Sensors

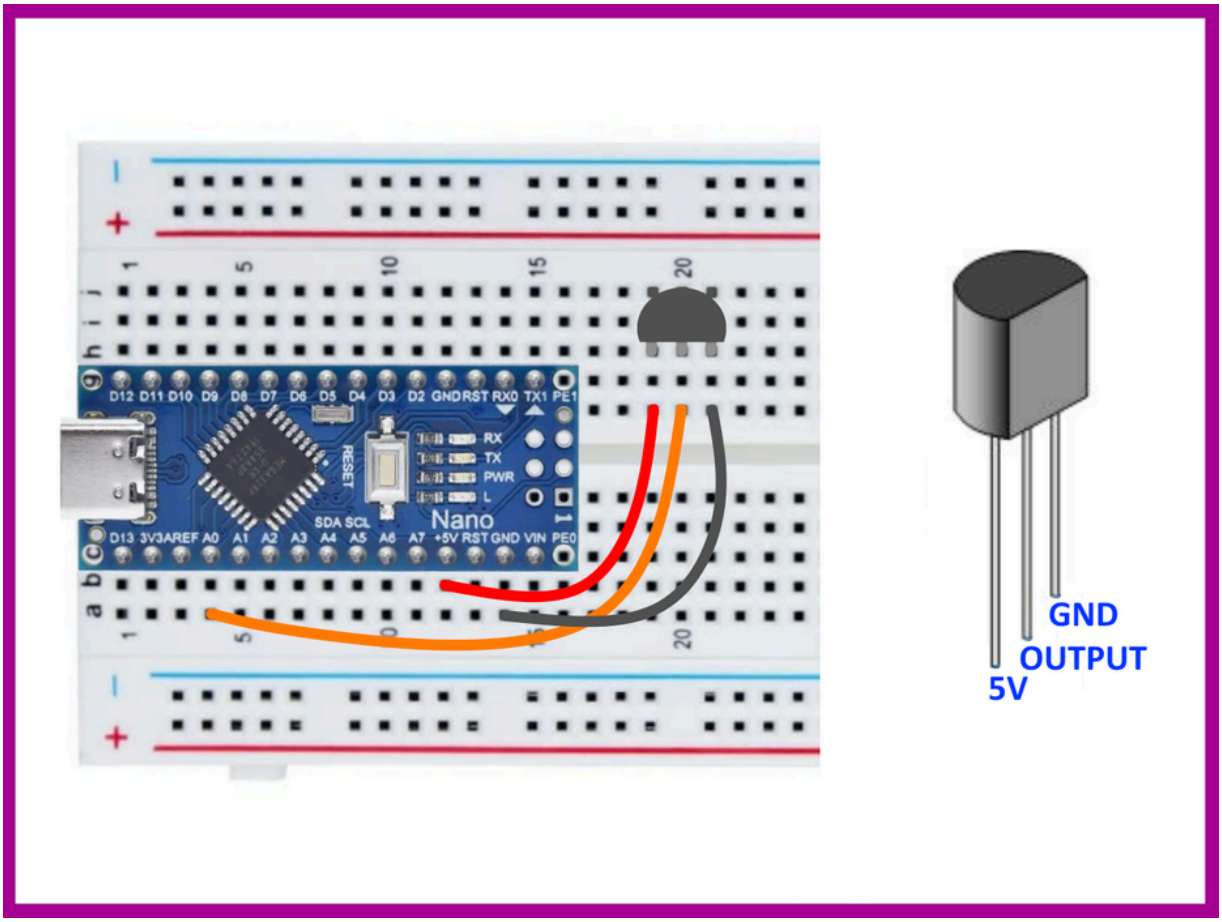


A common light sensor is a Photo Resistor, which is also known as a Photo Cell or Light Dependent Resistor (LDR). An LDR is a resistor with a resistance that decreases when brighter light shines upon is photosensitive surface.

Wire up the Photo Resistor and a 10K Ohm Resistor in a Voltage Divider structure as shown. Connect the output of the Voltage Divider to pin A0 of the Nano as shown. Use the *ReadVoltage.ino* sketch to read and display the output of the voltage divider. Open the Serial Monitor to view the output printed through the Nano's serial port. Notice now the voltage changes when the Photo Resistor is shielded from ambient light or when a brighter light shines upon it.

Can you figure out what will happen if you swap the LDR and the 10K resistor around so they are on different sides of the voltage divider? Give it a ponder and then try it out to test your theory.

Step 19: Temperature Sensors



Wire up the TMP36GT9Z Temperature Sensor (datasheet) to the Nano as shown. Be careful to identify the correct component from the part number on the flat surface of the body. Also be careful to correctly orient the component according the flat surface of the body.

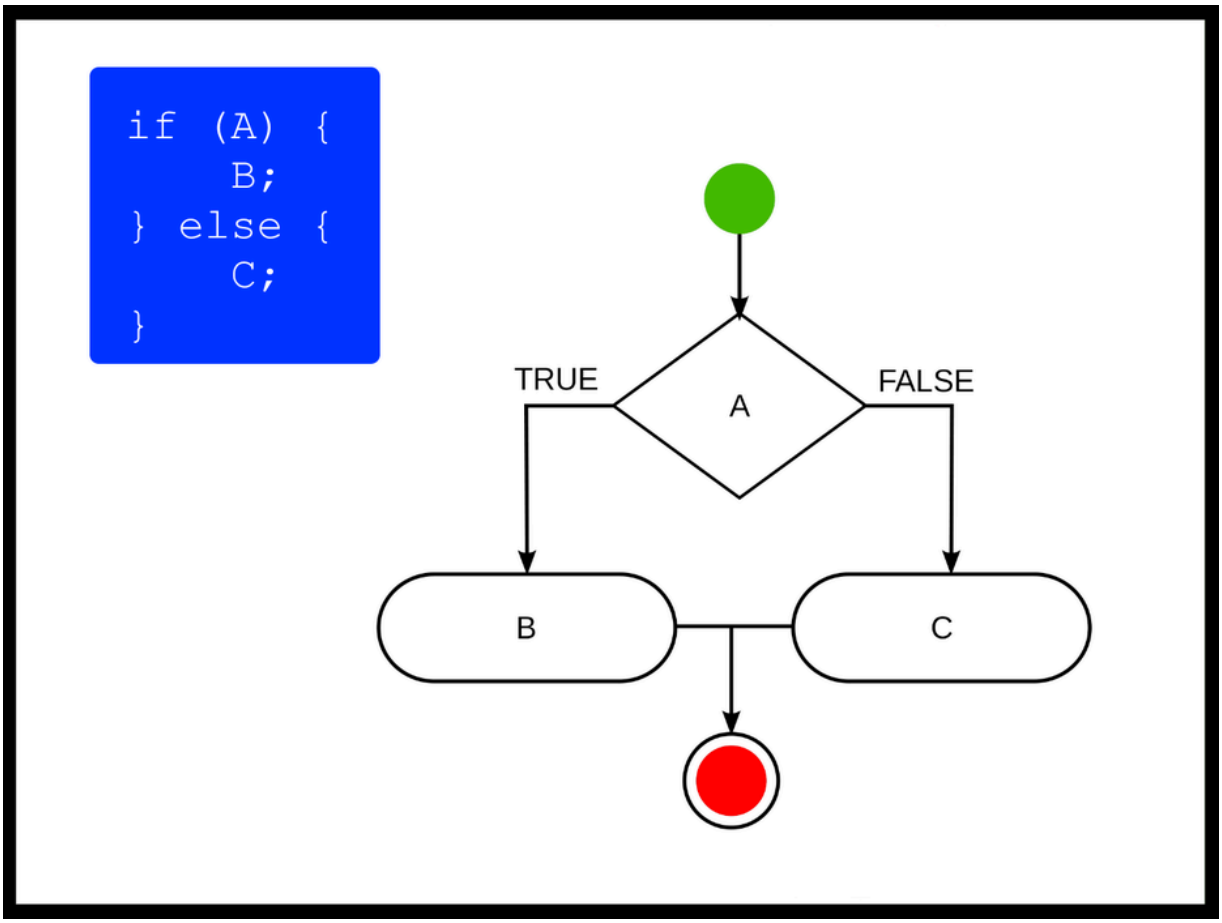
Download the attached *ReadTemp.ino* sketch file and program it to the Arduino Nano. Open the Serial Monitor to view the output printed over the Nano's serial port. You may notice that the measured temperature is close but not correct. Any variation is often due to the USB power supply voltage not being exactly 5.00V. Calibrating sensors against noise, power fluctuations, etc. is a common matter for careful attention in electronic design. For this education purpose, we will just accept that it is not perfect.

Identifying the TMP36 Component: Your parts kit includes SIX components that look like the one in the image. That "look" is called a TO-92 package component. TO-92 components generally have a small black body with one flat side and three leads extending from one end. TO-92 components are often automatically identified as

transistors. While some TO-92 packaged devices certainly may be transistors, not all of them are.

Of the SIX TO-92 components in the parts kit, ONE of them is the TMP36 Temperature Sensor and FIVE of them are 2N2222A Bipolar Transistors. Examining the flat surface of the component's body (perhaps with a magnifying lens) will reveal some very tiny printing that will indicate the difference between the TMP36 and 2N2222 devices.

Step 20: Program Control Flow



Most of the program code we've looked at so far has simply used the setup() and loop() functions to control the flow of the program. Whatever we put in setup() happens once when the program starts. Whatever we put in loop() literally loops around and keeps happen for ever.

Looking at the *ReadButton.ino* sketch that we used earlier, you will see an if-else block. This is illustrated in the image above: If A is true, then do B. Otherwise (else) do C. Examine how that structure is used in the *ReadButton.ino* sketch to do different things when the button is pushed versus when the button is not pushed.

Sometimes we need more control. Let's consider some examples...

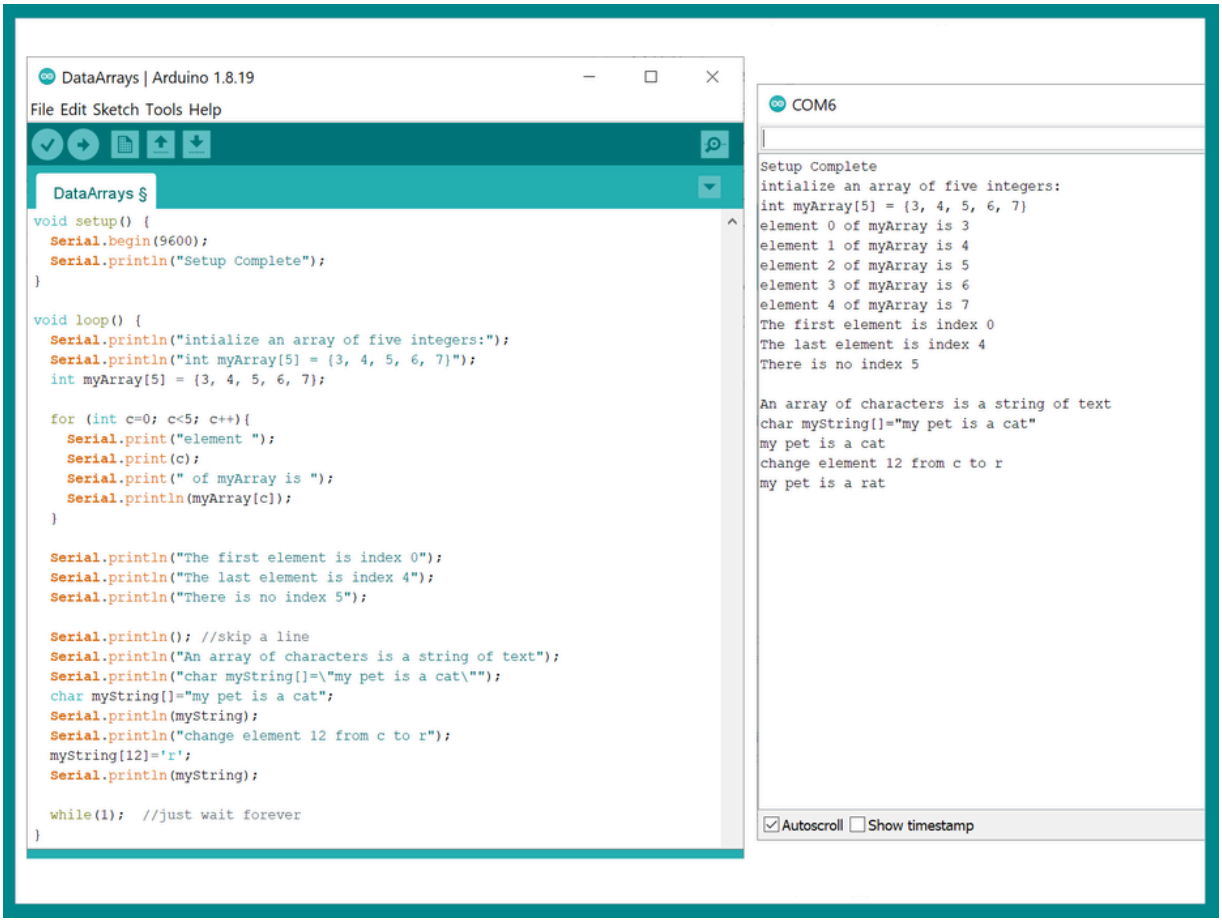
- sit and do nothing until an input is received
- keep doing something periodically until an input is received
- do something 10 times
- do something 10 times but stop if an input is received
- do something 10 times where each of those does something else four times
- receive a keyboard input and do different things based on which key was pressed

Download and run the *ControlFlow.ino* sketch attached here. The sketch demonstrates simple examples of the four most commonly used program control flow mechanisms: if, else, for, while. Carefully examine how they are used in the sketch.

More information on these four along with the other available control mechanisms (do...while, switch...case, return, break, continue, goto) can be studied in the Arduino Documentation under Control Structure. Over time, you will encounter examples of these various forms. You will develop a feeling for which ones you like using for certain types of tasks. Many tasks can be accomplished just as well using two or three different control mechanisms, but some tasks lend themselves more to one specific type of control mechanism.

Program Comments (also called **inline documentation**) are important notes placed inside a program while writing it. They can tell others what the programmer was thinking when they wrote the program. They can also remind the programmer what as going on when they look at the code again later. Notice the comment block at the top of the *ControlFlow.ino* sketch demonstrating the use of // and /*...*/ structures for commenting your code. Commenting your code is more important than you might realize right now. Trust us... learn it, love it, do it.

Step 21: Storing Data in Arrays



So far, we have used variables to store information. Variables have types like integer (int), character (char), or floating point (float).

Multiple pieces of related data can be stored in a collection called an array.

Each piece of data in the array is called an element and an index number is used to select the various elements. Just like a variable has a type, the elements of an array have types. In fact, all of the elements have the same type. So we say that it is an array of integers, or an array of characters, etc.

Download and run the *DataArrays.ino* sketch attached here. The sketch demonstrates a couple of examples of creating and using arrays. Carefully examine how they are used in the sketch.

First, an array of five integers is created (declared) and initialized:

```
int myArray[5] = {3, 4, 5, 6, 7};
```

Initializing means starting the array with its initial values preset. An array can also be created without initializing its values.

Examine how the for loop is used to index through the array named myArray. The index variable c counts from 0 to 4 to access each element of the array.

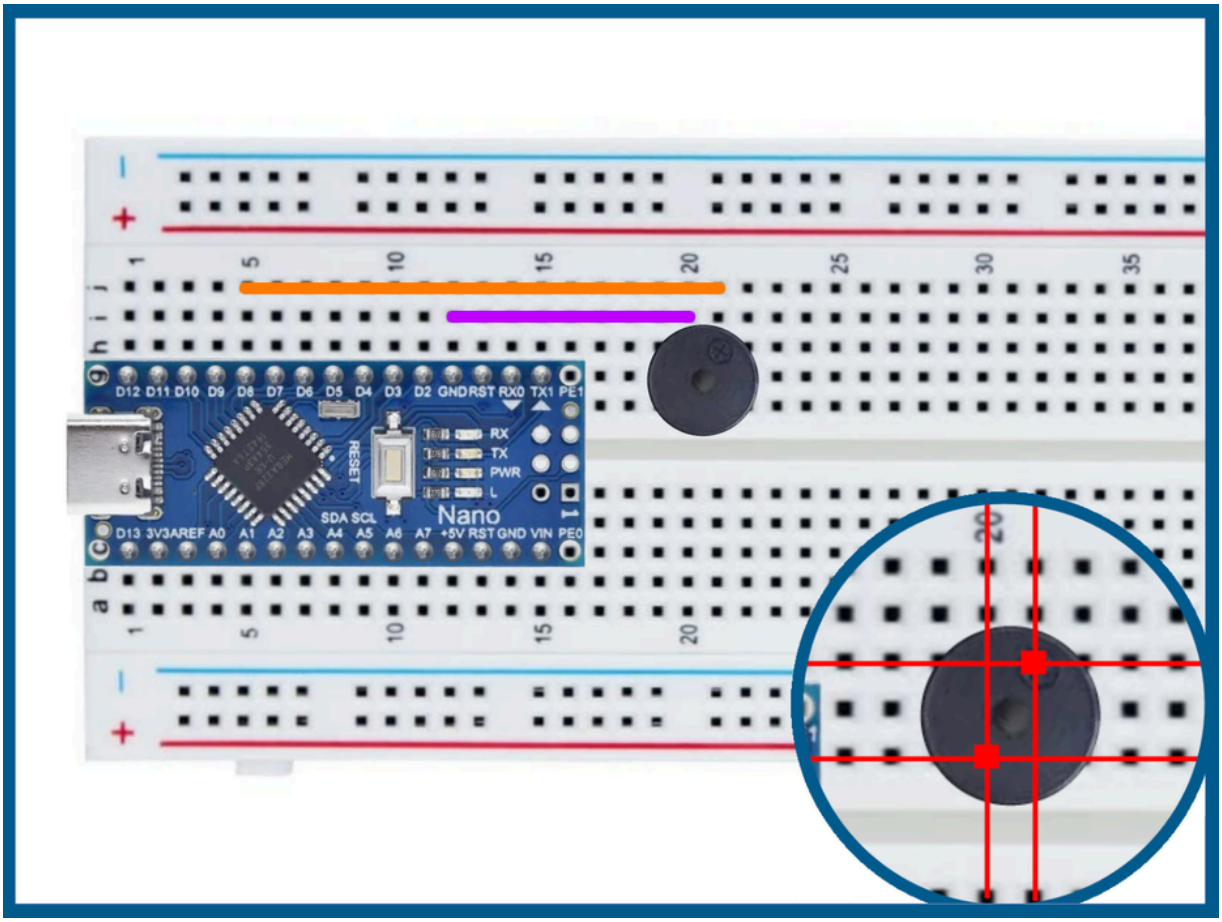
The second example shows how an array of characters can be used to store words. An array of characters is also called a string, text string, or a string of characters.

Look at the last line of the loop() function:

```
while(1); //just wait forever
```

Since "1" is always true, this while() conditional will execute its contents forever. However, its contents are empty. There aren't even curly braces, just a semicolon. So "executing the contents of the while loop" really means "do nothing". Since "1" is always true, this simple line "does nothing" forever. Execution is just stuck right there (forever) so this line of code accomplishes "halting" the loop() function so that it doesn't keep looping. This is a simple trick that is worth remembering.

Step 22: Generating Sound



Piezoelectric speakers (also known as piezo buzzers) generate sound using the piezoelectric effect. This is in contrast to the electromagnetic coils used to generate motion (vibrations) in a traditional speaker. Piezoelectric crystals physically deform slightly (compress or expand) when electricity is applied. Likewise, the crystals also build up electric charge in response to mechanical stress such as bending or squeezing.

The electrical deforming property of piezo crystals can be used to generate sound (vibrations). While that sounds is not high fidelity, it is very efficient. A piezo buzzer can be driven directly from an I/O pin without any amplifier circuitry.

Wire up the Piezo Buzzer to the Nano as shown. Since the pins of the buzzer do not have spacing that matches the solderless breadboard it is helpful to insert the buzzer at an angle as illustrated.

Download and run the *Sounds.ino* sketch attached here. The sketch demonstrates using the tone() command to generate sounds. The tone() can take three parameters:

```
tone(pin, frequency, duration)
```

The first parameter *pin* specifies which I/O pin the buzzer is connected to. The second parameter *frequency* specifies the frequency of the tone to generate, and the third parameter specifies for how lone to generate it (in milliseconds).

After the five notes are played, there is a delay of four seconds before looping around to play the notes again.

Using #define Preprocessor Directives

A #define is handled by as a preprocess before a program is even compiled. It simply replaces any instance of the first portion with the second portion. While it might seem like a variable, it is not. We cannot store anything in it or modify it at runtime. It doesn't even exist in the view of the compiler because it is handled as a preprocess. It is just a simple text replacement to make code easier to read and fixed values (such as constants and I/O pin numbers) easy to globally modify from one place.

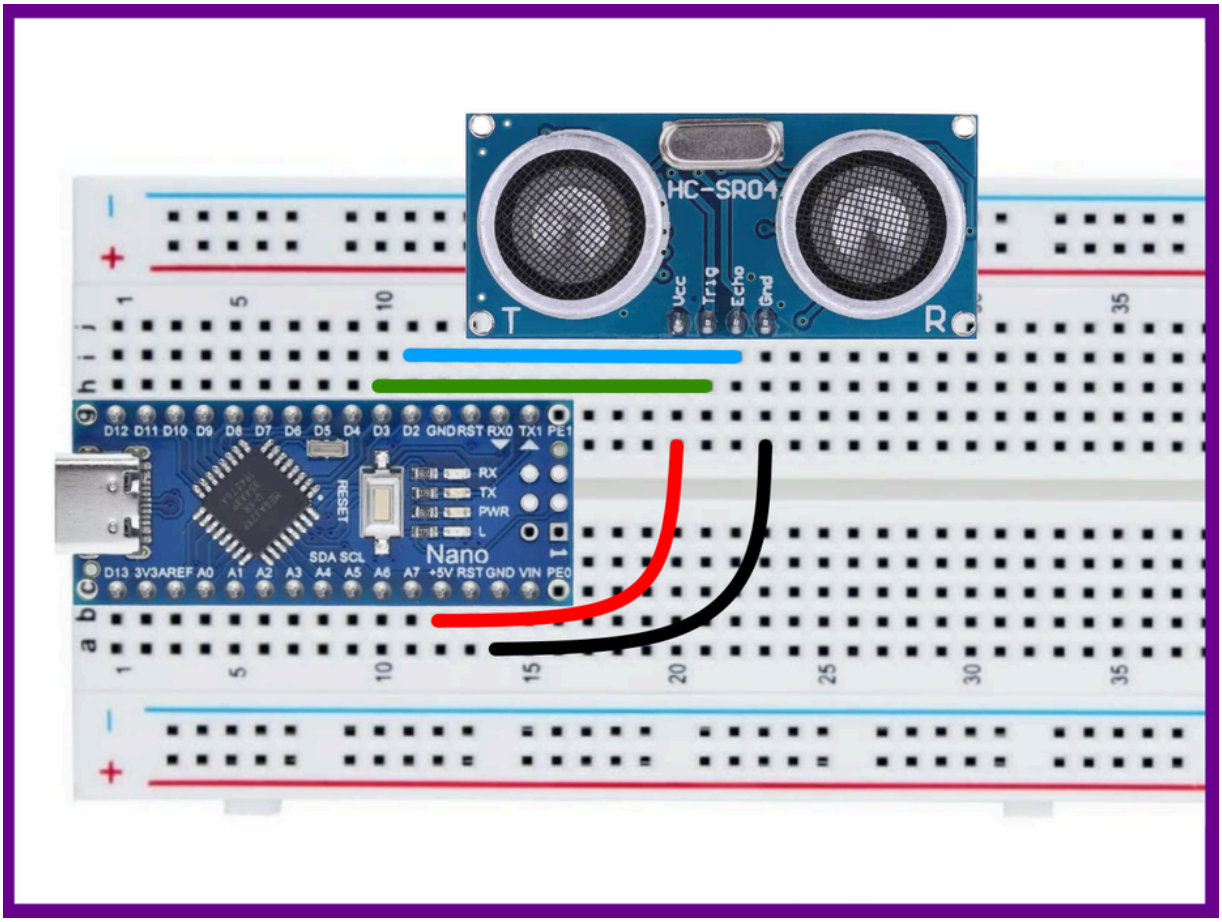
In the current example. five #define lines provide the frequencies (in Hz) for the notes to play:

```
#define NOTE_D7 2349
#define NOTE_E7 2637
#define NOTE_C7 2093
#define NOTE_C6 1047
#define NOTE_G6 1568
```

A few more #define lines specify the duration (in ms) for each note and the pause time that includes the note duration plus a little more to provide space between the notes. There is also a #define to represent the I/O where the buzzer is connected:

```
#define noteDuration 800
#define notePause 900
#define buzzerPin 8
```

Step 23: Measuring Distance



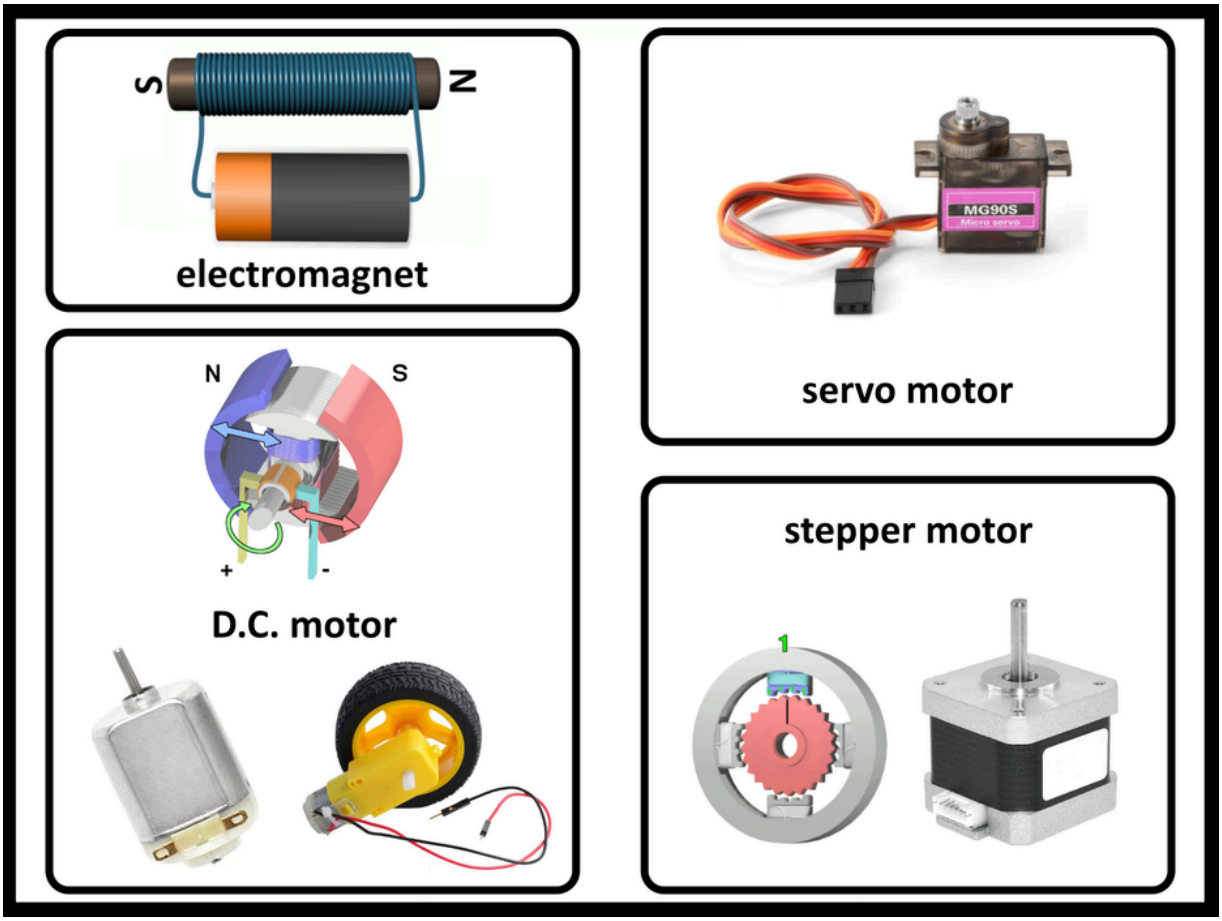
The HC-SR04 ultrasonic sensor module is like a bat. It includes a transmitter for radiating ultrasonic waves at 40kHz (40,000 Hz). The upper frequency limit of human hearing is around 20kHz so frequencies higher than that are called ultrasonic. The HC-SR04 module also includes a receiver to detect any ultrasonic waves that bounce back to it. The radiated ultrasonic waves propagate through the air and reflect off any surfaces they encounter. Some of the reflected waves bounce back to the module.

The microcontroller can measure the time between transmitting the waves and receiving the reflections. Comparing this echo time to the known speed of sound through air allows the microcontroller to calculate the distance from the sensor to the reflecting object.

Wire up the HC-SR04 ultrasonic sensor module to the Arduino Nano as shown. The trigger (trig) pin is used to tell the module to transmit its ultrasonic waves. The echo pin allows the module to tell the microcontroller when it detects the reflected (echo) of the ultrasonic waves.

Download and run the *Ultasound.ino* sketch attached here. The sketch triggers a short pulse from the ultrasonic transmitter and then measures the amount of time for a reflected pulse to be detected at the ultrasonic receiver. Since that echo time is round-trip, it is divided by two to find the one-way trip duration. Finally, the one-way time is multiplied by the speed of sound to find the distance of the echo in centimeters. The distance is displayed on the serial monitor.

Step 24: Electromechanical Motion



Electromagnets serve as the nexus between electricity and physical motion or movement.

An electromagnet is a type of magnet in which the magnetic field is produced by an electric current. Electromagnets generally have a wire wound into a coil around a core. When electrical current flows through the wire, such as from a battery, a magnetic field is created allowing the core to act much like a permanent magnet. However, the magnetic field disappears when the electrical current is not flowing through the wire.

The magnetic field can be turned on, off, or reversed using electricity. The magnetic field can be used to move (attract or repel) other physical objects or structures. Accordingly, physical motion can be generated, stopped, and reversed under electrical control.

Direct current motors (DC motors) are rotary motors that can be found in toys, tools, and appliances. The small, silver DC motor in the illustration is shown on its

own and as part of a yellow gearbox with an attached wheel. Such motors are often used in toys and hobby projects.

DC motors have an arrangement of coils (electromagnets) and permanent magnets that can convert electrical current into rotational motion. DC motors have structures that periodically change the direction of current in part of the motor thereby allowing the motion to spin around in a circle as the current changes.

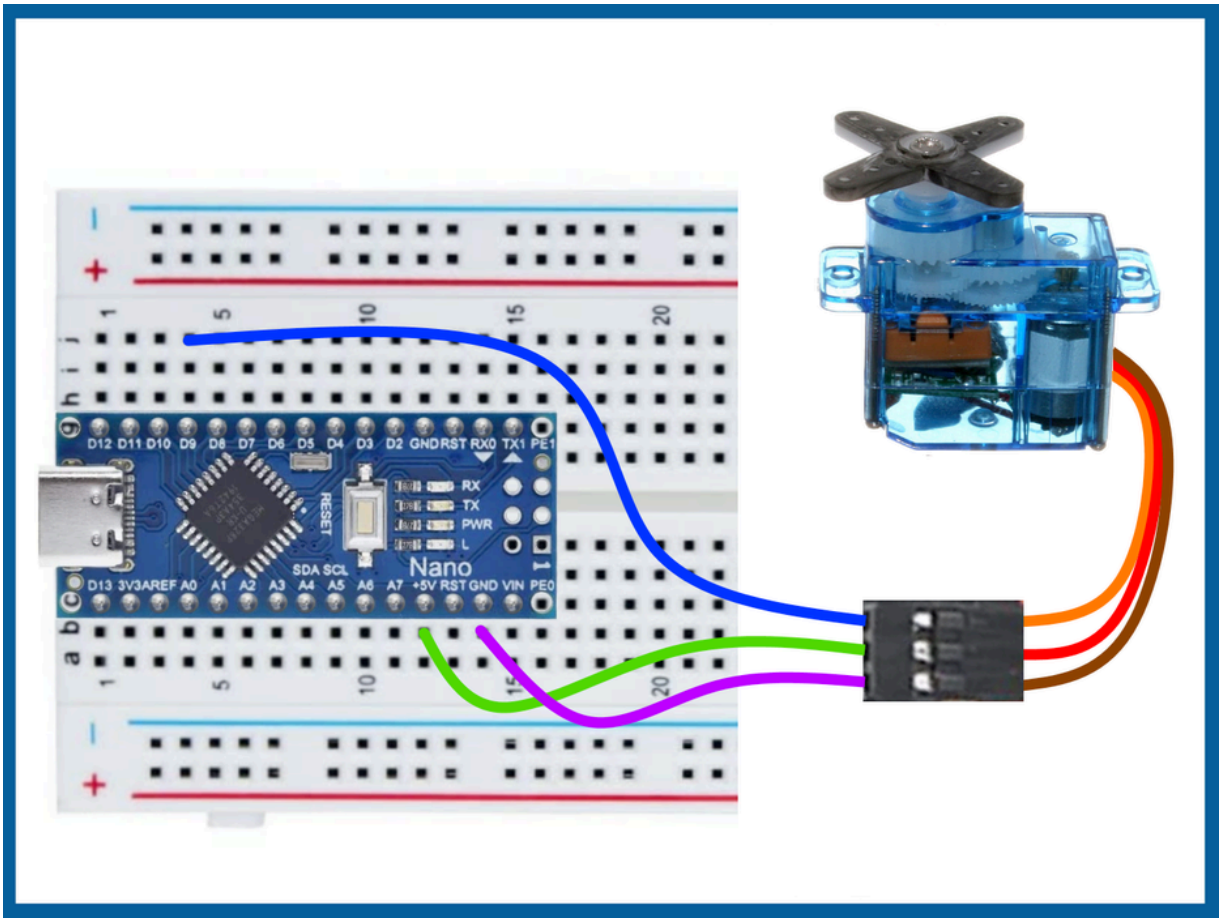
The diagram of a simple DC motor shows a stationary set of magnets around the outside called the stator. The diagram also shows windings of wire in the center forming electromagnets. This center can rotate and is called the rotor. The connections in front of the rotor cause the current flow to change as the motor spins and forms a structure called the commutator.

Servo motors are simple motors coupled to closed loop control mechanisms that are often built into each motor. The control mechanism includes a controller and a sensor for position feedback. A servo motor can usually turn automatically to any angle instructed by the controller. Servo motors are used in applications such as robotics, model planes, and CNC (computer numerical control) machinery common in manufacturing.

Stepper motors have multiple notched or toothed electromagnets arranged as a stator around a central rotor. Each full rotation of motion is divided into a number of equal steps related to the spacing of the notches or teeth in the electromagnets. The motor position can be commanded to move to one of these steps. The electromagnets are energized by a control circuit that is usually external to the stepper motor.

Unlike servo motors, stepper motors generally employ open-loop control. The motor itself does not incorporate a position sensor for feedback, so the motor does not "know" where it is. This state information must instead be collected and maintained by the control electronics connected to the stepper motor. Stepper motors in your scanner or printer usually have to move to one end of their motion range to reset their position every time the system is powered up. You are probably accustomed to hearing this startup process occur and now you know why.

Step 25: Controlling Servo Motors



The Arduino IDE includes a built-in Servo Library capable of controlling multiple servo motors making careful use of timing mechanism within the microcontroller. The library can control 12 different servos using only only timer.

Wire up the servo motor to the Arduino Nano as shown using the wire harness built into the servo and three male-to-male jumper wires. The servo wiring harness has three color-coded lines: 5V, Ground, and Signal. As shown, the Signal line connects to I/O pin 9 of the Arduino Nano. Pulses on the Signal line instruct circuitry within the servo to move the motor shaft to different angular settings.

It is useful to push one of the included servo attachments onto the output gear of the servo to make movements of the servo easier to see.

Open the sketch:

File > Examples > Servo > Sweep

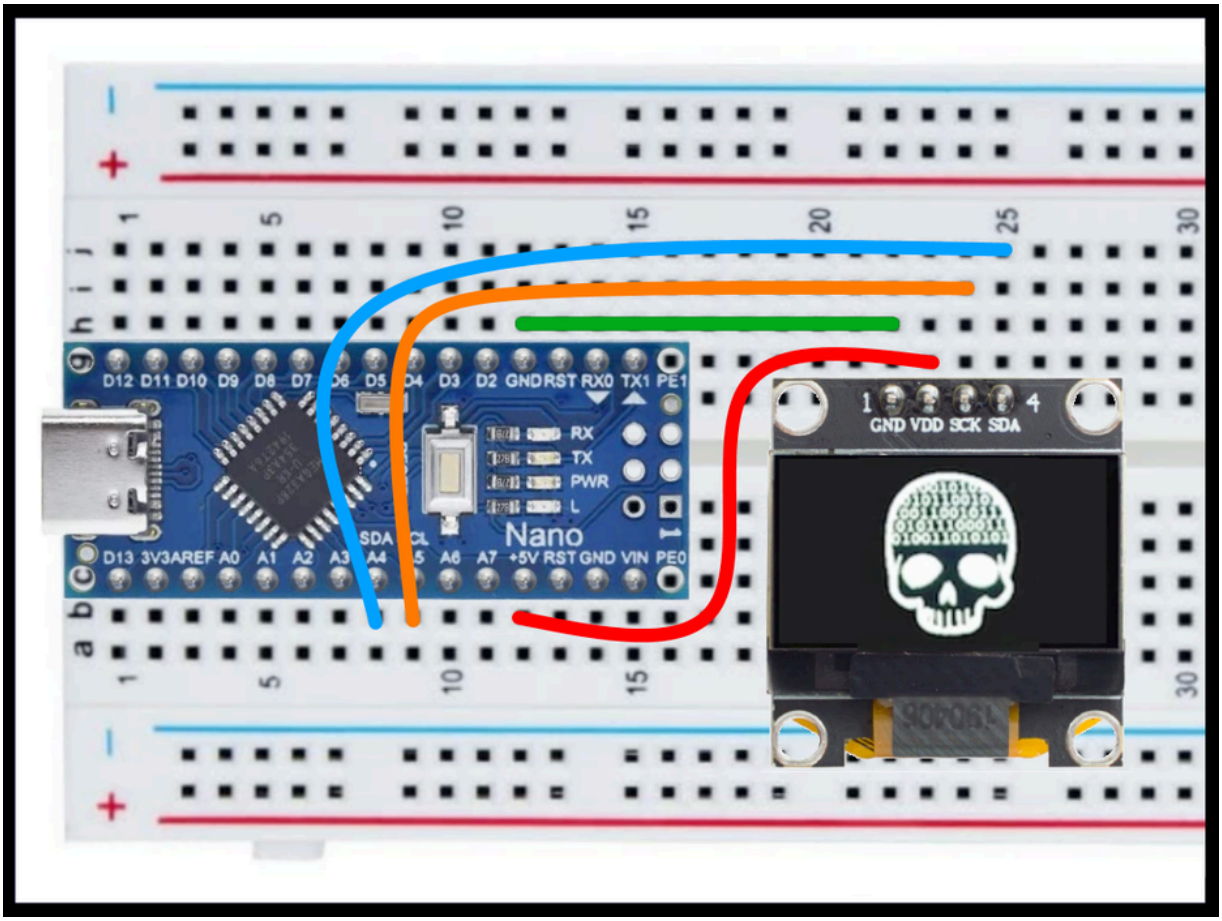
Download and run the sketch on the microcontroller. As you can probably guess from looking at the sketch code, it will sweep the shaft of the servo motor from 0 to 180 degrees and then back again.

What do you expect if you change the loop() function to contain only:

```
myservo.write(random(180));
delay(5000);
```

How could this be used to replace rolling dice or using a "spinner" on a board game?

Step 26: Displaying Graphics and Text



The OLED display module measures a tiny 0.96 inch but has a resolution of 128 X 64 pixels.

In addition to the pins for 5V Power (VDD) and GND, there are two pins for the IIC (inter-integrated circuit) bus. The IIC bus is also known as the I2S bus. The two I2C bus pins are SCK (Serial Clock) and SDA (Serial Data). The four pins for the display should be wired to the Arduino Nano as shown.

In the last step, we used a built-in library for servos. Now it is time to pull in an external library. External libraries are extremely useful for adding additional functionality to the Arduino IDE and sketches that we build within the IDE.

The external library that we will install is the **Adafruit SSD1306 Library**. The driver chip inside the OLED display module is an SSD1306 so this library is designed to allow your Arduino sketch to communicate with this driver chip.

In the Arduino IDE, navigate to **Tools > Manage Libraries**

In the window that pops up, enter SSD1306 in the search box. A few different libraries will come up in the search, so be sure to hit install under the entry for **Adafruit SSD1306**.

The installation process will ask if it should also install the dependency **Adafruit GFX Library**. Be sore to click to allow that Adafruit GFX dependency to also be installed. After this, your two new libraries will be installed into the Arduino IDE and ready to use.

Download and run the *OLEDtext.ino* sketch attached here. Notice that the sketch uses #include to invoke the new GFX and SDD1306 libraries through the Arduino IDE. Once the sketch runs as provided, try changing the settings for TextSize, cursor position, and the string being "printed" to the display.

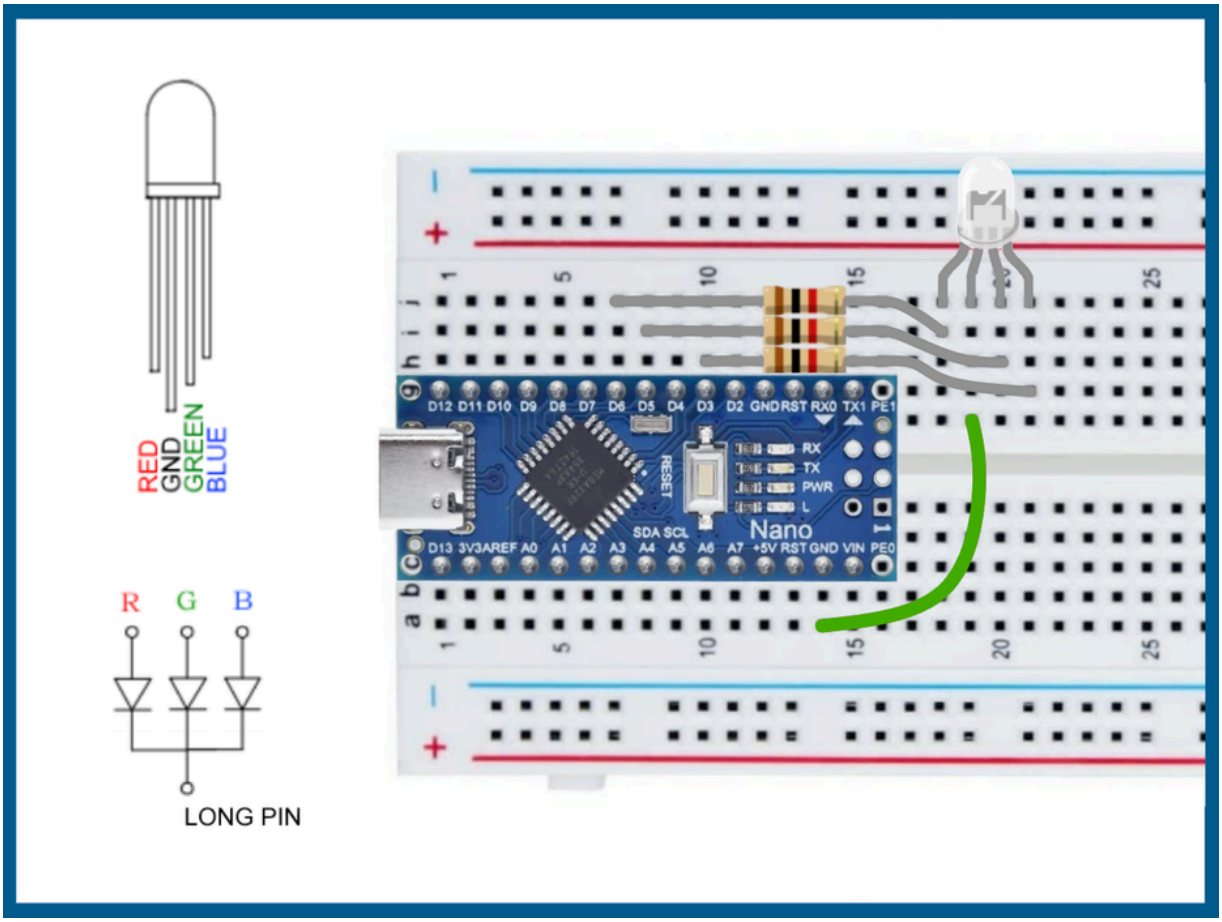
Libraries usually come with example programs for demonstrating use of the library. Give this one a try:

File > Examples > Adafruit SSD1306 > SSD1306_128x64_I2C

Once the sketch opens up, change the value in #define SCREEN_ADDRESS from 0x3D to 0x3C

Run the sketch to see a nice variety of graphics and text display examples.

Step 27: Full Color LEDs



The Common Cathode RGB LED is actually three LEDs inside of one package. The RGB stands for red, green, and blue. These are the colors of each of the three LEDs.

Recall from our earlier LED work that each LED (or any diode for that matter) has an anode terminal and a cathode terminal. The LED is forward biased, and can light up, when the higher voltage (for example +5V) is applied to the anode, and the lower voltage (for example GND) is applied to the cathode. For this reason, the anode and the cathode and often referred to as the positive and negative terminals respectively.

The three LEDs inside this one LED package have their cathode terminals connected together, which is why it is referred to as a "common cathode" arrangement. We will call this one shared cathode terminal the ground terminal for our purposes here.

The three separate anodes for the different colored internal LEDs can be driven with 5V to light up the individual LEDs as desired. Just as with the signal LEDs used earlier, a 1K resister is placed inline with each of the three colored LEDs to limit the total current flowing through the LED.

Wire up the Common Cathode RGB LED and three 1K Resistors to the Arduino Nano as shown.

Download and run the *CommonCathodeRGB.ino* sketch attached here. The sketch uses PWM to achieve the desired brightness from each of the red, green, and blue LEDs allowing the three colors to mix together to create other colors as shown with in the sketch.

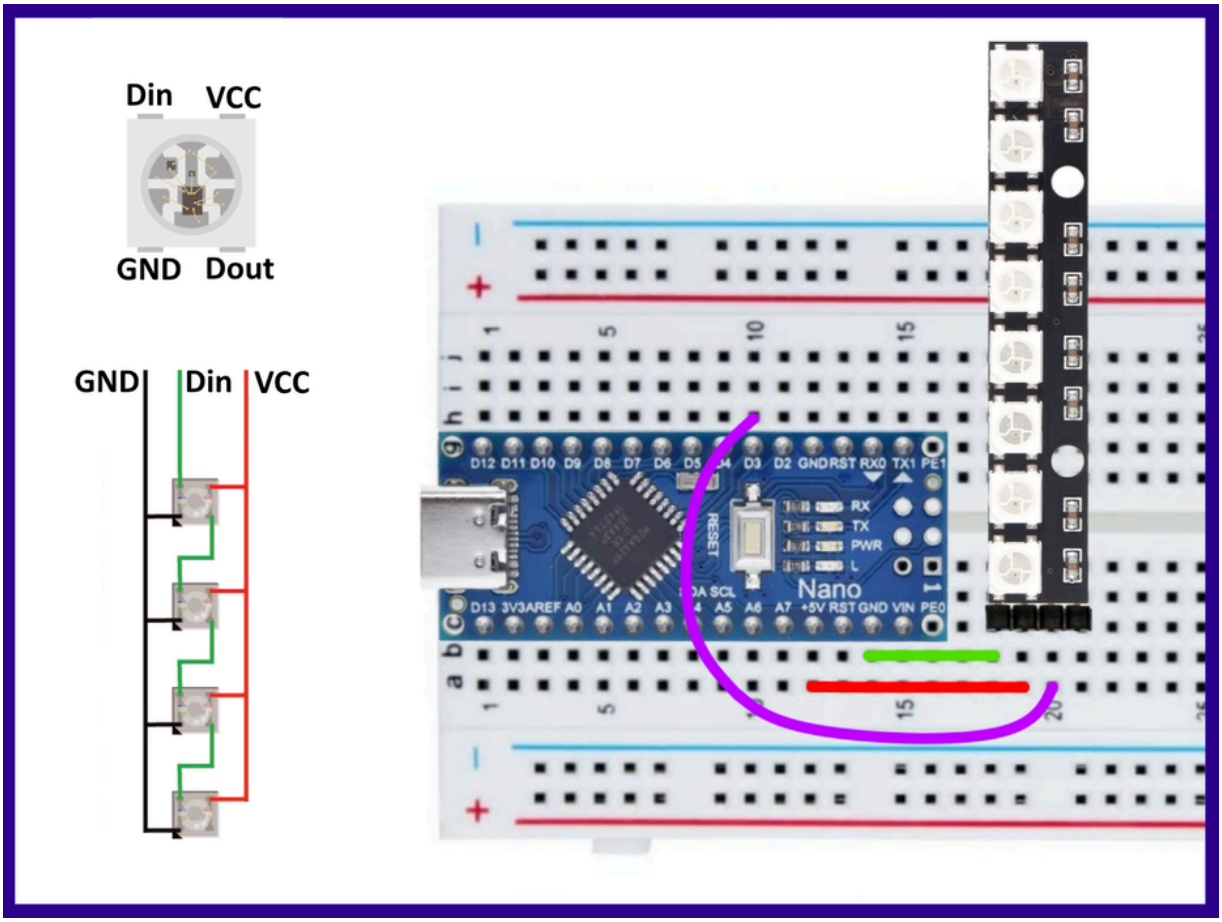
Features of I/O Pins

Recall that PWM outputs are generated by the Arduino Nano using the analogWrite() function. As shown in the function's documentation, only pins 3, 5, 6, 9, 10, and 11 of the Arduino Nano can be used for PWM. That is why pins 3, 5, and 6 are used in this experiment and pin 4 is not. It is important to check the features of specific I/O pins on an MCU when selecting which pins you will use for which purpose. You will learn that for different MCUs, certain pins may be I/O (both input and output) while some are input only and some are output only. Some pins may have specific I/O characteristics or bus applications. Some pins was be able to trigger interrupts while some may not. Some pins may have optional pull-up resistors, pull-down resistors, both, or neither. There are many options to check up on when it comes to micro controller I/O pins. Note that I/O pins are also sometimes called GPIO (general purpose I/O) pins.

Quantity of I/O Pins

Since only pins 3, 5, 6, 9, 10, and 11 of the Arduino Nano can be used for PWM and the common cathode LED requires three of those to display full colors, only two such RGB LEDs can be driven by the Arduino Nano. In additional to what specific functionality each I/O pin may have, we also have to pay close attention to how many I/O pins there are in total and how we allocate them in our project. This often forces us to figure out tricks to get additional functionality from the MCU. For example, you probably see projects with far more than two RGB LEDs all the time. Next we will look at one of the more popular ways to get there.

Step 28: Serial Addressable LEDs



One of the easiest techniques for controlling multiple RGB LEDs with one microcontroller involves serial, addressable LEDs. Certain examples of these are commonly referred to as NeoPixels or RGB Pixels.

While such a device is often called "an LED", each one actually contains 3 LEDs (one red, one blue, and one green) along with an embedded, or integrated, control circuit.

The control circuit of each device can be feed control information through one pin (data in or Din) and thus only requires one I/O pin from the microcontroller. The control information is sent from the microcontroller to the first addressable LED in a serial fashion, meaning one bit at a time - in a series. First, eight bits are sent defining the amount of green light to be emitted, then eight bits defining the amount of red light, and finally eight bits defining the amount of blue light. Those 24 bits are grabbed, or "latched", into the controller. Those 24 bits define a possible total of 16,777,216 (2 to the power of 24) different colors.

In addition to its *data in* pin, each RGB pixel also has a *data out* (or Dout) pin. The first pixel's Dout pin is daisy chained to the second pixels Din pin, and so forth until all

of the pixels are connected in a single chain. This structure allows the entire chain to be fed from a single I/O pin of the microcontroller. Once each RGB pixel latches the first 24 bits it receives, the control circuit outputs any additional bits on its Dout pin. From the Dout pin, the additional bits are sent along to next RGB pixel in the chain.

Wire up the Eight-Pixel Addressable RGB LED Module as shown

The module includes eight daisy-chained WS2812Bs devices. You can read the ES2812B datasheet from the manufacturer if you wish.

Install the FastLED Library

In the Arduino IDE, navigate to **Tools > Manage Libraries**

In the window that pops up, enter FastLED in the search box. A few different libraries will come up in the search, so be sure to hit install under the entry for FastLED by Daniel Garcia.

The FastLED library comes with some nice example programs. Load this one for now:

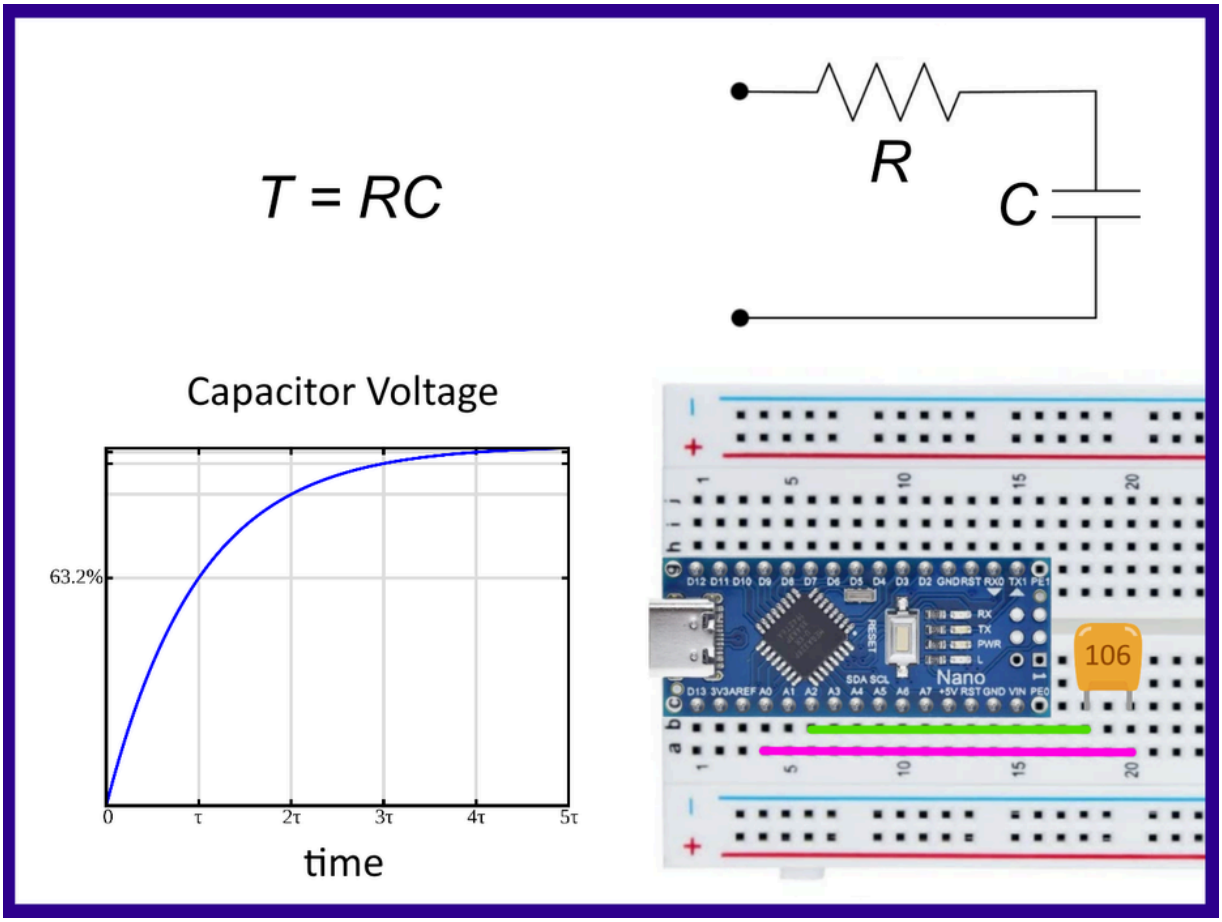
File > Examples > FastLED > DemoReel100

Once the sketch opens, change two defines to match these:

```
#define LED_TYPE  WS2812B
#define NUM_LEDS  8
```

Run the sketch. After enjoying the results, play around with the sketch to see what fun can be had with these serial, addressable LEDs. As you can see, they are extremely versatile.

Step 29: Measuring Capacitance



A capacitor is a two terminal device capable of storing energy in an electric field. A capacitor can be thought of as a very fast rechargeable battery where energy is stored in an electric field instead of as chemical energy. Since an electric field can be generated an discharged rapidly, the capacitor operates on a much faster time scale than a chemical battery.

The simplest structure for a capacitor is two parallel conductive plates. Between the plates, there is usually a nonconducting dielectric such as ceramic, glass, plastic, paper, mica, or air. Alluding to that parallel plate structure, the schematic symbol for a capacitor is two parallels lines.

The effect of a capacitor is known as capacitance and is measured in the unit Farads. One Farad is huge, so practical capacitors are usually measured in pico Farads (10 to the power of -12 Farads), nano Farads (10 to the power of -9 Farads), or micro Farads (10 to the power of -6 Farads).

We will measure a ceramic capacitor with marking "106" which equals a capacitance value of 10µF. The marking represents a value in picofarads starting with two digits

"10" followed by the multiplier factor 6 (ten to the power of six) or 1,000,000. Giving us the value of 10 * 1,000,000pF or 10µF.

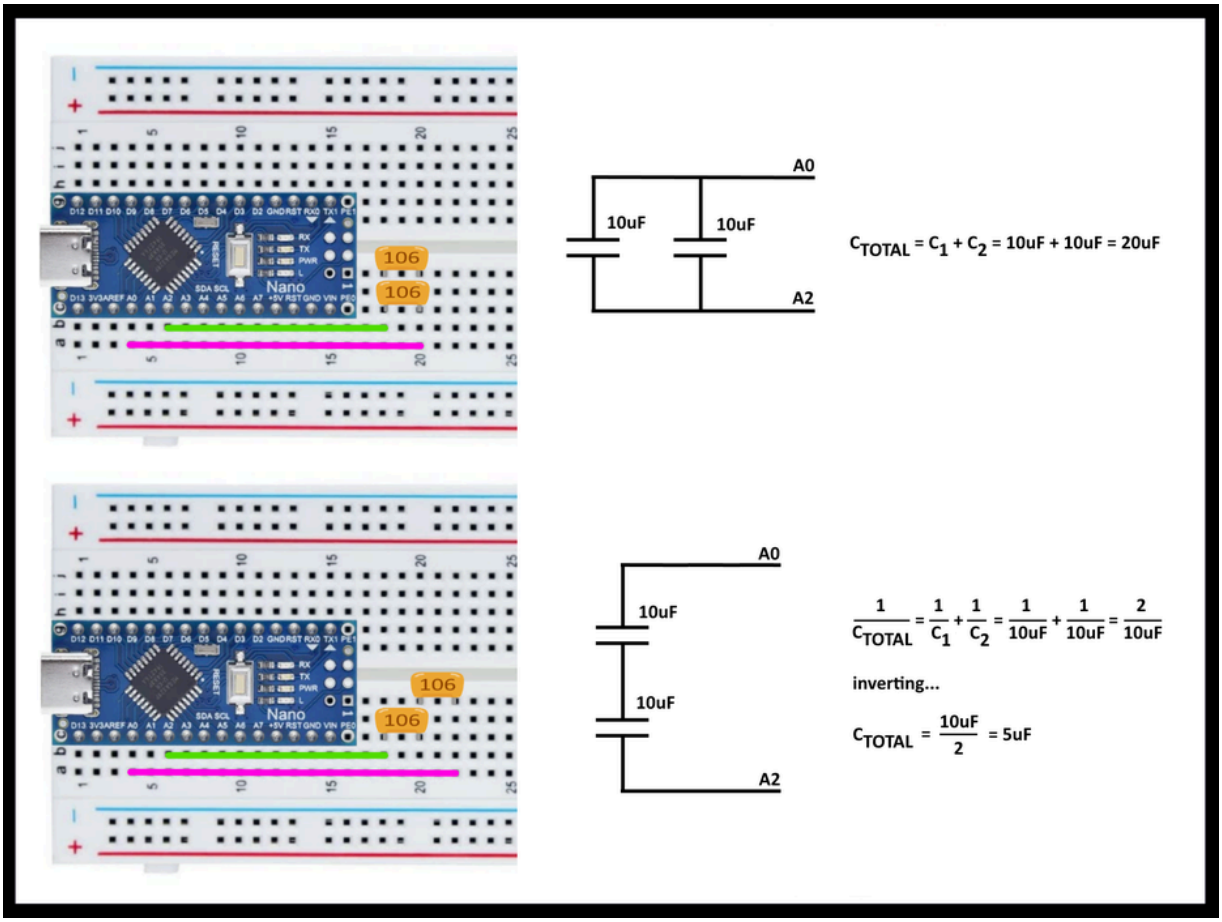
When a voltage is applied across a capacitor, the electric field within the capacitor is charged up and energy is stored with the capacitor. This energy can then be discharged (used up) out of the capacitor. How rapidly the capacitor charges or discharges can be used to calculate the value of the capacitor.

Microcontroller code can measure the time required for charging or discharging the capacitor to calculate the capacitance between two pins. Wire the 10uF ceramic capacitor between pins A0 and A2 of the Nano as shown.

Download the attached *Capacitance.ino* sketch file and program it to the Arduino Nano. Open the Serial Monitor to view the output printed over the Nano's serial port. Notice how the capacitors marked as 10uF capacitors will have measurements varying from about 8 to 11 uF. This is normal for the type of capacitor we are working with.

How does this timing work? According to basic physics, it requires one *time constant* to charge a capacitor from zero up to 63.2% of the applied voltage. In this case, that would be 63.2% of 5V. But what is the *time constant*? It is just R*C, where R is the resistance of the circuit (in Ohms) and C is the capacitance (in Farads). It may look like there is no resistor in the circuit, but in fact the microcontroller's internal pullup resistor (having approximately 34.8 Ohms) is used to charge up the external capacitor. The code in the sketch is a little complicated, but you can certainly explore how the RC time constant and the known pullup resistance of the microcontroller are used to make the measurement calculations.

Step 30: Capacitor Structures



Modify the previous capacitor circuit by combining two 10uF capacitors in parallel and then in series as shown here. In each instance, observe the output of the serial monitor to measure the equivalent capacitance of the combined capacitors. Remember that the capacitors are not exactly 10uF even though that is how they are marked.

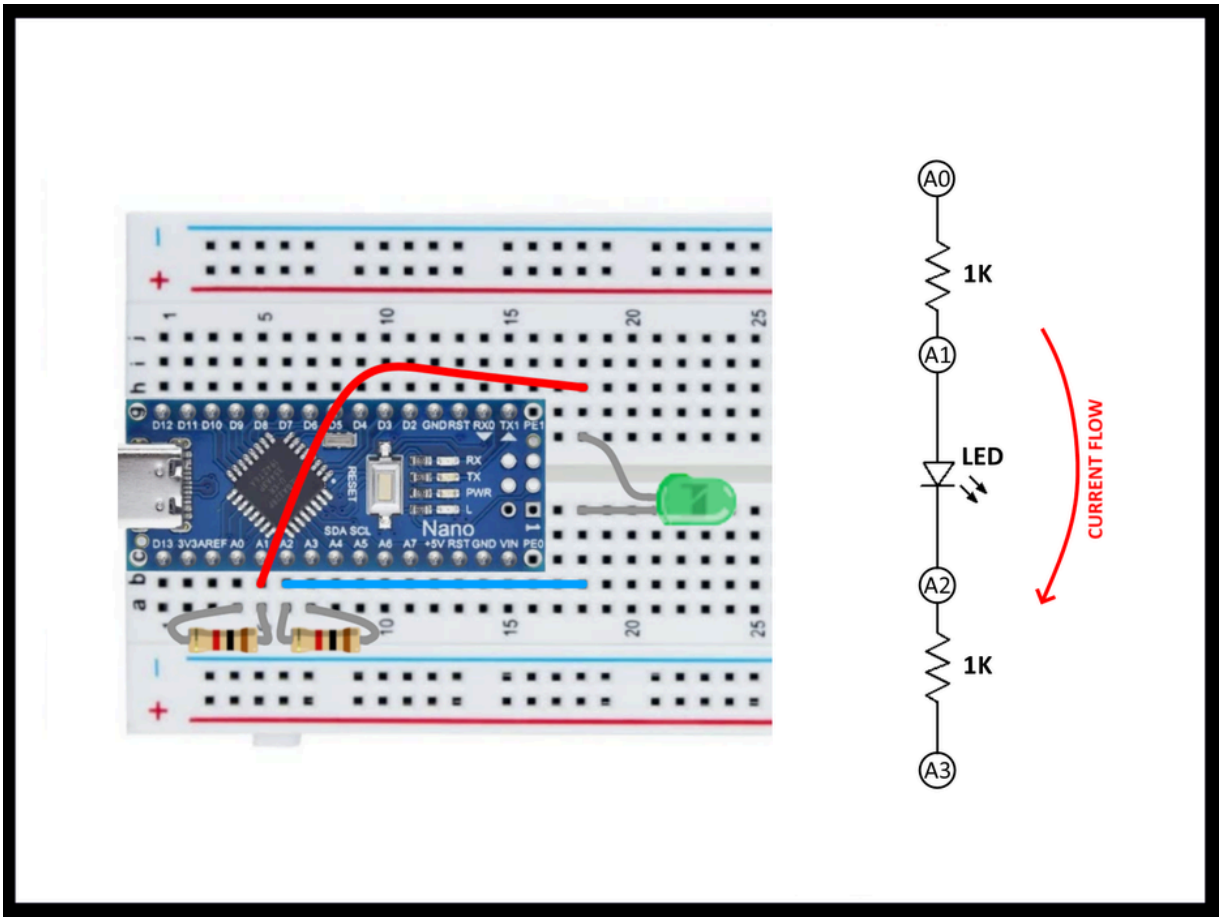
Revisit the results found in **Step 15: Resistor Structures**. Notice how resistors and capacitors combine in similar, but opposite, fashions. Serial resistors add together while parallel capacitors add together. The equivalent resistance of parallel resistors is the inverse of the sum of the inverse of the individual resistors. Similarly, the equivalent capacitance of series capacitors is the inverse of the sum of the inverse of the individual capacitors.

Consider delving deeper into capacitor structures by combining all three of the 10uF capacitors in different ways.

For an advanced exercise, measure the capacitance of each capacitor alone and then plug those actual individual capacitances into the parallel and serial equivalence

formulas to see how close the combined measurements are to theory.

Step 31: Electron Flow Through Diodes



Semiconductor diodes are like one-way valves. They only allow current to flow in one direction, and not in the opposite direction. This is true for all types of Diodes, including LEDs (Light Emitting Diodes).

Wire up two 1K resistors and an LED as shown in the diagram.

Download the attached *DiodeTest.ino* sketch file and program it to the Arduino Nano. Open the Serial Monitor to view the output printed over the Nano's serial port.

With the LED wired as shown (long pin to A1 and short pin to A2), the serial monitor will indicate that current flows from A1 to A2 but current cannot flow from A2 to A1.

With the direction of the LED swapped, current will flow from A2 to A1, but not from A1 to A2.

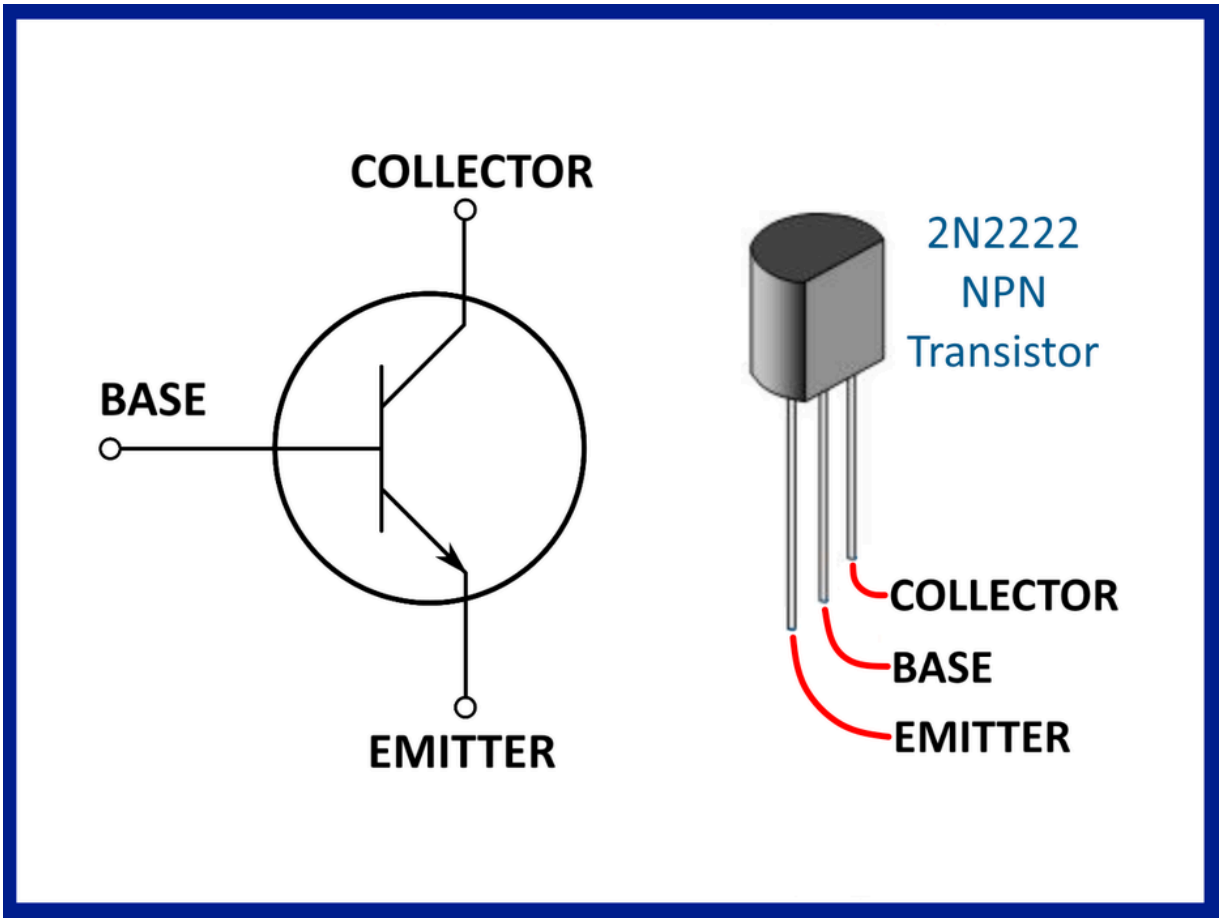
With the LED removed, current will not flow in either direction.

With A1 connected directly to A2, current will flow in both directions.

Carefully examining the sketch code will reveal how two additional pins (A0 and A3) are used to control 1K resistors connected (respectively) to A1 and A2. The first test in the code sets A3 to ground which means the 1K resistor connected to A2 is grounded (pulling low). Then the code configures A0 as an input meaning that A0 is not driving high or low, but instead is floating. With A0 floating, the resistor connected to A1 will not be pulled high or low. Pin A1 is however connected directly to 5V. So one side (A1) of the DUT (device under test) is set to 5V and the other side (A2) of the DUT has a 1K resistor to ground and can also be read as an analog input to the microcontroller.

Depending upon what is between A1 and A2, current will either not flow (keeping the original 5V difference between A1 and A2). However, if current is flowing through the DUT, it will also flow through the 1K resistor since they are in series. The voltage drop in the 1K resistor will make the difference between A1 and A2 less than 5V. Sensing this lower voltage difference allows the code to identify that current is flowing. This trickery probably starts out sounding a lot more complicated than it really is. Stepping through the measurement process several times or until everything clicks is a worthwhile undertaking if you're up for it.

Step 32: Transitors

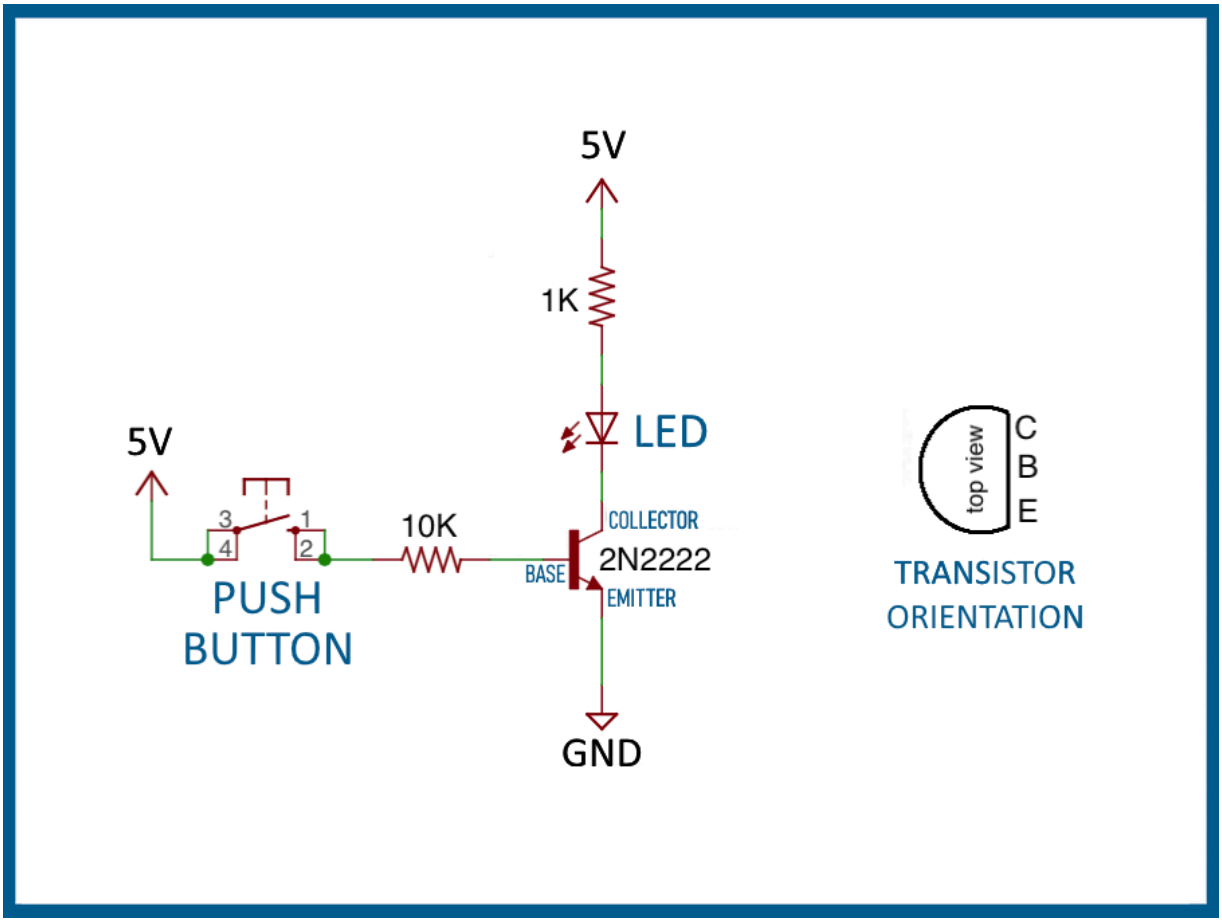


A transistor is a semiconductor device generally having three terminals. Transistors are capable of switching or amplifying electrical signals. An input signal at a first pair of the terminals can switch or amplify the signal passing through a second pair of terminals. We will be starting with 2N2222A NPN Bipolar Transistors. The three terminals of a bipolar transistor are called base, collector, and emitter.

In addition to NPN transistors, there are also PNP transistors. In addition to bipolar transistors, other common transistors include field effect transistors (FETs), metal oxide semiconductor FETs (MOSFETs), and complementary metal oxide semiconductor (CMOS) transistors. The three terminals of all these types of FETs are called gate, source, and drain.

Individual transistors, like the 2N2222, only have one transistor in a three-pin package, but modern electronic devices often pack many, many (even billions) of transistors into an integrated circuit.

Step 33: Transistors As Switches



A 2N2222 transistor can be used as a switch. This circuit is very much like the earlier circuits, "Control Electron Flow With A Switch" and "Control Electron Flow With A Pushbutton". Instead of a mechanical switch or button, the collector-emitter path through the transistor is used to open and close the path for electron flow through the LED.

Build this circuit on the breadboard using one transistor, one push button, two resistors, and an LED.

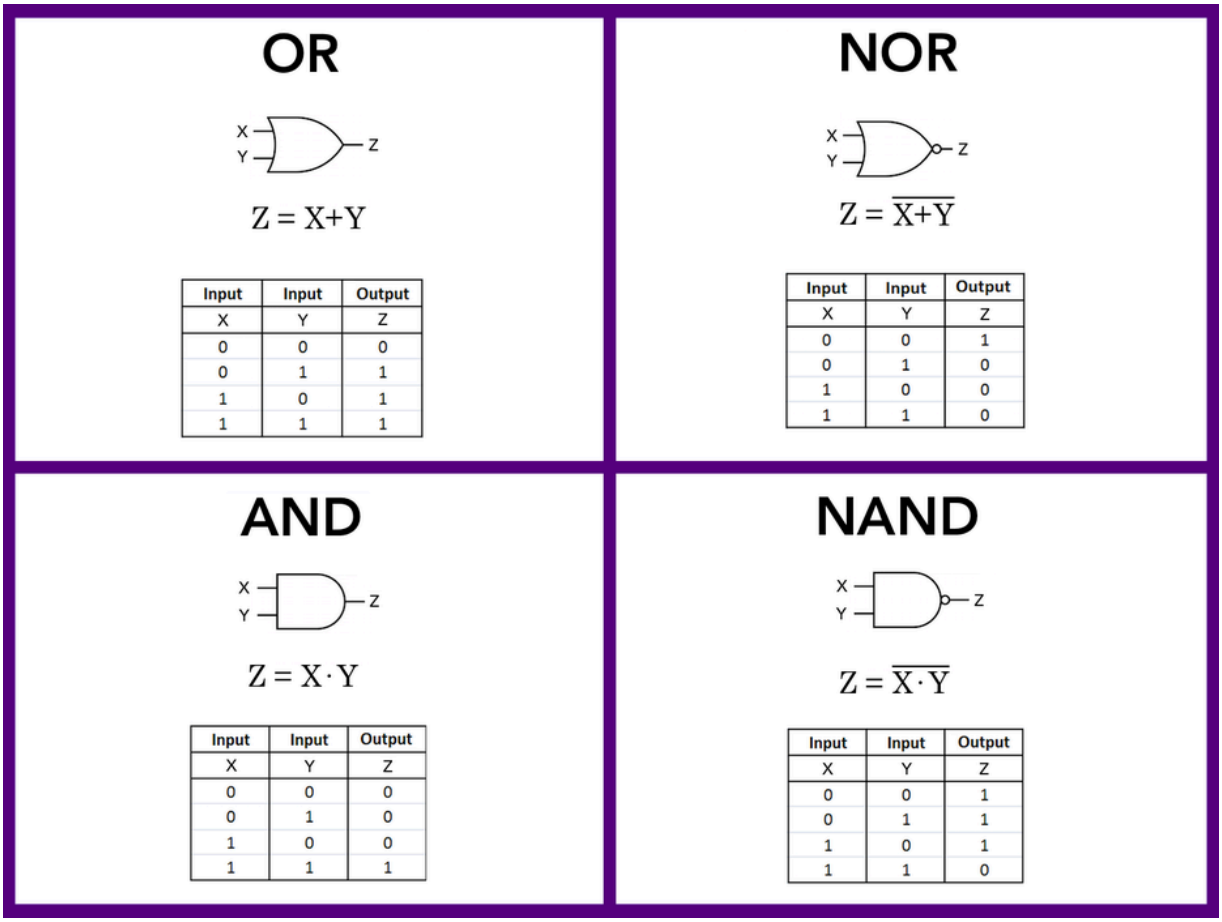
When the button is closed, a 5V signal is applied between the base and the grounded emitter (Vbe = 5V). This "control signal" forces Vce (the voltage from collector to emitter) to zero such that the transistor acts like a short or closed switch. This allows current to flow through the LED causing it to light up.

When the button is open, Vbe = 0V, and the transistor acts like an open switch and current does not flow through the LED. The LED remains unlit. These two lit/unlit conditions show how the control signal at the base turns the transistor on and off like a switch. Notice that the LED draws current through the 1K resistor and not through

the 10K resistor. Only a very tiny amount of current comes through the 10K resistor to activate the transistor "switch". This illustrates how we can control a large amount of current with quite a small amount of current using a transistor as a switch.

This circuit can be called a buffer or pass-gate. From a logic (HIGH / LOW) perspective, the circuit's output matches (buffers, or passes) its input. The opposite logic element is called an inverter or NOT gate. The NOT gate outputs a 1 (HIGH) when its input is a 0 (LOW) and vice-versa, which implements logical negation. The NOT circuit is illustrated in Step 5 of the HackerBox 0039 Box Guide.

Step 34: Digital Logic



Many modern electronics systems are based on digital (also known as, binary or Boolean) logic.

Here we see four of the most common logical operators: OR, NOR, AND, NAND. Each one is shown in three different common representations: schematic symbol, Boolean logic expression, and truth table.

The output of the OR operator is true when either its first input is true OR its second input is true. The logical OR is an "inclusive OR" meaning that the output is true also when both inputs are true.

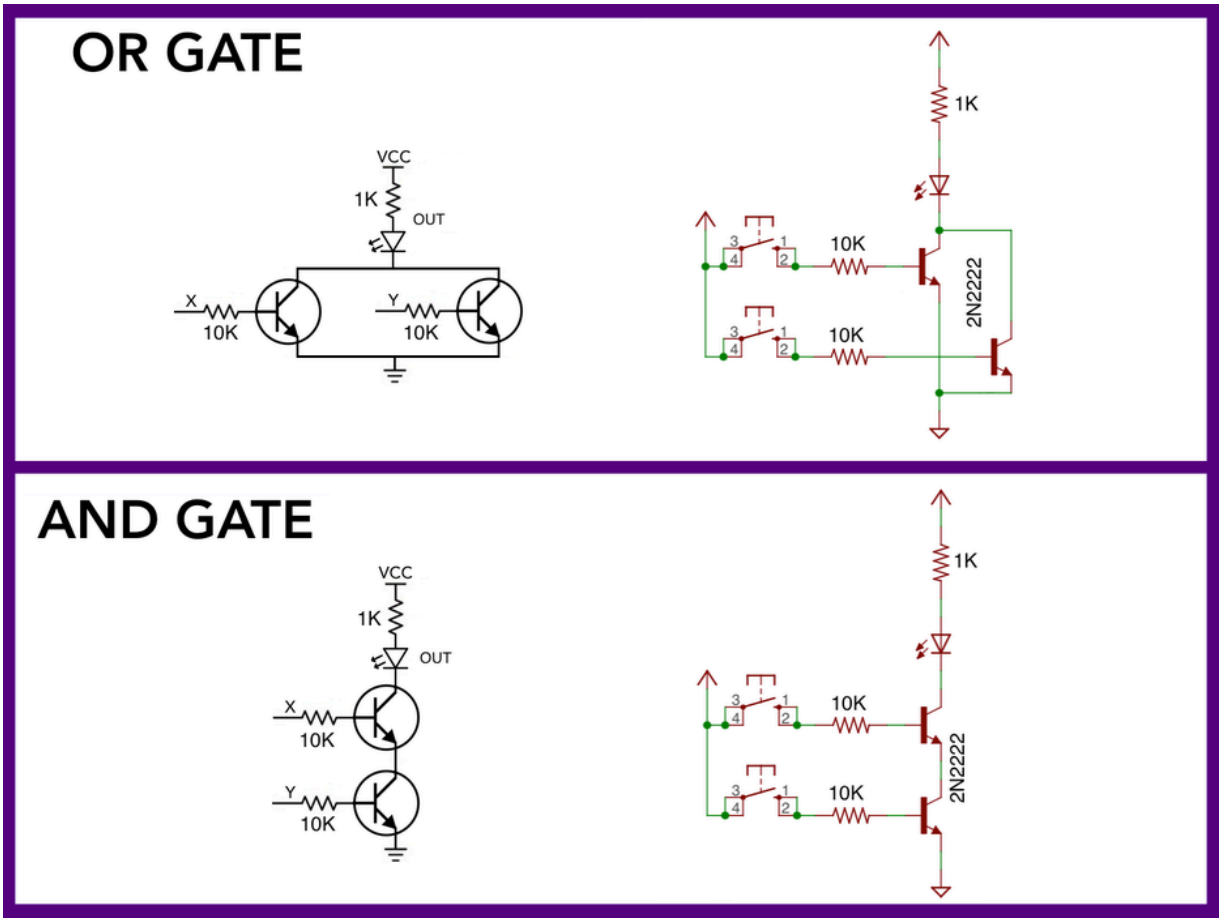
The output of the NOR operator is simply NOT OR, which is to say the logical opposite of the OR operator.

The output of the AND operator is true only when both its first input is true AND its second input is true.

The output of the NAND operator is simply NOT AND, which is to say the logical opposite of the AND operator.

Two additional logic operators worth becoming familiar with are XOR and XNOR.

Step 35: Logic Gates From Transistors



Electronic logic gates can be constructed from transistors. Each transistor acts as a switch as we saw in our last experiment.

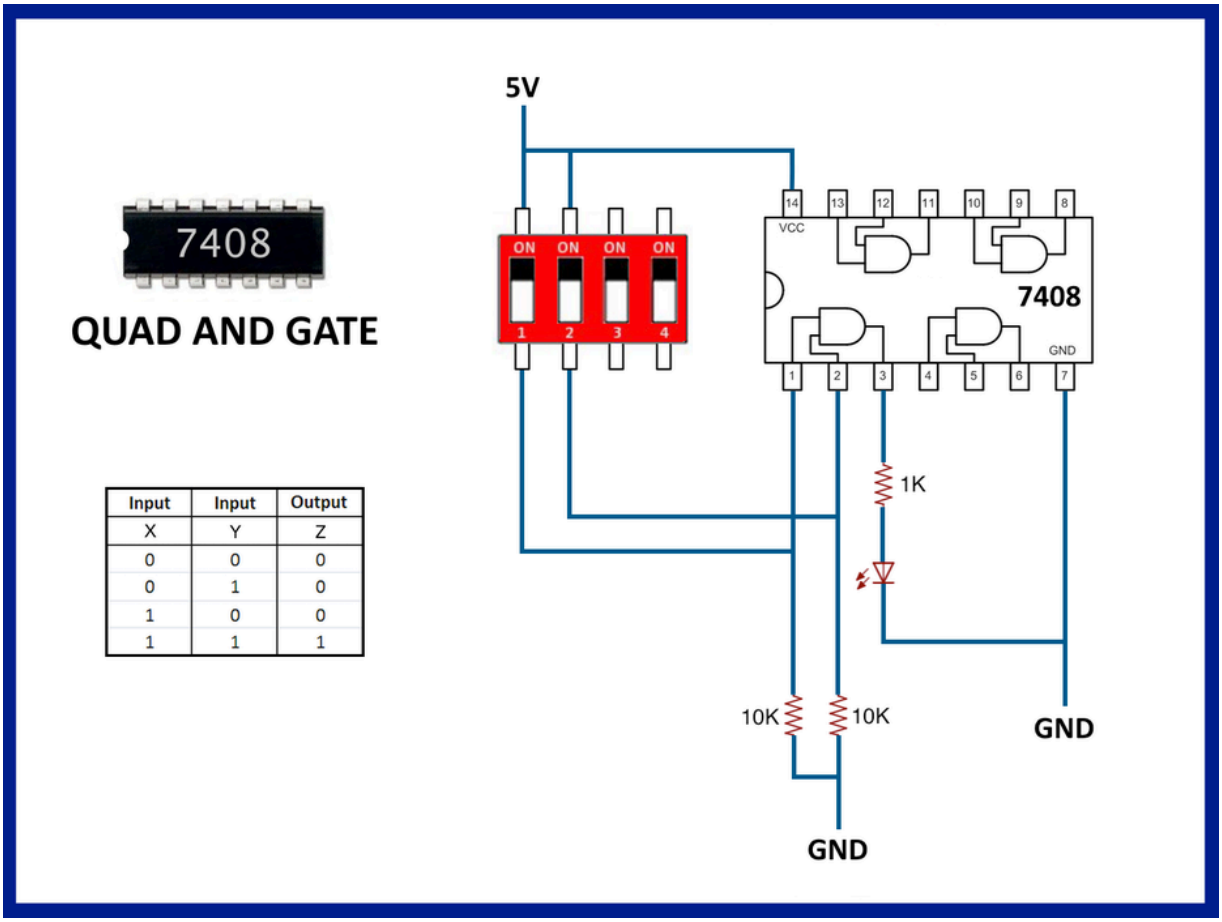
An OR GATE is formed from two transistor switches arranged in parallel. Given this parallel form, the gate is ON (or TRUE) when either the first input is ON (or TRUE) **OR** the second input is ON (or TRUE). In parallel, either transistor conducting will allow current to flow.

Build this circuit on the breadboard using two transistor, two push buttons (inputs), three resistors, and an LED (output). Compare its operations with the truth table for the logical OR operator.

An AND GATE is formed from two transistor switches arranged in series. Given this series form, the gate is ON (or TRUE) only when both the first input is ON (or TRUE) **AND** the second input is ON (or TRUE). In series, both transistors much conduct to allow current to flow.

Build this circuit on the breadboard using two transistor, two push buttons (inputs), three resistors, and an LED (output). Compare its operations with the truth table for the logical AND operator.

Step 36: Integrated Logic Chips



An integrated circuit (commonly referred to as a chip) generally contains a large number of transistors integrated into a single device. There are integrated circuits for performing all manner of electronic feats including microprocessors, audio amplifiers, network interfaces, cryptographic engines, graphics processors, flash memory, and on and on.

In the 1960s, a whole series of digital integrated circuits became available with many of the initial chips implementing logic gates. For example, the 7408 chip is a Quad AND Gate, which means that the chip contains four individual AND logic gates as shown here. All of the gory details of the 7408 chip can be seen in its datasheet from Texas Instruments.

The illustrated circuit demonstrates the use of one of the AND gates within a 7408 chip. The circuit can be assembled on the breadboard using a 4-bit DIP switch (for the two inputs), three resistors, and an LED (for the output).

What are the resistors for?

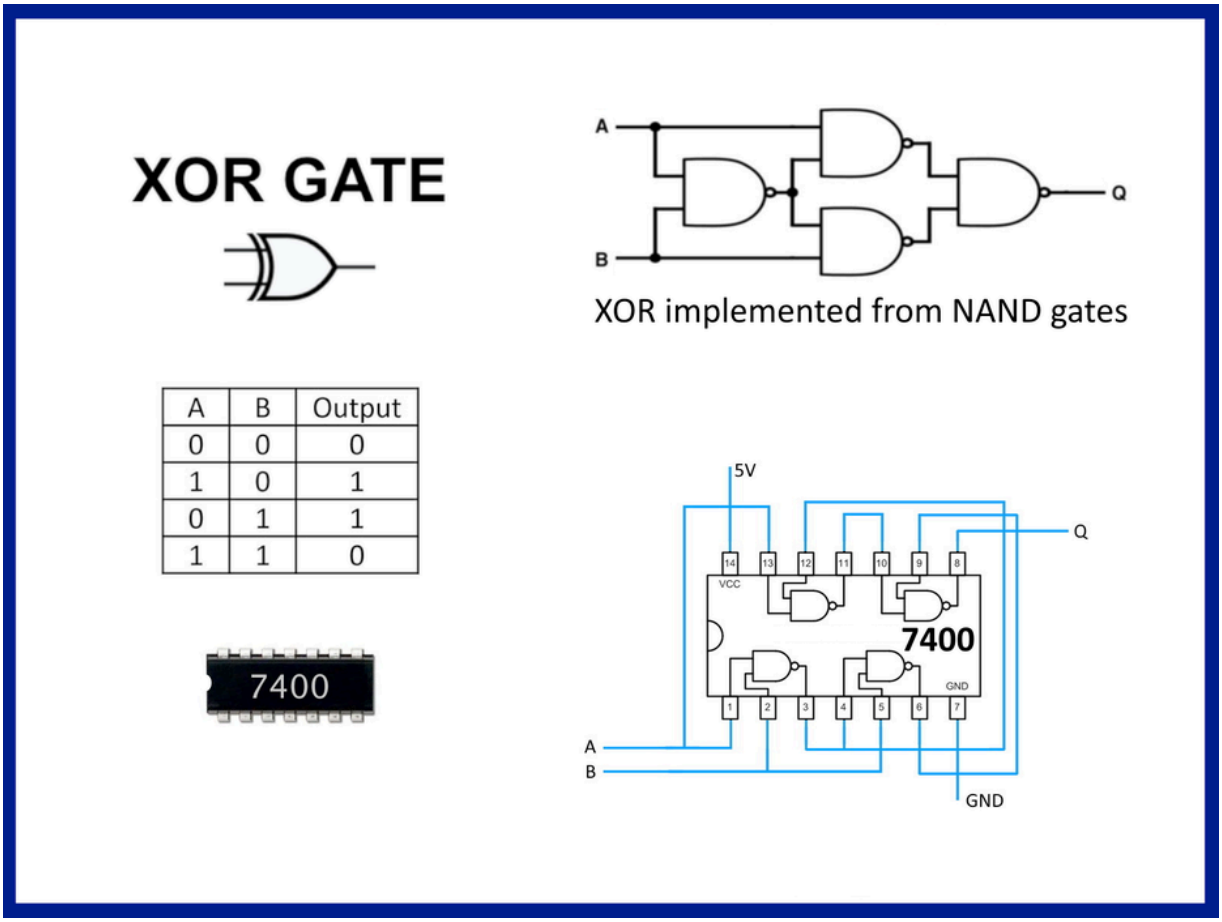
The 1K resistor is a current limiting resistor to keep the LED from drawing too much current. It is the same "current limiting" resistor application we've used in earlier experiments.

The two 10K resistors are "pull-down" resistors that gently set the logic inputs at pins 1 and 2 to low (or 0V) when the respective input switch is open. An open signal is also referred to as "floating" as it can float to any voltage level in a nondeterministic fashion. Since a floating input can take on many different values, it provides an unknown input, which is obviously not good. Gently pulling the line down to 0V makes each input zero instead of floating. Since 10K is a pretty high resistance (very unlike a direct short), we can think of that as gently pulling the voltage level. Then when the switch is closed, the line is very firmly (by a direct short) connected to high (or 5V) which easily overrides the gentle pull-down. So the 10K pull-down resistor lets a single switch provide both a HIGH and LOW input even though the switch is only really connected to HIGH (5V).

Other Gates

Assemble similar demonstration circuits for an OR gate using the 7432 chip (datasheet) and for a NAND gate using the 7400 chip (datasheet). Verify the correct operation against the expected truth table for each logic operator.

Step 37: XOR Implemented From NAND Gates

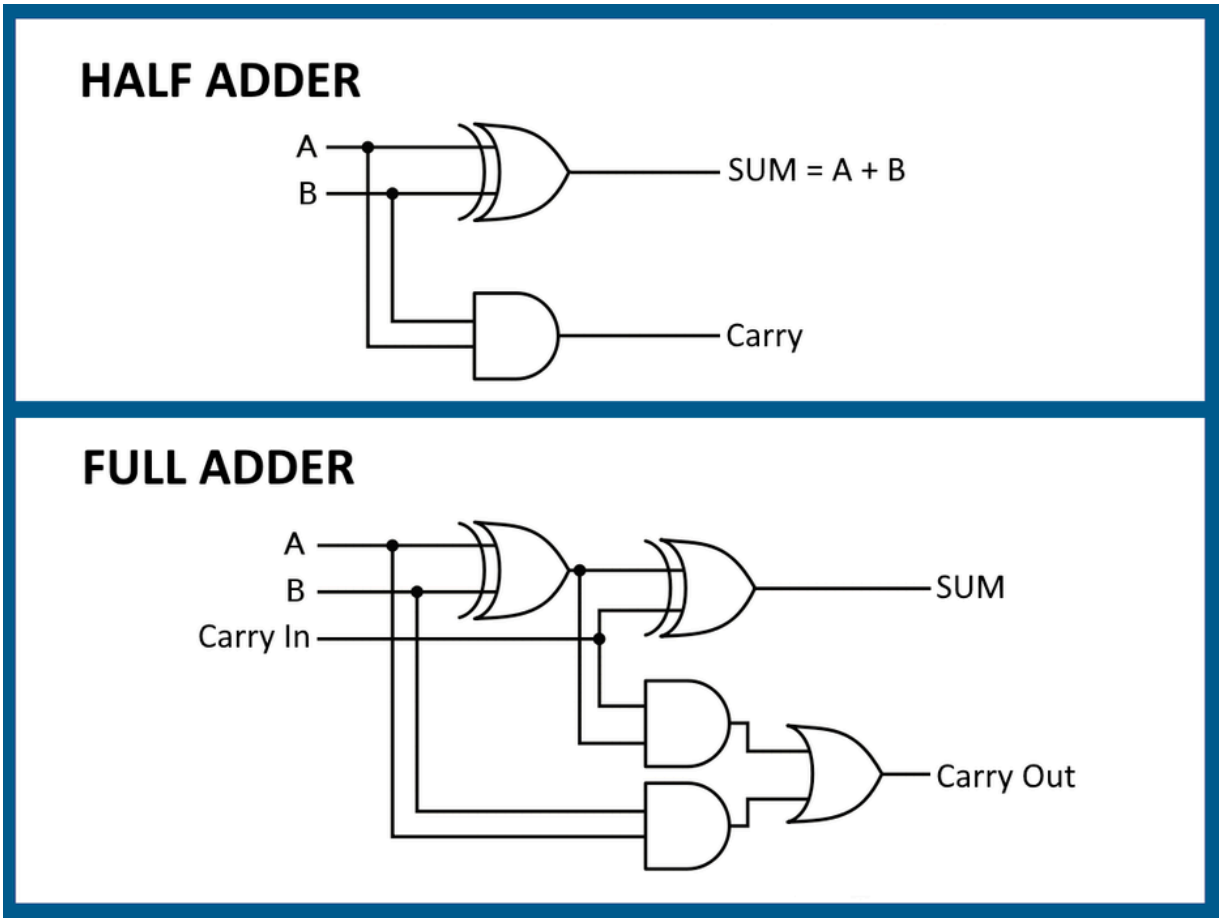


As mentioned earlier, the logical OR operation is an "inclusive OR" meaning that the output is true also when both inputs are true. The "exclusive OR" (or XOR) excludes the condition were both inputs are true. This is demonstrated in the truth table shown here.

The XOR logic can be implemented by combining all four of the NAND gates of a 7400 quad NAND chip (datasheet). The XOR circuit can be assembled on the breadboard in a very similar fashion to the previous AND gate circuit. A 4-bit DIP switch (for the two inputs A and B) and two 1K pull-down resistors feed input pins 1, 2, 5, and 13 as shown. A 1K resistor and an LED are connected at pin 8 to display the output (Q).

Note that we suggest 1K pull-down resistors here, not the 10K used in the AND circuit. Feel free to try it both ways, but you will likely find that 1K works better.

Step 38: Combining Logic Gates



We've combined transistors to construct gates, now let's combine gates to do some math.

Combinational Logic generates outputs based only on the present inputs. In fact, combinational logic is sometimes referred to as time-independent logic because it has no memory. Later we will see how sequential logic can compute outputs based on present inputs and also on history.

Mathematical operations are generally combinational (memoryless or stateless). When you are multiplying two numbers together, it doesn't really matter what numbers you multiplied together yesterday or who won the last world cup.

Half Adder

The half adder illustrated here is quite simple. It only adds one bit to another bit (labeled A and B).

Consider the possible outcomes:

```
A   B   SUM
0   0     0
0   1     1
1   0     1
1   1    10
```

Notice that the lowest bit of the SUM is just an XOR and the higher bit is only high when both A and B are high.

We call the low bit SUM and generate it using one XOR gate.

We call the high bit CARRY and generate it using one AND gate.

Consider how we add two base ten (decimal) numbers. We need to use a carry when two digits sum up to 10 or more because the value overflows into the next higher digit. In base two (binary), we need to carry when two digits sum up to two or more, which is also 10 in binary. In binary the value two (written as 10) overflows into the next higher digit. Carrying is the same concept in base two as it is in base ten.

The half adder can be constructed on the breadboard in a similar fashion to the gate circuit exercises: Use the 4-bit DIP switch for the two inputs (A and B) along with two 1K pull-down resistors. Implement the XOR gate by combining all four NAND gates of the 7400 quad NAND chip. Use one AND gate of the 7408 chip. Finally, two LEDs, each with its own 1K current limiting resistor are used to display the SUM and Carry outputs. Remember to connect Vcc and GND of both chips to the 5V and GND power supply rails respectively.

Full Adder

An obvious weakness of the half adder is that it can only add two bits so there is no way to add in the carry from the previous digit.

The full adder allows us to add A + B and also the carry bit from the previous (lower) digit. We can make an 8 bit adder by chaining 8 full adders together with the Carry Out from the lowest bit wired to the Carry In of the next higher bit and continuing this chaining from Carry Out to Carry In through all 8 adders. This method can be simply extended to make a 64 bit adder (or any other word size) by chaining 64 full adders.

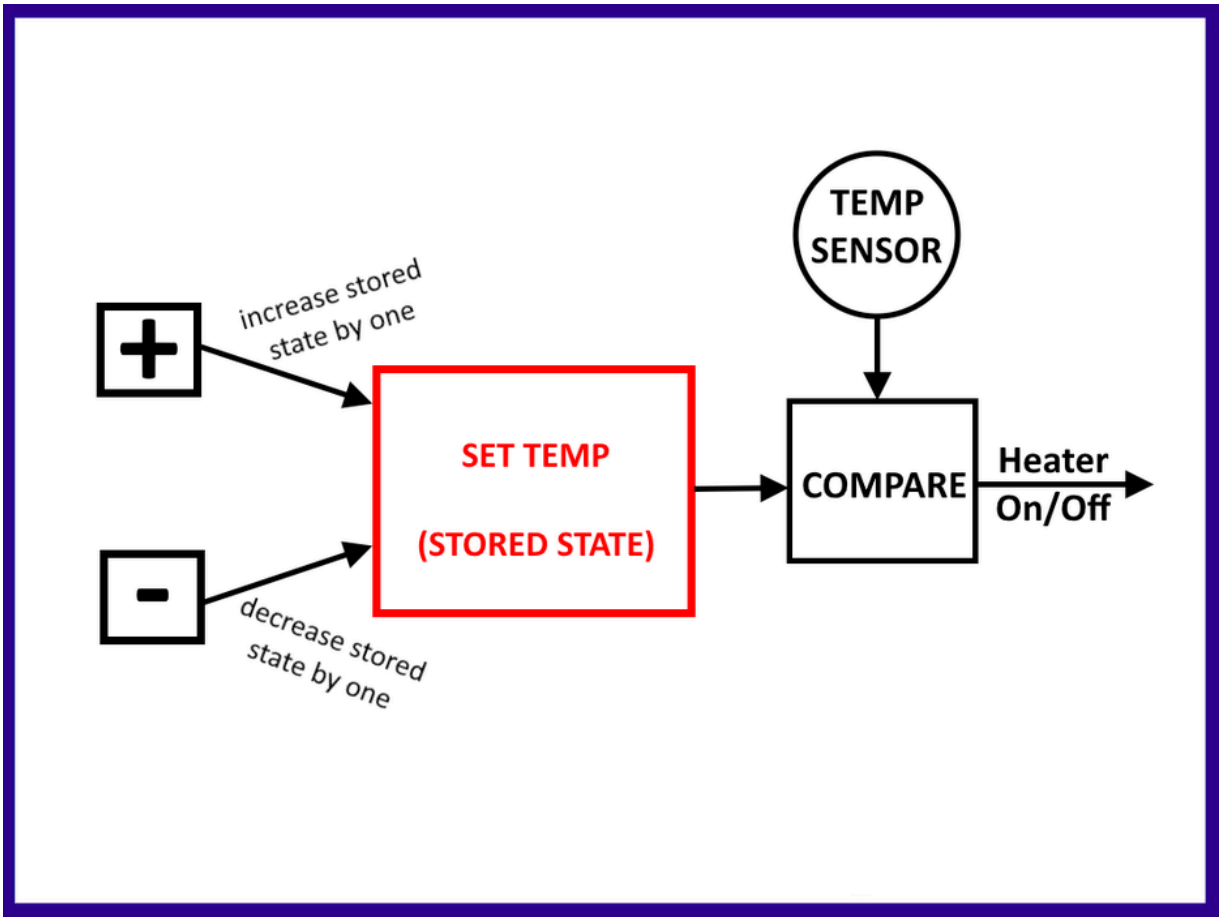
You can attempt to assemble the full adder on the breadboard using the two 7400 NAND chips to implement two XOR gates along with two AND gates from the 7408 chip and an OR gate from the 7432 chip.

Notice how this is getting a little messy. We will need a much higher level of integration to start doing very useful mathematics. It will use exactly the same gates as we've been working with, but just a whole lot more of them.

Arithmetic Logic Unit (ALU)

In a computer CPU, the ALU is a combinational digital logic circuit that does math. Generally an ALU is given two sets of input bits that represent two numerical values along with some control bits indicating which mathematical operation to perform on the two inputs. The ALU then generates a set of output bits representing the result of the computation. The ALU includes the circuitry necessary to perform the computation on the input bits. For example, the adders that we've built would be used when the control bits indicate that an addition is to be performed.

Step 39: Storing Information



Thus far, we have only seen logic circuits that compute an output based on the present inputs. When the inputs go away, the outputs change. This type of circuit is memoryless or stateless meaning that it has no memory and thus it cannot maintain state information. Making an analogy to switches, a momentary pushbutton does not "remember" that it was being pressed once it is released. Its state is lost because it has no memory. In contrast, a toggle switch (like a light switch) can "remember" when it is on or off without needing to be "held" in that state by the operator.

Logic circuits that can maintain state are called Sequential Logic. Sequential logic has memory and can generate outputs based on not only the present inputs, but past inputs, and even based on sequences of past inputs.

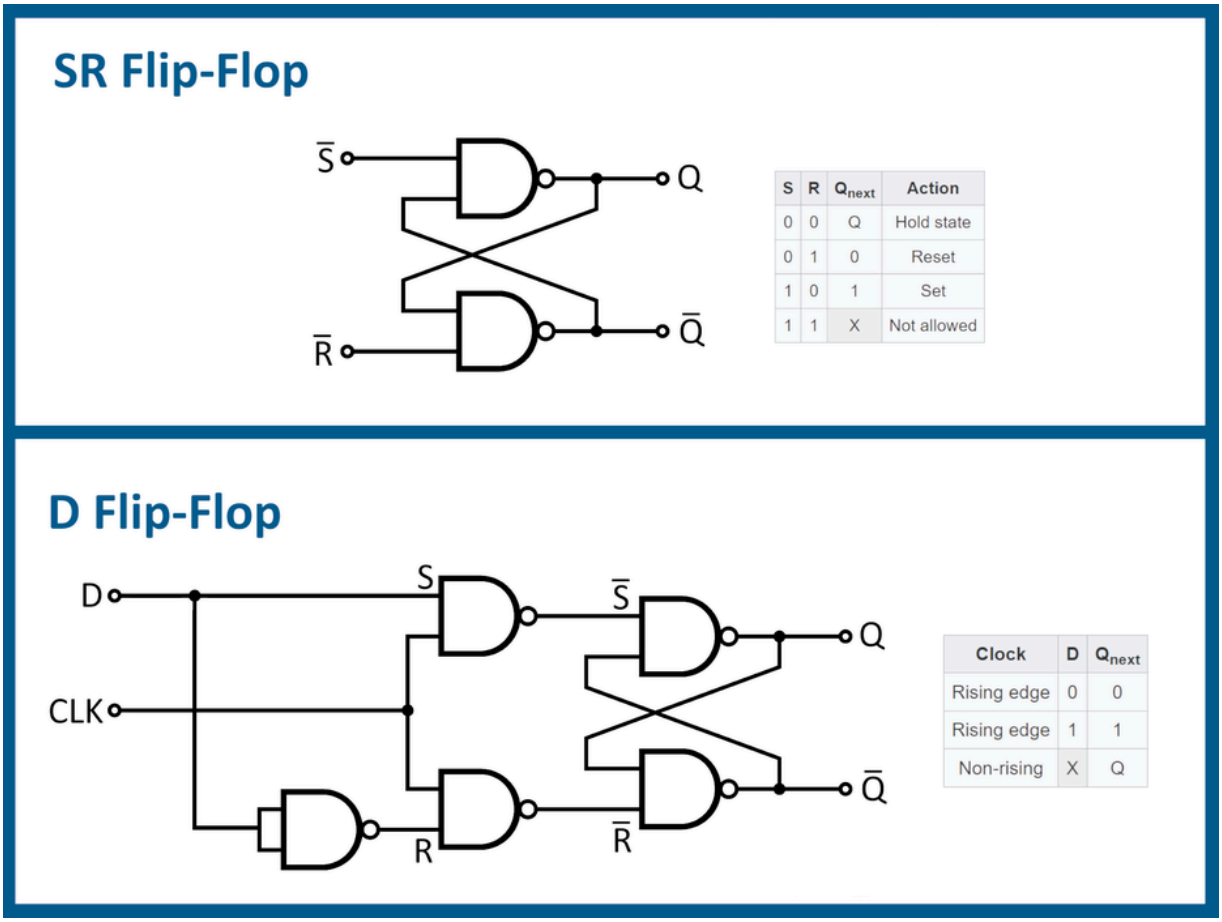
In the illustrated example, a simplified digital thermostat must maintain the state of its set temperature. The operator can adjust the set temperature up one degree by pressing the "+" button. If the set temperature was previously at 60 degrees, that maintained state information must be used by the circuit (along with the "+" input) to determine that the new set point must be 61 degrees. If the operator presses the "+"

button again, the set point will increase to 62 degrees. This demonstrates that the state information represents the results from a sequence of past inputs.

Since we've already looked at computer code on a microcontroller, this issue of state information may seen overly simplistic or even obvious. The state of the thermostat can simply be placed in a variable within a computer program. However, a digital logic circuit does not have variables. In fact, when we store information in a variable within a computer program, the computer is actually leveraging an electronic storage element - a sequential circuit.

Just as the combinational logic gates we learned about were the basis of the mathematic operations in the ALU of a computer, the storage elements we will learn about are the basis of the registers and memory of the computer.

Step 40: NAND Gate Flip-Flops



The basic electronic storage element is the flip-flop. Similar to how we combined NAND gates together to form an XOR gate, the same NAND gates (from the same 7400 quad NAND chip) can be combined to form flip-flops.

The simplest flip-flop is called the "set, reset flip-flop" or more typically the SR Flip-Flop. It only requires two NAND gates as shown here. The Q terminal is the output of the flip-flop and Q-Bar will always be the logical opposite of whatever Q is.

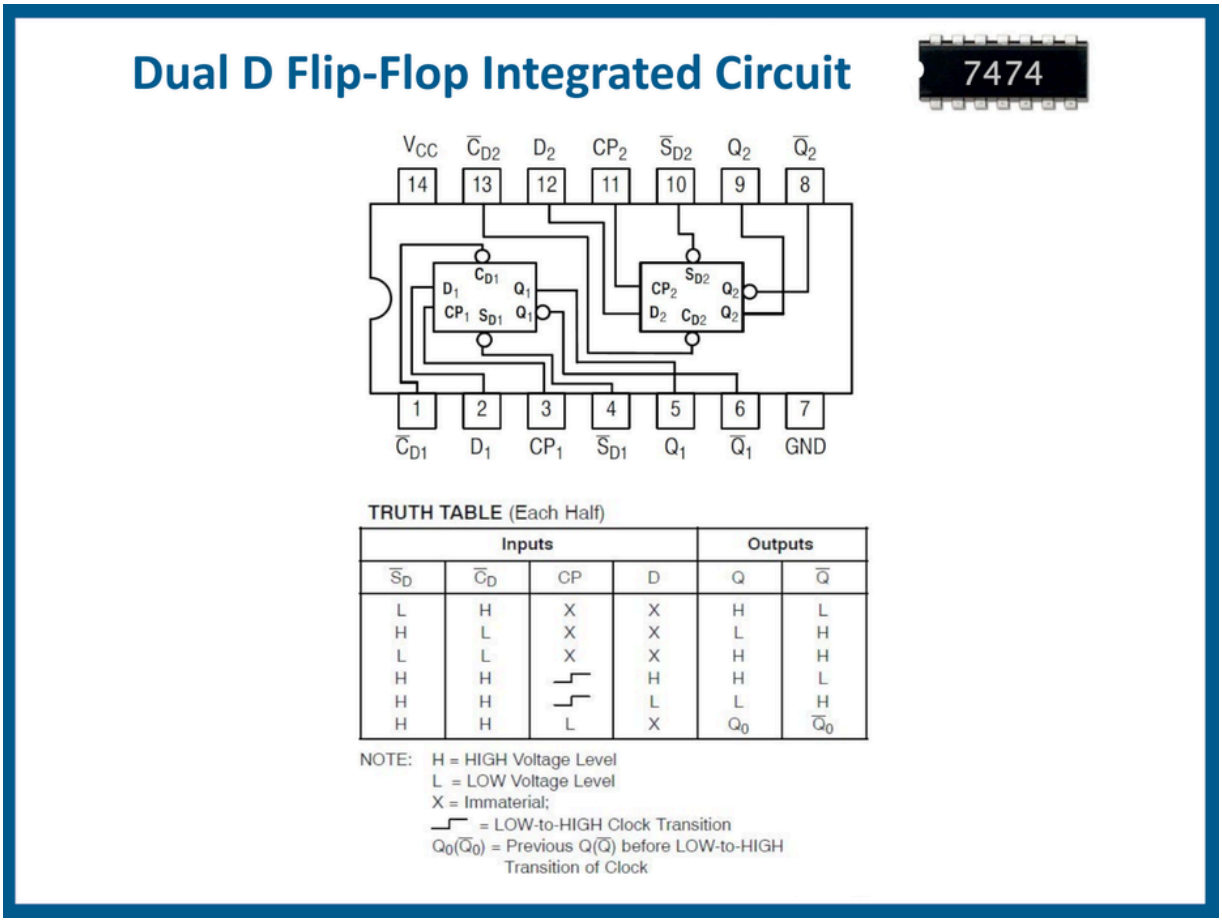
When the SR flip-flop is set (S = HIGH, meaning S-Bar = LOW), the output (Q) is set or HIGH. When the SR flip-flop is reset (R = HIGH, meaning R-Bar = LOW), the output (Q) is reset or LOW. When neither S or R are asserted (meaning they are both LOW or ZERO), the SR flip-flop is holding the output (Q) meaning that Q does not change. Since Q does not change in this hold condition, the SR flip-flop can be said to be storing a bit in memory or maintaining state.

A more versatile type of flip-flop is the D Flip-Flop, which can be implemented using five NAND gates (requiring two 7400 quad NAND chip). The D flip-flop locks its input (D) onto the output (Q) whenever the clock transitions from low to high (called "on the

rising edge"). When the clock signal is not rising, the output (Q) is maintained at the last value that was clocked-in without any care for what happens at the input (D). Again, the flip-flop can be said to be storing a bit in memory or maintaining state.

If you'd rather not wire up these examples on the breadboard, the digital logic simulator Logic.ly allows us to play with logic circuits right in a browser window. When the page first opens, you will see some samples to explore. One of them is a D flip-flop.

Step 41: D Flip-Flop Integrated Circuit



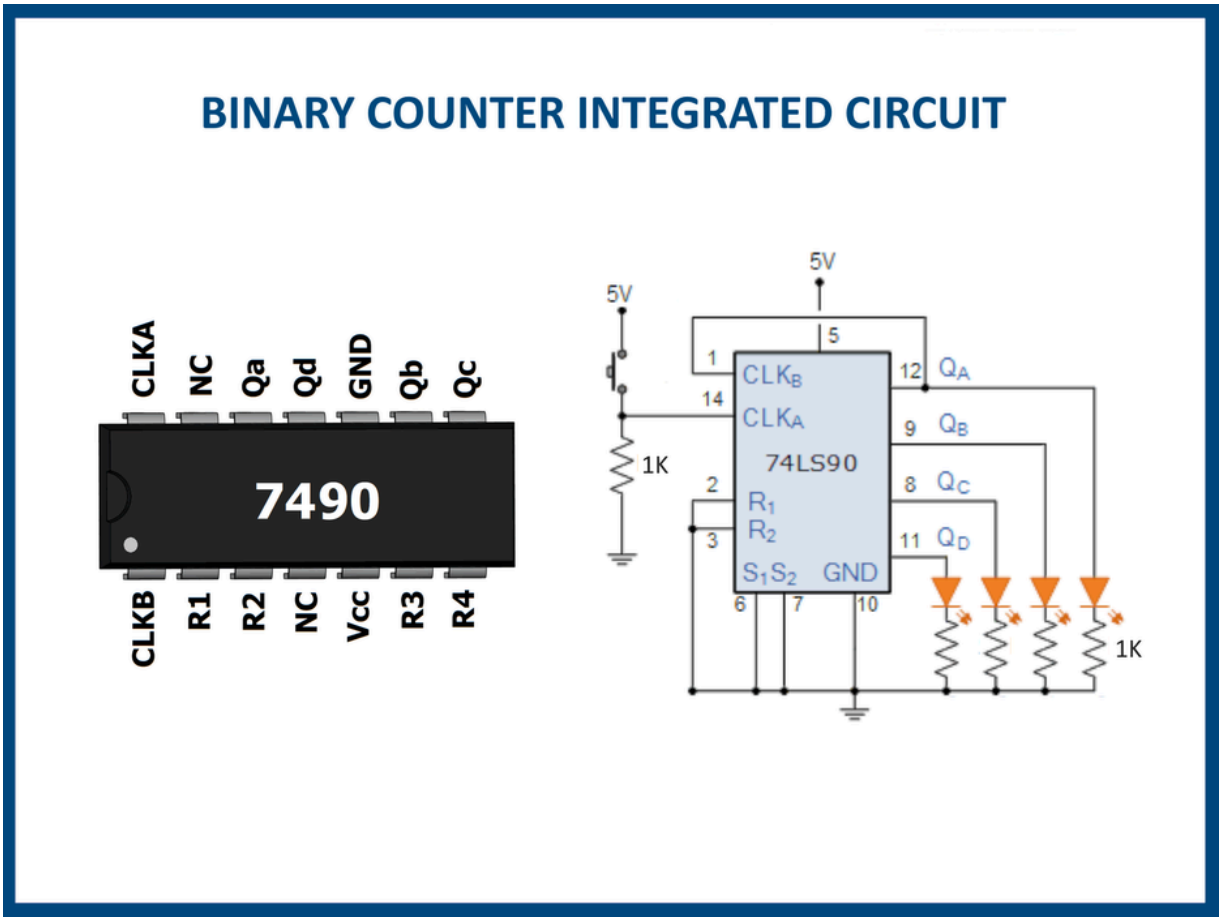
While it useful to understand how to implement flip-flips from individual gates, flip-flops are also available ready-made in integrated circuits, such as the 7474 Dual D-Type Flip-Flop chip (datasheet).

The two flip-flops in the 7474 chip have the typical D-Type signals (D, CLK, Q, and Q-Bar). Note that CLK is called CP in this diagram. In addition to these signals, each flip-flop also has a set (S) and clear (C) signal. These are basically set and reset, so the flip-flops are D-type with additional SR-type features. Note, in the truth table, that when S and C are both disabled (both HIGH) then the flip-flop acts just as our earlier D-type flip-flop. Otherwise, set and clear (S and C) perform their expected set and reset functionality overriding the data (D) and clock (CP) inputs entirely.

Why did we say that S and C are disabled them they are both HIGH? That is because they are both active-low signals. This is indicated by the bar on their names and the little circle at the input point in the diagram. An active-low signal is active when it is LOW and inactive when it is HIGH, which is opposite the normal expectation.

To try out a flip-flop from the 7474, use an on-off slide switch on the D input, a momentary input on the clock (CP) input, the usual pull-up/pull-down resistors, an LED on the output (Q), and also tie set (S) and reset (C) directly to 5V (HIGH) to force them to inactive. Observe that D can be set to either value but that value value doesn't lock into the state/output (Q) until there is a rising edge on the clock (CP) line.

Step 42: Binary Counter

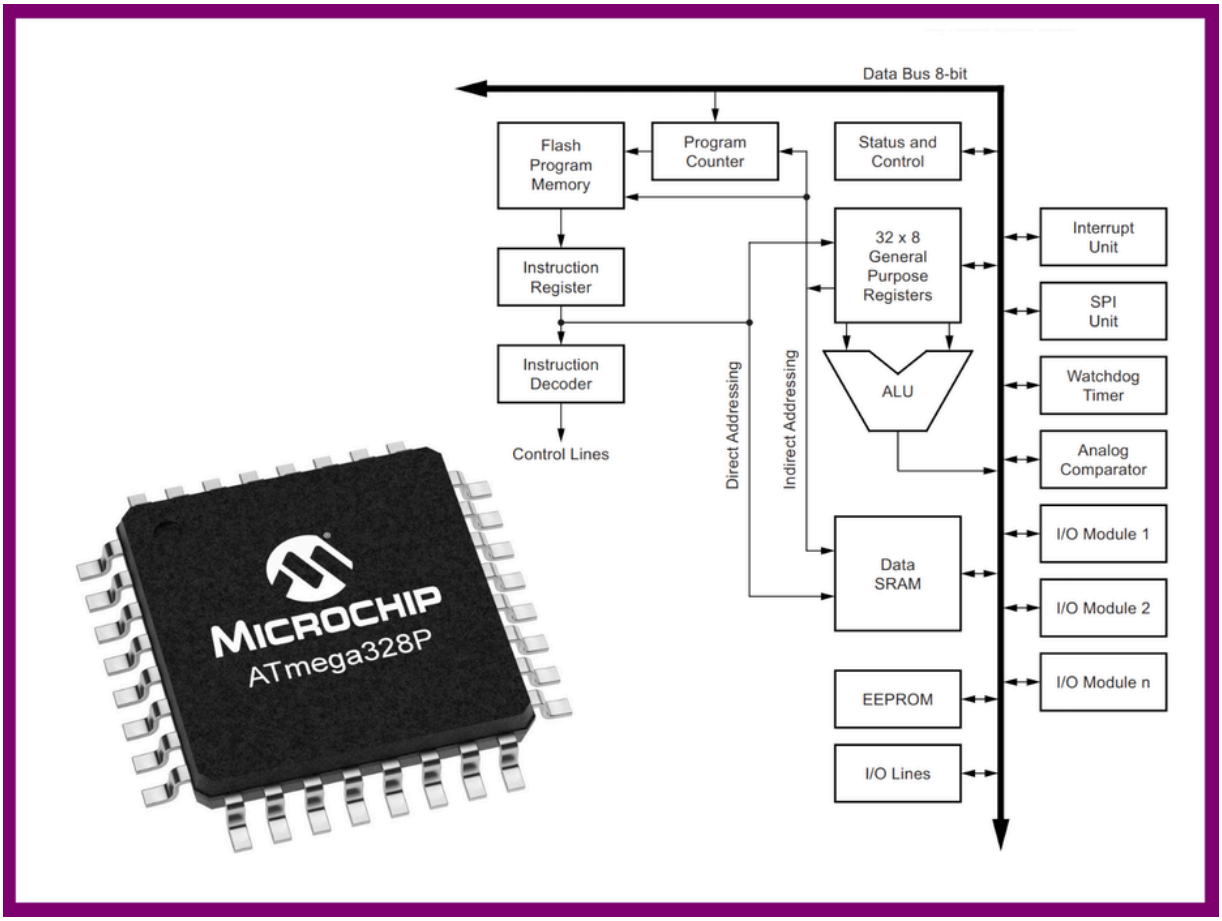


As discussed, the 7474 chip is a level of integration higher than a bunch of NAND gates. There are plenty of even more integrated chips that have multiple gates and multiple flip-flops. One such example is the 7490 Four-Bit Counter Chip (datasheet).

In the 7490, four bits of state Qa, Qb, Qc, and Qd are maintained (using flip-flops) and the binary value represented by this nibble (half of a byte) is incremented (counted up by one) every time the clock is pulsed. Assemble the 7490 circuit on the breadboard and give it a spin.

Approximately how many NAND gates would it takes to implement this four-bit counter? How many transistors would it take to implement those NAND gates? And 16 times that many to implement a 64 bit counter. It is quite easy to appreciate the power of successive integration.

Step 43: Computer Architecture



Computer Architecture is an area of Computer Engineering concerned with the structures of computer processors. These structures include functional blocks, memories, buses, and control signals. The microarchitecture of a processor includes blocks and interconnections that implement the control path describing what the processor does and also the data path describing how data moves through the processor. The arithmetic logic unit (ALU) we've already discussed is an important example block of the architecture.

The instruction set architecture (ISA) specifies the instructions that the processor can execute and how they relate to the system's memory, data registers, and buses. A microprocessor generally includes a very small number of memory units called the registers. The instructions performed by the processor operate directly on these registers and then output results directly to the registers. The contents of the registers can be loaded in from the main memory or stored out to the main memory. The main memory is generally much, much larger than the register space (also called the register file).

The ATmega328P microcontroller chip that we are already familiar with has a RISC architecture 8-bit AVR processing core. The processor support 131 instructions that are mostly capable of executing in a single clock cycle. The processor includes a register file of 32 8-bit registers. According to the ATmega328P datasheet:

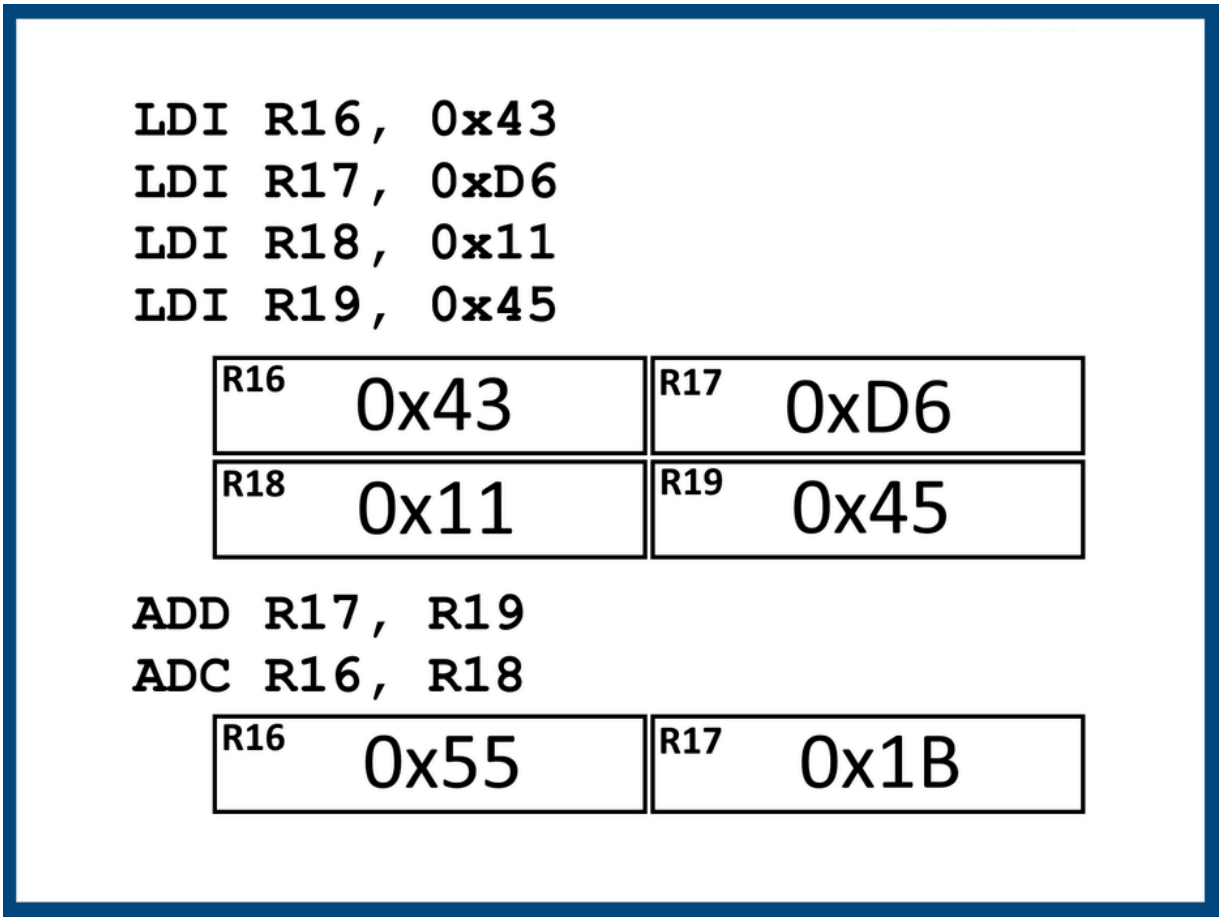
The AVR uses a Harvard architecture – with separate memories and buses for program and data. Instructions in the program memory are executed with a single level pipelining. While one instruction is being executed, the next instruction is pre-fetched from the program memory. This concept enables instructions to be executed in every clock cycle. The program memory is in-system reprogrammable flash memory.

The fast-access register file contains 32 8-bit general purpose working registers with a single clock cycle access time. This allows single-cycle arithmetic logic unit (ALU) operation. In a typical ALU operation, two operands are output from the register file, the operation is executed, and the result is stored back in the register file – in one clock cycle.

The ALU supports arithmetic and logic operations between registers or between a constant and a register. Single register operations can also be executed in the ALU. After an arithmetic operation, the status register is updated to reflect information about the result of the operation.

Program flow is provided by conditional and unconditional jump and call instructions, able to directly address the whole address space. Most AVR instructions have a single 16-bit word format. Every program memory address contains a 16-bit or 32-bit instruction. Program flash memory space is divided in two sections, the boot program section and the application program section. Both sections have dedicated lock bits for write and read/write protection.

Step 44: Assembly Language and Machine Code



From the 131 instructions that the AVR processor can execute, lets use just three of them to write a little program. We will use LDI (load immediate), ADC (add with carry), and ADD (add without carry).

Of the 32 8-bit registers inside the processor, we will be using four of them. They are called R16, R17, R18, and R19.

The program adds two 16-bit values together 0x43D6 and 0x1145. Since the registers are only 8 bits wide, each of those values will require two registers. The first four operations load the values that we want to add into the registers as shown by the four blocks in the center.

Next, the two lower bytes are added together using ADD and lastly the two higher bytes are added using ADC. When the higher bytes are added using "add with carry" any carry generated by adding the lower bytes is also added into the higher bytes.

While very close to the machine language used by the processor itself, assembly language is still reasonably readable by a human. A program called an assembler can

be used to convert assembly code into machine code, which in this case would be:

```
E4 03 ED 16 E1 21 E4 35 0F 13 1F 02
```

The machine code 0xE403 represents "load 0x43 into register 16", 0xED16 represents "load 0xD6 into register 17", and 0x1F02 represents "add register 16 and register 18 along with any carry from the most recent addition and put the results in register 16".

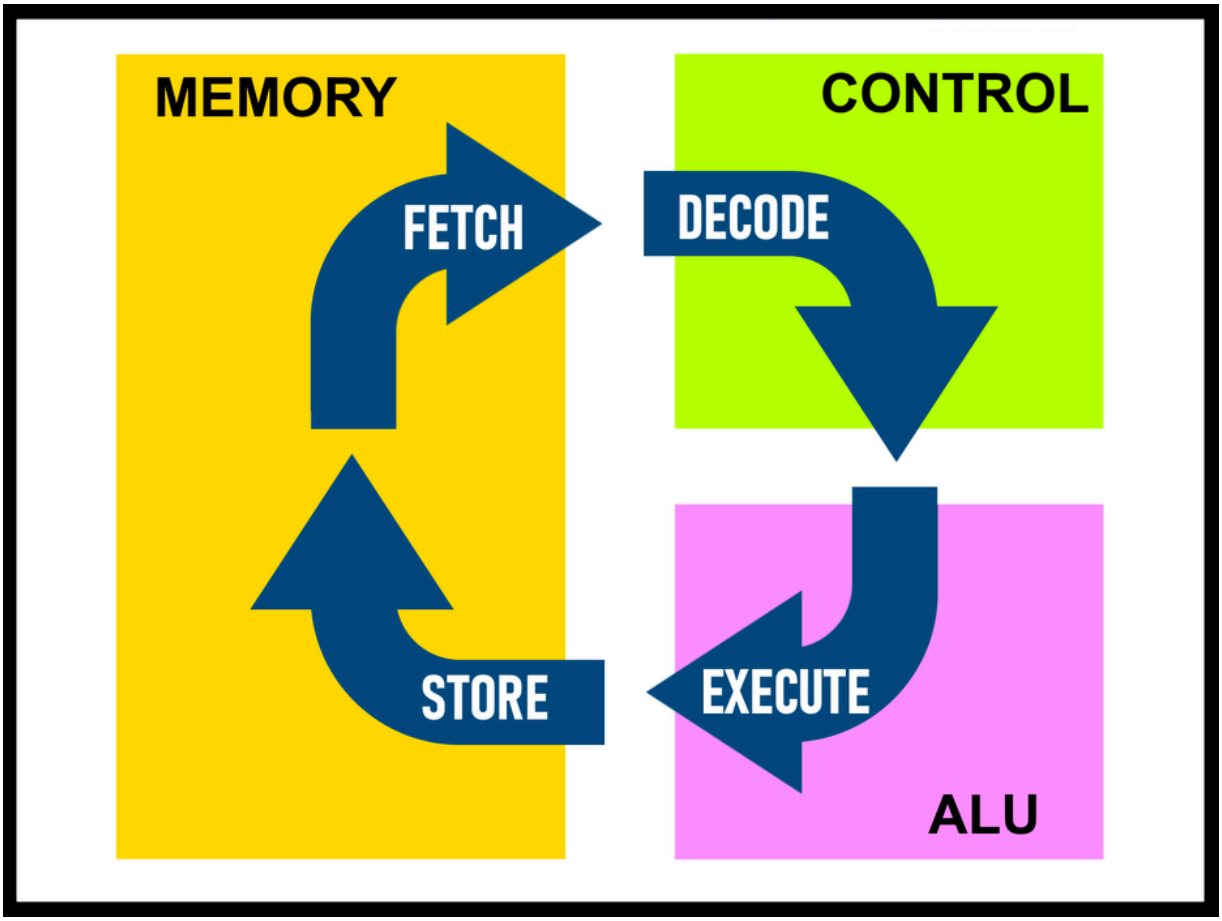
If you are very brave, you can precisely decipher the machine code using the AVR Instruction Set Manual. For example, the LDI instruction is broken down on page 115.

This is quite messy, which is why assemblers were developed to allow humans to efficiently write programs using assembly language instead of machine code. Of course, higher level languages like C/C++, or Python allow us to simply write something like:

```
register int sum = 0x43D6 + 0x1145;
```

We've come a long way!

Step 45: Instruction Cycle



The instruction cycle of a processor includes three stages: fetch, decode, and execute. We might also include store as a fourth stage or simply an implied stage.

It is useful to consider the blocks of the processor architecture while considering these stages.

Fetch: The program counter is a register that always contains the memory address of the current instruction being executed. That address gets pushed to the address bus to read the instruction from the main system memory into the instruction register. The program counter is incremented for use in the next cycle.

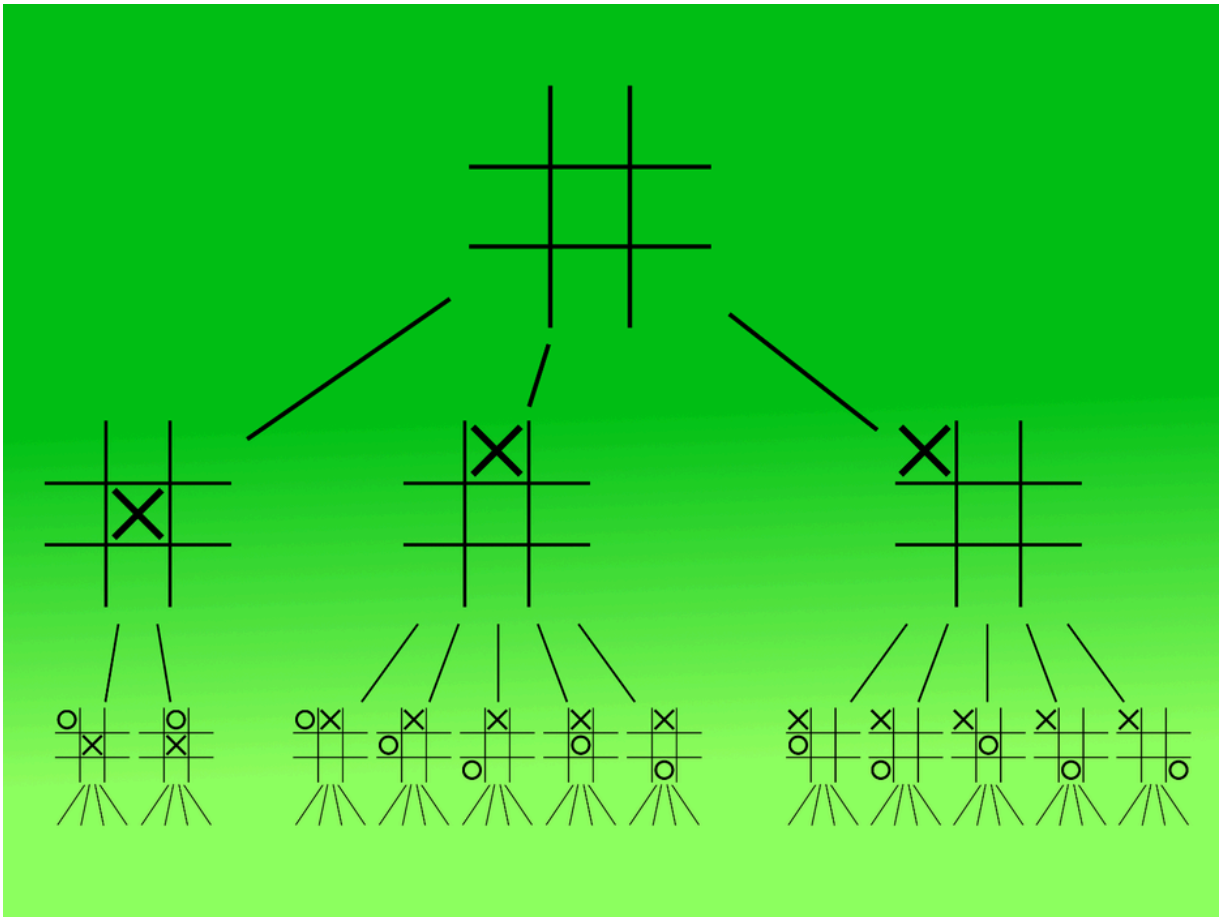
Decode: The instruction decoder parses the value in the instruction register to determine what the processor is being instructed to do. This is represented as control signals fed to other blocks within the processor.

Execute: The decoded instruction is carried out - generally by the ALU or as a memory operation.

(Store): A value may be stored from a register out to the main system memory.

The instruction cycle repeats forever.

Step 46: Algorithms and Heuristics

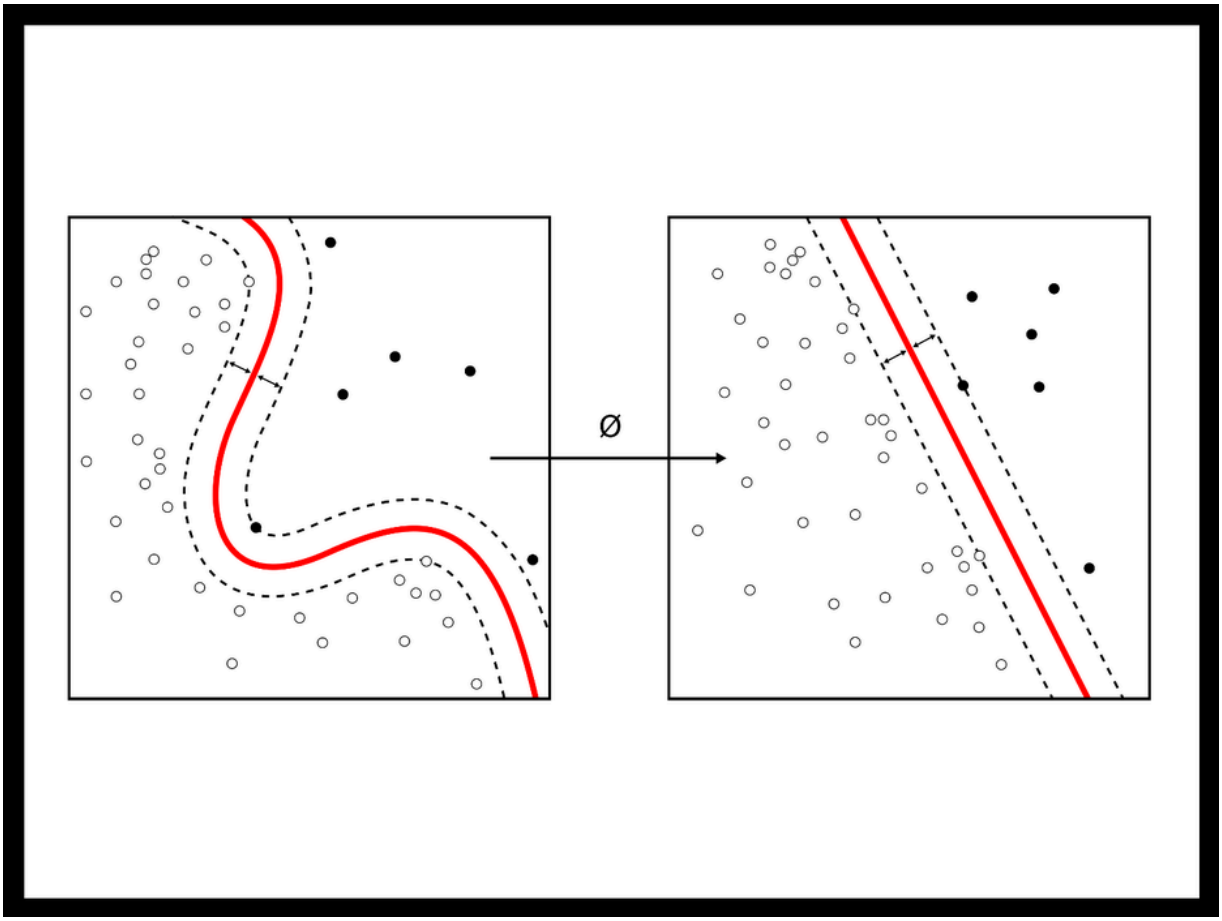


Algorithms, like programs we've encountered thus far, are specific sequences of steps or instructions to solve particular problems. The results of an algorithm are usually predictable and repeatable. Common algorithms in computer science include searching, sorting, graph traversal, game trees, and numerical algorithms such as root finding, integration, or transformations.

A heuristic approach, unlike an algorithmic solution, is generally an approximation or "best guess" to a problem that is often incompletely defined or too complex for timely exploration of a closed-form solutions. The results are usually neither predictable nor repeatable.

Using two examples from game-theory, an algorithm for tic-tac-toe would simply generate and explore all possible branches of the entire game tree. In contrast, a heuristic approach to playing chess might ignore (prune away) large portions of the game tree that do not roughly match certain characteristics of the present game. In the heuristic case, the entire tree is not searched, but the most likely portions are searched as a "best guess" approach.

Step 47: Machine Learning



Machine Learning (ML) techniques allow machines to discover solutions to problems. Machine Learning is especially useful when a problem is hard to define or changes in real-time rendering the development of explicit human-programmed algorithms too static or costly.

ML approaches can be categorized into three types:

Supervised Learning: A training set provides the system with correct examples of inputs and outputs. The system develops a trained model capable to map the inputs to the outputs.

Unsupervised Learning: The system is given uncategorized input data and seeks to discover structure in the data. The discovered patterns may be useful on their own or may be used to establish a trained model.

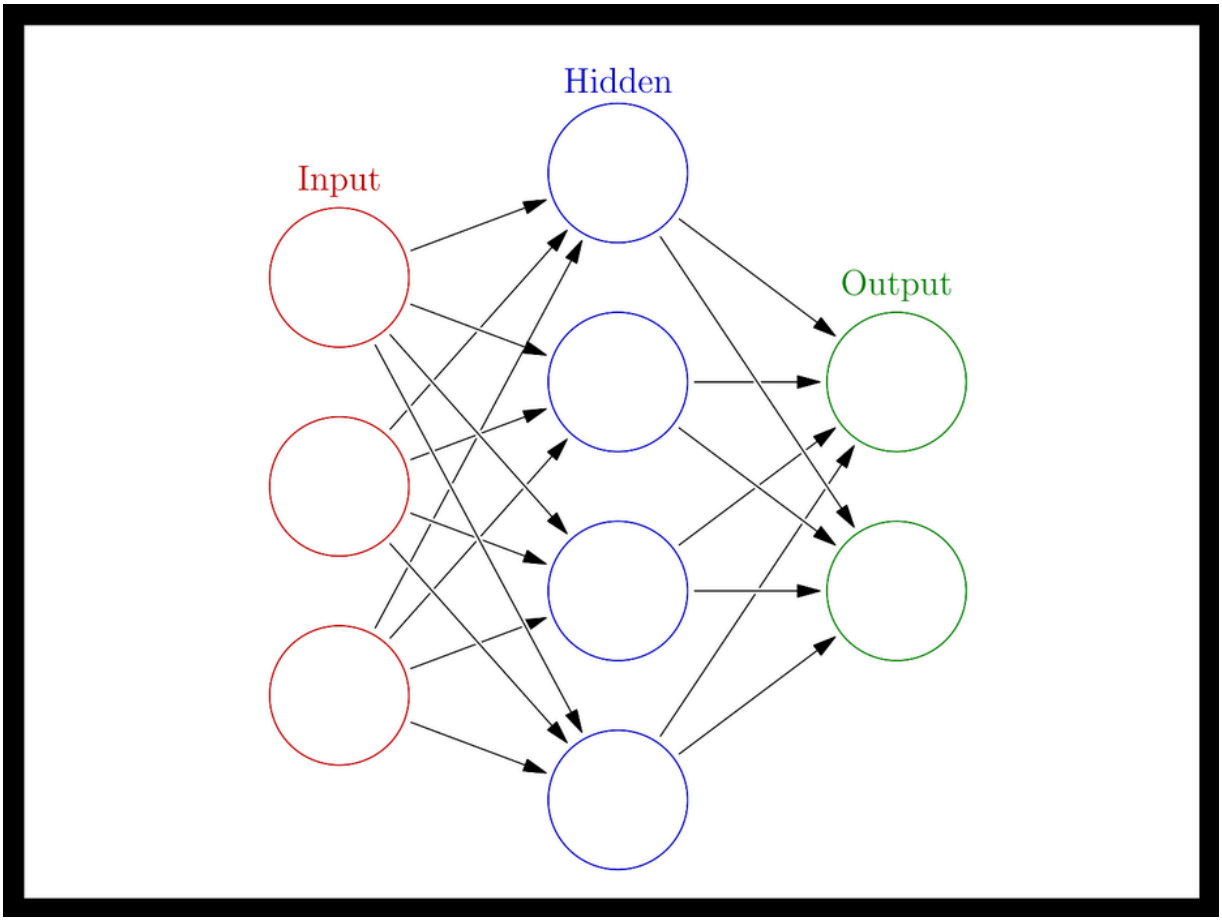
Reinforcement Learning: The system performs a task such as playing a game or picking stocks. The system's performance is reinforced (rewarded) for being successful and the system seeks to maximize obtaining rewards.

While ML is often associated with large data sets and hefty compute resources, however TinyML can run on tiny microcontrollers (such as our ATmega328P) with low power consumption. TinyML allows ML capabilities to be integrated into virtually any device.

This example of Arduino Machine Learning demonstrates hot to train a TinyML model using an off-line python program that implements a classifier based on decision tree learning.

The example then executes the trained model within an impressively lightweight Arduino sketch.

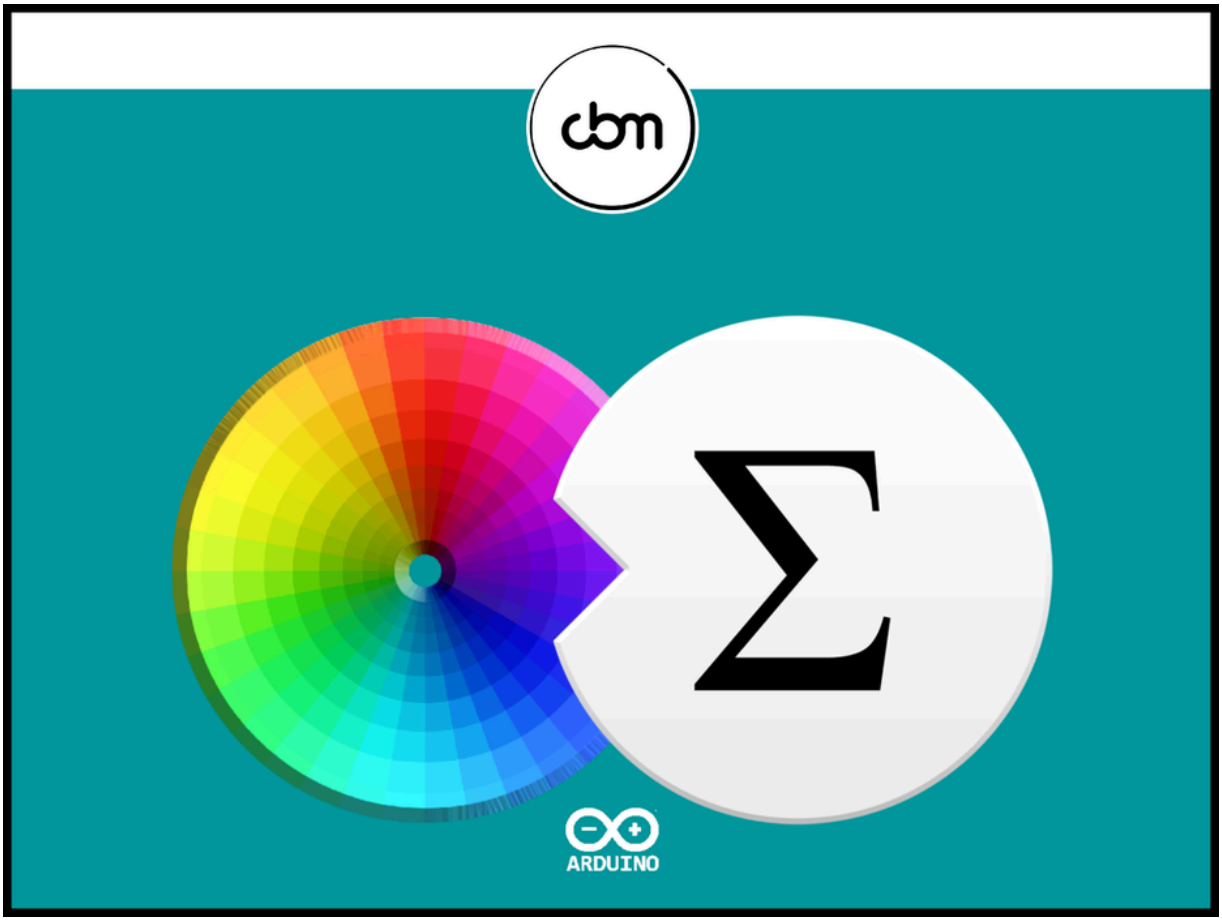
Step 48: Artificial Neural Networks



Neural Networks, or more correctly artificial neural networks (ANNs) mimic the biological neuron structures of brains to implement specific types of machine learning. In ANNs, sets of artificial neurons are connected by weighted signals analogous to synaptic connections between biological neurons.

The output of each neuron is computed by a linear sum of its inputs. The connections between neurons have a weight that is established by the training of the network. The weight increases or decreases the strength of the connection. The neurons are generally aggregated into layers including a first layer (the input layer) and a final layer (the output layer). One or more layers between in the input and output layers may be referred to as hidden layers.

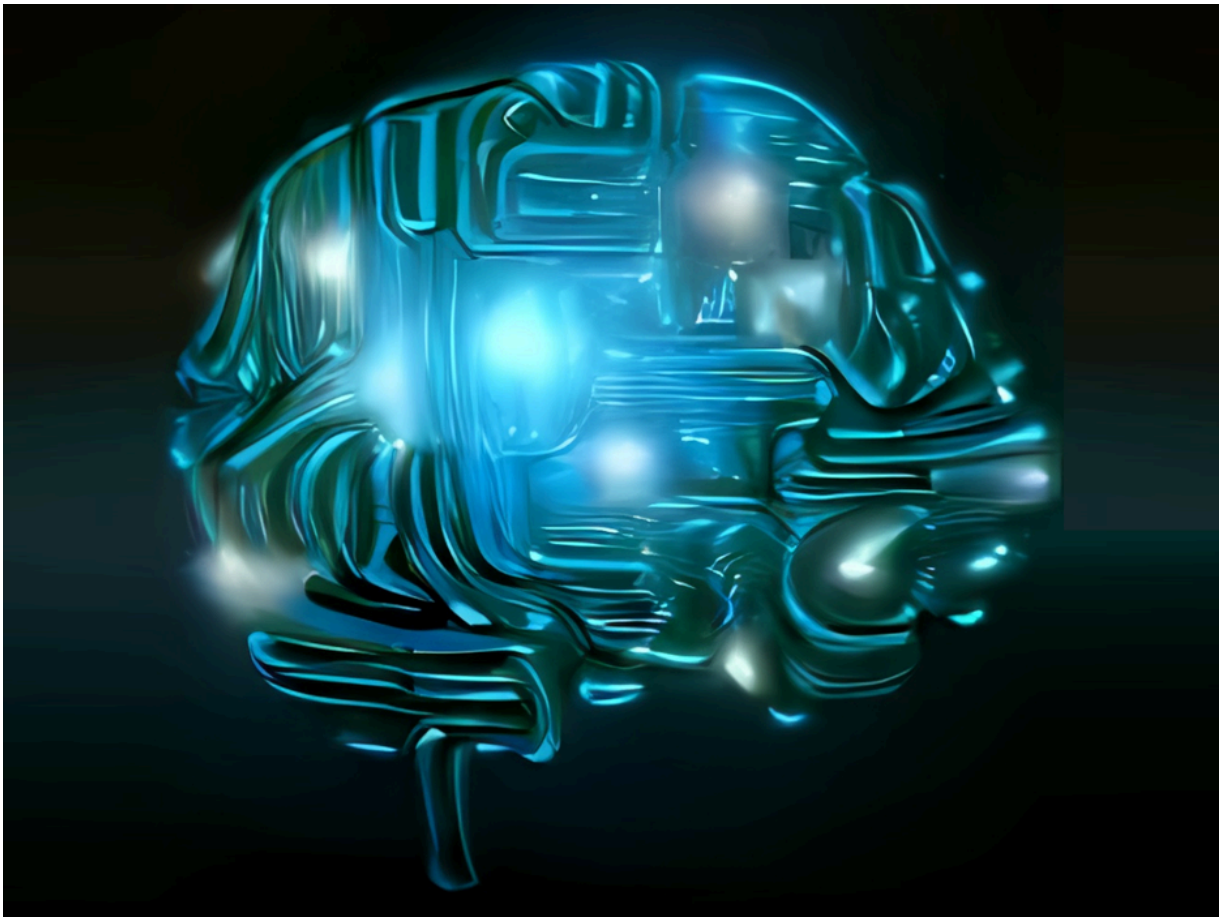
Step 49: Embedded Neural Networks



Efficient implementations of ANNs can even be embedded into small microcontrollers. For example, the Arduino Neruona Library from Caio Benatti Moretti allows Arduino boards to load Artificial Neural Network (ANN) structures and perform tasks such as pattern recognition (classification), non-linear regression, function approximation, and time-series prediction.

The ColorSesnor demo that comes with the library trains a neural network using the photo resistor to collect reflected light from the RGB LEDs. The trained network is capable of classifying colors as demonstrated by Moretti in this blog entry.

Step 50: Artificial Intelligence



Artificial Intelligence (AI) is a vast field comprising many different problem categories and at even more solution techniques.

A.I. was founded as an academic discipline in 1956 and has since moved through multiple cycles of optimism followed by disappointment and loss of funding. As of 2012, when deep learning surpassed previous AI techniques, there has been a vast increase in funding and interest. (Wikipedia)

The A.I. technologies we have surveyed in this tutorial, including statistical learning, game trees, and neural networks, are surely just the tip of the iceberg.

"The first ultra-intelligent machine is the last invention that man need ever make."

- I.J. Good, A.I. Researcher, 1965

Step 51: Hack the Planet

We hope you are enjoying the **HackerBox Basics Workshop**. To continue your adventure into electronics, computer technology, and hacker culture, visit HackerBoxes.com where you can find the HackerBox Soldering Workshop, and HackerBox Core Workshop. Together, these three workshops build towards increasingly advanced electronics projects, such as those found in the monthly HackerBox subscription boxes.