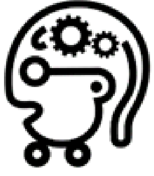


Experiment 1:

Simple Navigation, Servos, Sound and LEDs

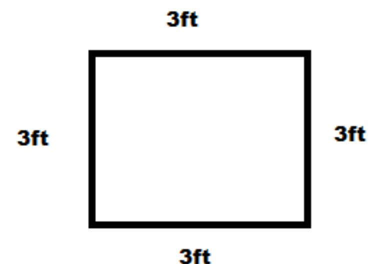
Purpose: Complete assembling the robot by testing its onboard computer, motors and power source then have the robot drive in whatever direction you command it to for a particular amount of time.

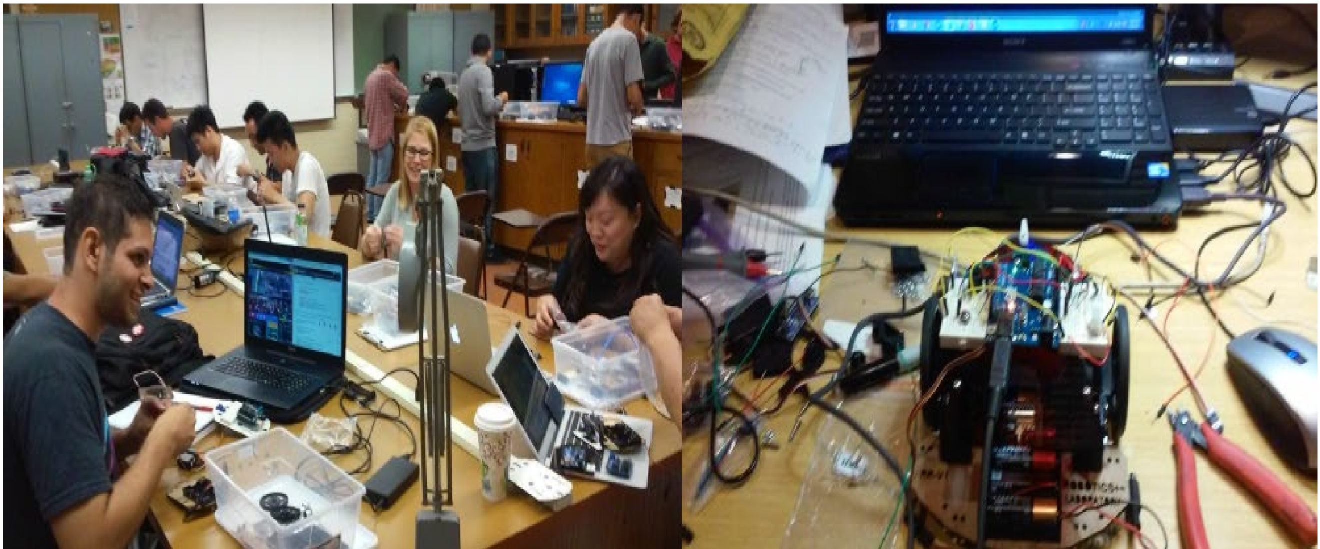
Note: If you are taking the class in person here are the requirements of what you need to show at the end of this experiment. Press a switch to start a program that makes beeping sound for 2 seconds before starting then have the robot move in a square shape (3 ft-squared path marked by blue tape on the floor) while making a beeping sound at every turning corner then stop where the robot started at. The robot might never go perfectly straight because we are not using sensors on the wheels to verify that is doing the correct motion so do not worry if the robot veers to the side a little.

For the lab report please include code and photos of your robot (if you can) for all labs.

What you will learn:

- test your assembled robot
- know common robot parts such as servos, microprocessors, sensors and materials
- learn how to install/configure and program the Arduino board
- learn about servo motor calibration and how to control speed and direction via pulse width modulation
- learn about piezo (speaker) and tones for sound
- learn about LEDs, polarity and timing
- learn about Arduino shields





In this experiment we will use the robot kit from roboticscity.com This kit uses a microcontroller board knows as Arduino UNO board. The kit also comes with a myriad of sensors, actuators and other components. You will be learning both hardware and software components used in mechatronic devices. You will gain an understanding of how to interface and control sensors plus actuators using commands sent by a microcontroller to create autonomous robots. You will use software to program logic using a C++ like language that make machines think and act. There are many How To guides on the Arduino online and on the lab computers – check them out!

Arduino Background

Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It was originally intended for artists, designers, hobbyists and anyone interested in creating interactive objects or environments so it becomes the perfect platform to learn the essentials of robotics.

Arduino can sense the environment by receiving input from a variety of sensors and can affect its surroundings by controlling lights, motors, and other actuators. The microcontroller on the board is programmed using the Arduino programming language (a combination of C/C++ language, but much simpler to understand) and the Arduino development environment (based on the Processing language https://processing.org/). Arduino projects can be stand-alone or they can communicate with software running on a computer

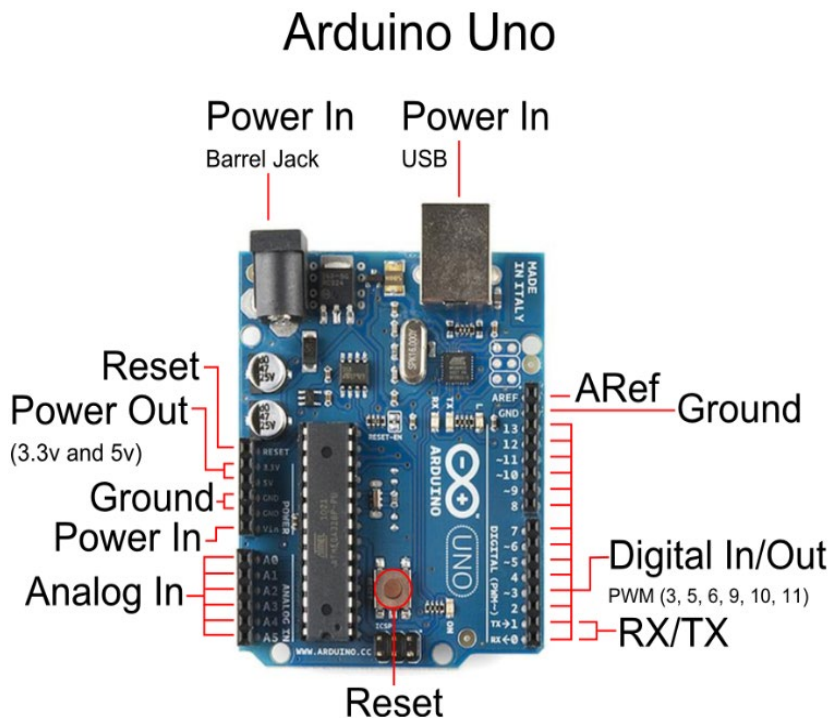


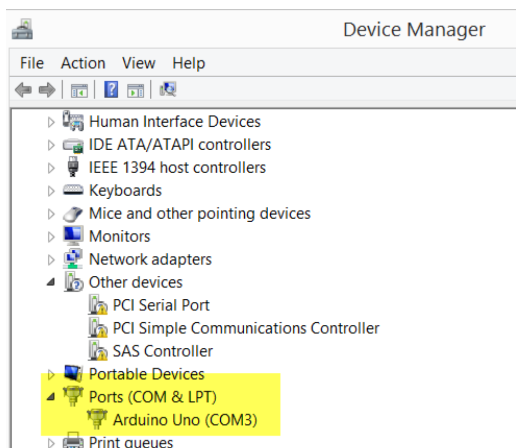
Fig.1 – this shows the Arduino development board and all of its inputs and outputs plus USB (printer style) programming cable

Software Installation:

Installing drivers for the Arduino Uno with **Windows 7/8/10**

- Download the free Arduino software from http://www.roboticscity.com/ or from https://www.arduino.cc/ and OPEN/RUN the installation .EXE file your downloaded from roboticscity.com or the Arduino website and wait for the installation to complete (or you will get an error). You may also download versions of Arduino for Macs and LINUX. See https://www.arduino.cc/ for more info.
- Plug in your USB cable to the robot's Arduino board and wait for Windows to begin it's driver installation process. After a few moments the driver will finish installing (you can see the small green USB icon changing and you can click on it to see the progress.

- Note: If the driver the process fails. (in Windows 7/8/10/Mac drivers are installed automatically, but sometimes it fails)
- Click on the Start Menu, and open up the Control Panel.
- While in the Control Panel, navigate to System and Security. Next, click on System. Once the System window is up, open the Device Manager.
- Look under Ports (COM & LPT). You should see an open port named "Arduino UNO (COMxx)". If there is no COM & LPT section, look under "Other Devices" for "Unknown Device".



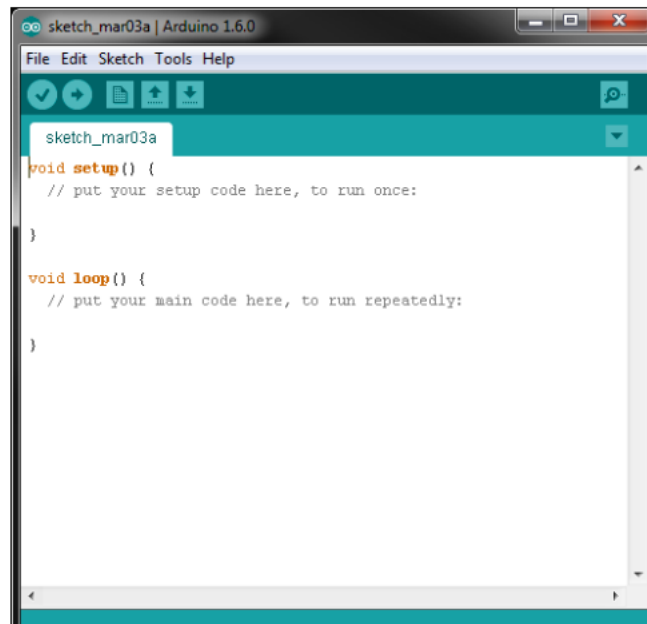
- Right click on the "Arduino UNO (COMxx)" port and choose the "Update Driver Software" option.
- Next, choose the "Browse my computer for Driver software" option.
- Finally, navigate to and select the driver file named "arduino.inf", located in the "Drivers" folder of the Arduino Software download. This folder is typically c:\Program Files\Arduino\drivers or C:\Program Files (x86)\Arduino\drivers for 64bit Operating System

Windows will finish up the driver installation from there and now we are ready to learn about the Arduino (IDE) Integrated Development Environment.

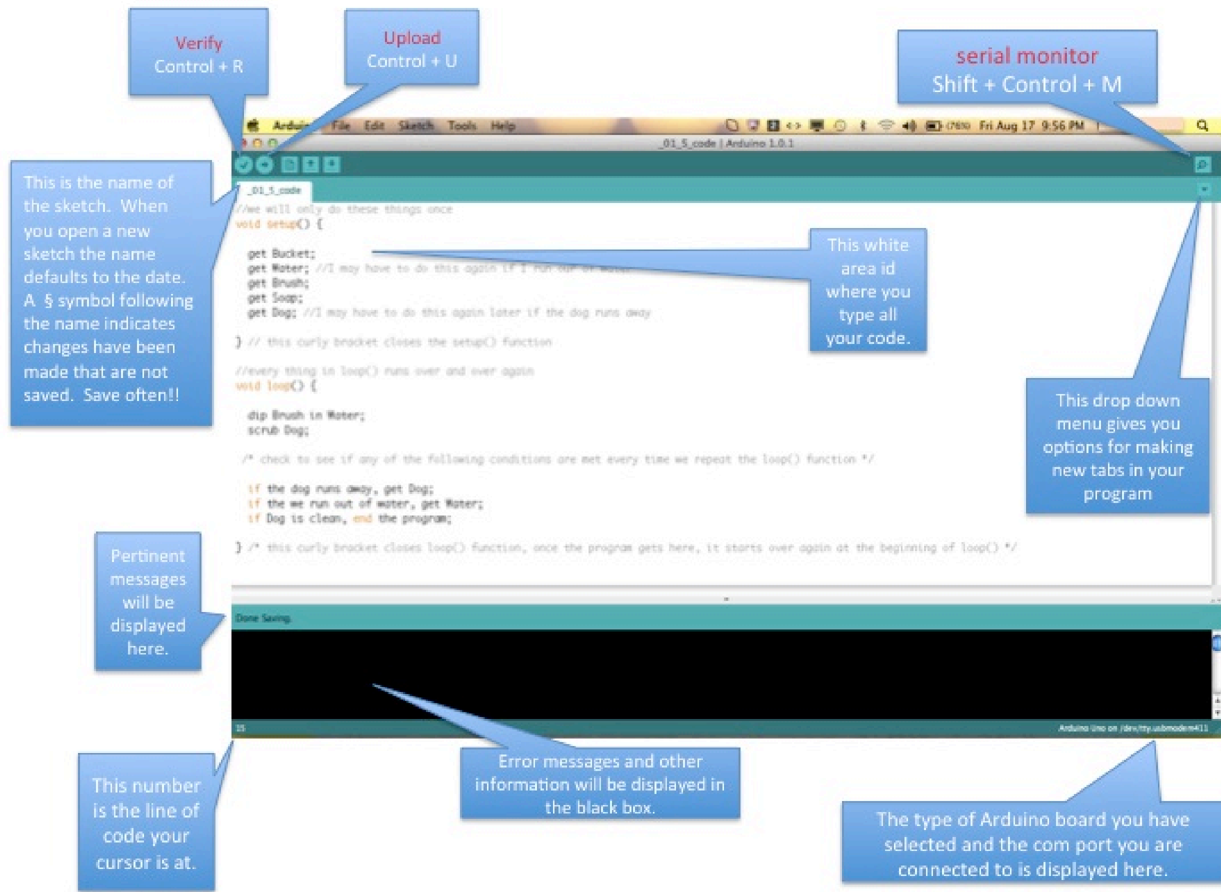
Launch the Arduino application and Testing the Arduino board using the built-in Blink LED sample

Once the program has been installed, double-click the Arduino application icon.

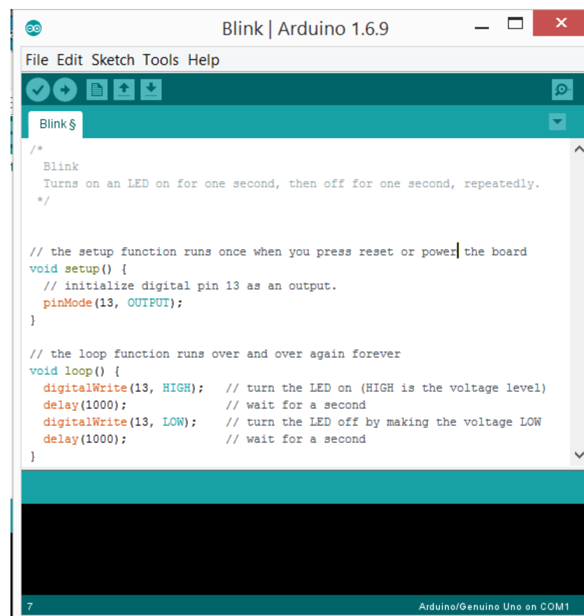
The Arduino IDE (integrated development environment)



Arduino IDE Buttons and Status Bars

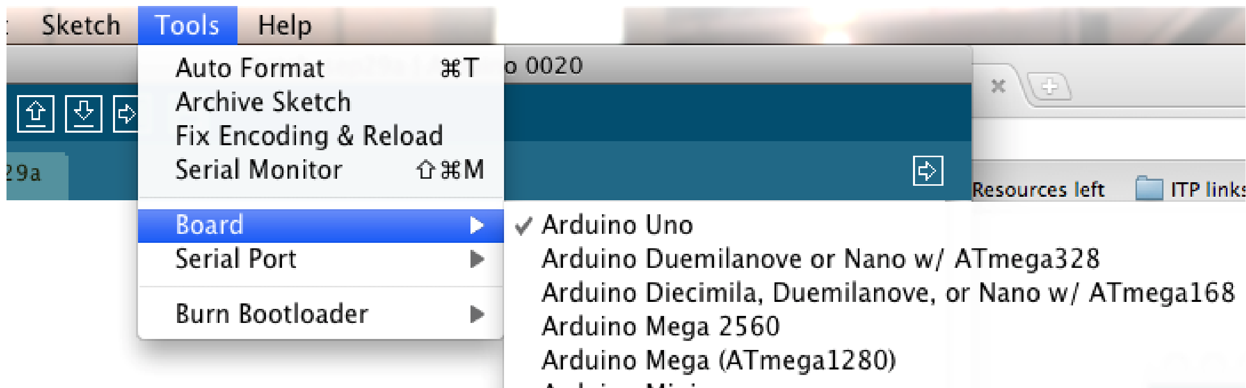


Open the LED Blink example program that came with the Arduino software by going to the menu and select File then Examples>0.1 Basics>Blink

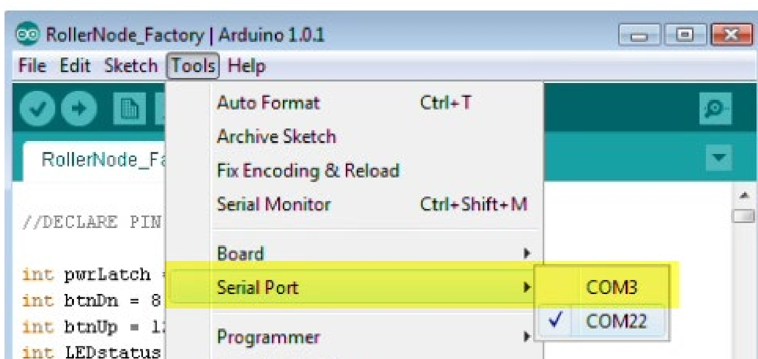


Note: Programs in the Arduino environment are referred to as sketches so we will call programs sketches from now on.

On the menu select your board under Tools\Board. This robot uses the Arduino UNO board



Select your serial port



Select the serial device of the Arduino board from the Tools | Serial Port menu. Select one of the ports listed. Typically COM4 or higher. To find out, you can disconnect your Arduino board and re-open the menu; the entry that disappears should be the Arduino board. Reconnect the board and select that serial port.

Upload the program

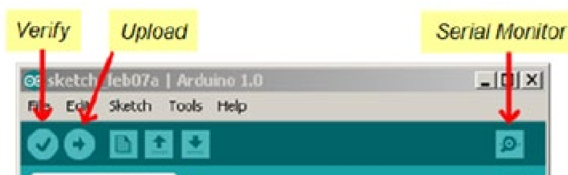
Now, simply click the "Upload" button in the environment. Wait a few seconds - you should see the RX and TX LEDs on the board flashing. If the upload is successful, the message "Done uploading." will appear in the status bar. You should see the built-in LED blinking on the Arduino board. This means your board is working! This program will be explained in detailed later on.

Troubleshooting: If you still can't see the serial port appear on the list simple unplug the Arduino board and plug it back in.

Learning more about the Arduino IDE - Write a Simple "Hello World" Sketch

Arduino calls their programs ***Sketches*** so you will see this word used throughout these manuals. The word sketch comes from the fact that the Arduino environment was meant to be used by artists so they could "sketch" their ideas or programs. **Note** the Serial Monitor icon on the right on the previous figure. When you click that a new window appears and it shows you the output results of your program on the computer screen. You will see how useful the Serial Monitor comes when you need to troubleshoot code, read sensor values or send/receive commands.

Let's try using the Serial Monitor



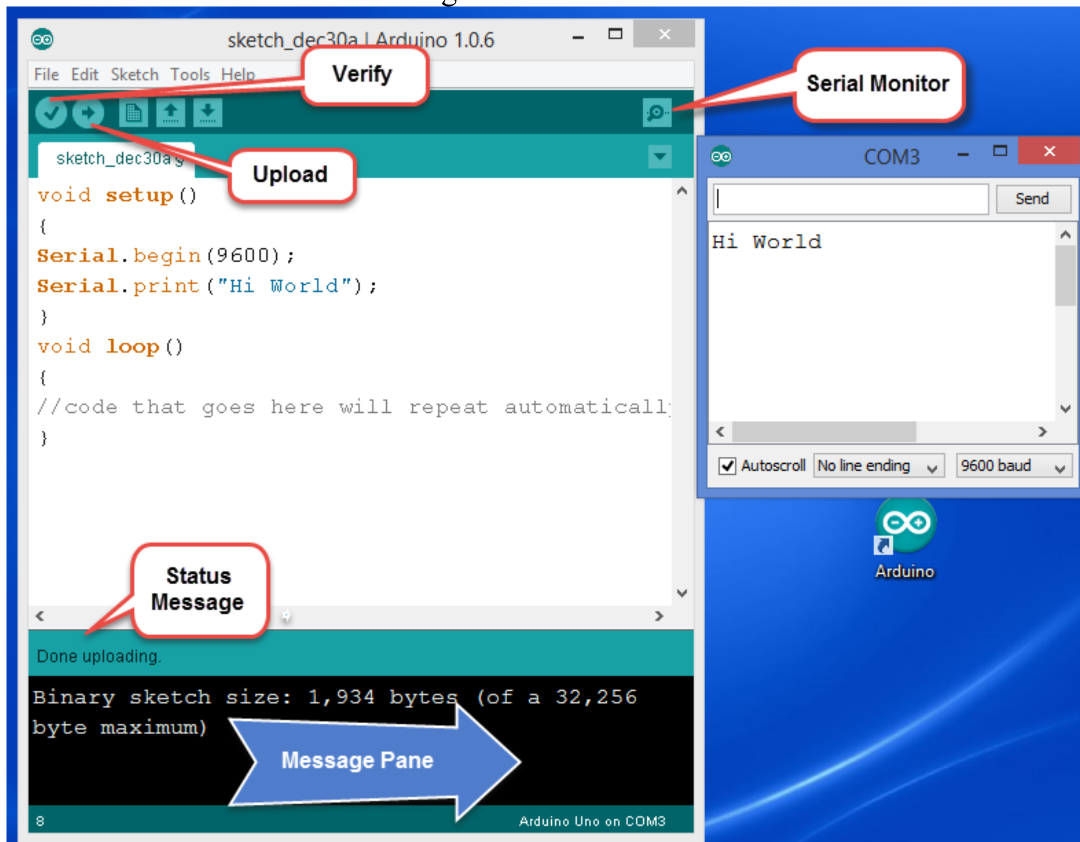
Open your Arduino software and carefully type in (or copy and paste) the code below. It must be typed exactly as you see it. Note: Capital letters must be capital as it is case sensitive for built-in statements like Serial.begin for example has a capital S. Everything except comments on the code need to be typed exactly as you see it on the examples. If you mistype something i

```
void setup()
{
Serial.begin(9600);
Serial.print("Hi World");
}
void loop()
{
//code that goes here will repeat automatically over and over
}
```

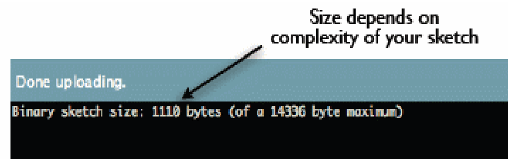
Also, notice in the figure that the sketch uses parentheses () and curly braces {}. Be sure to use the right ones in the right places!

Once you finish typing or copying the code connect your Arduino board (or robot) to the computer and select Verify. See the Status Message and Message Pane for a Done Compiling message.

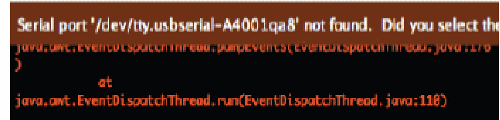
Figure: Arduino IDE



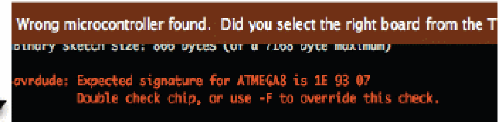
Uploading worked



Wrong serial port selected



Wrong board selected



Message pane messages: nerdy cryptic error messages

If there's a list of errors instead, it's trying to tell you it can't compile or convert your code. It is probably a typing error. Look at every letter, dot, comma, bracket carefully! Then click Verify again.

Once you can verify that is Done Compiling you are ready to upload to the Arduino board and make it run.

Click the **Upload** button (refer to the Arduino IDE figure above). The status line under your code will display "Compiling sketch…," "Uploading…," and then "Done uploading." **Note:** the sketch will not upload until you correct all errors and try to upload again. The Upload command compiles or converts your code to a format that the Arduino can understand and Upload compiles it and also sends it to the Arduino board **all in one command** so if you are comfortable with your code you can always just Upload so it does it all at one time.

- After the sketch is done uploading, click the **Serial Monitor** button (refer to previous figure)

- If the Hi World message doesn't display as soon as you open the Serial Monitor window opens, check for the "9600 baud" setting in the lower right corner of the monitor. (This is the speed at which the Arduino is sending out messages via the serial port (USB cable) to your screen, baud is bits per second). **Also, ensure the Arduino UNO board is selected and you can see the COM port as explained in the earlier steps of connecting your Arduino to your computer.**

- On the menu got to File → Save to save your sketch. Give it the name like HiWorld

The Code

Built in the Arduino programming environment Functions: Setup, Loop, Serial. begin, Serial. print

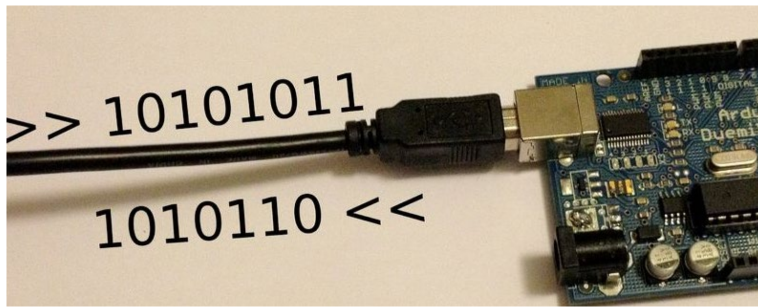
The Arduino language has two built-in functions: setup and loop. These two functions must be included in every sketch! The setup function is shown below. The Arduino executes the statements you put between the setup function's curly braces, but only once at the beginning of the program. Functions are preset instructions that perform a series of operations in it. The void setup( ) functions for example is in charge of settings up parts of the program that only need to run once in the beginning.

```
void setup()  // built in function
{
// anything inside these curly brackets only runs once
Serial.begin(9600);                // set the speed that the letters will output to the computer screen
Serial.print("Hi World");          //print on computer screen the words Hi World
}

void loop() // built in function
{
//code that goes here will repeat automatically over and over
}
```

In this example, both statements are function calls to functions in the Arduino's built-in Serial pre-written code library: **Serial.begin**(speed) and **Serial.print**(val). Here, speed and val are parameters, each describing a value that its function needs passed to it to do its job. The sketch provides these values inside parentheses in each function call. Arduino has **many** other built-in functions.

Serial.begin(9600); passes the value 9600 to the speed parameter. This tells the Arduino to get ready to exchange messages with the Serial Monitor at a data rate of 9600 bits per second. That's 9600 binary ones or zeros per second, and is commonly called a baud rate. This is the USB communication between the Arduino board and your computer.

Serial.print(val); passes the message "Hi World" to the val parameter. This tells the Arduino to send a series of binary ones and zeros to the Serial Monitor. The monitor decodes and displays that serial bit stream as the "Hi!" message.



Summary:

Anything you put under the void setup() function will only run once.

Anything you put under the void loop() function runs over and over forever or until you cut the power off.

Anything with the // signs indicates that it is a comment and is only used to tell the programmer or reader what that line of code will do. The Arduino compiler ignores it. (A *compiler* takes your sketch code and converts it into the microcontroller's native language.)

What is void? Why do these functions end in ( )? The first line of a function is its *definition*, and it has three parts: *return type*, *name*, and *parameter list*. For example, in the function **void setup()** the return type is void, the name is **setup**, and the parameter list is empty – there's nothing inside the parentheses ( ). *Void* means 'nothing'—when another function calls **setup** or **loop**, these functions would not return a value. An empty parameter list means that these functions do not need to receive any values when they are called to do their jobs. This idea will become important later on when you become more advanced and you want to pass on a specific parameter type to a function such as a number or a string (letters and words, specific values).

How to we make a Sketch repeat, adding Delays to process commands for a particular amount of time.

For many programs used in robotics we generally want to run them in a loop, meaning that one or more statements are repeated over and over again. For example, you want to scan an area with a sensor to avoid hitting something you need the process to repeat until something is seen or hit. Remember that the **loop** function automatically repeats any code in its *block.*

Let's try moving **Serial.print("Hi World");** to the **loop** function. To slow down the rate at which the messages repeat, let's also add a pause with the built-in **delay(*ms*)** function.

Save HiWorld as HiWorldRepeat.

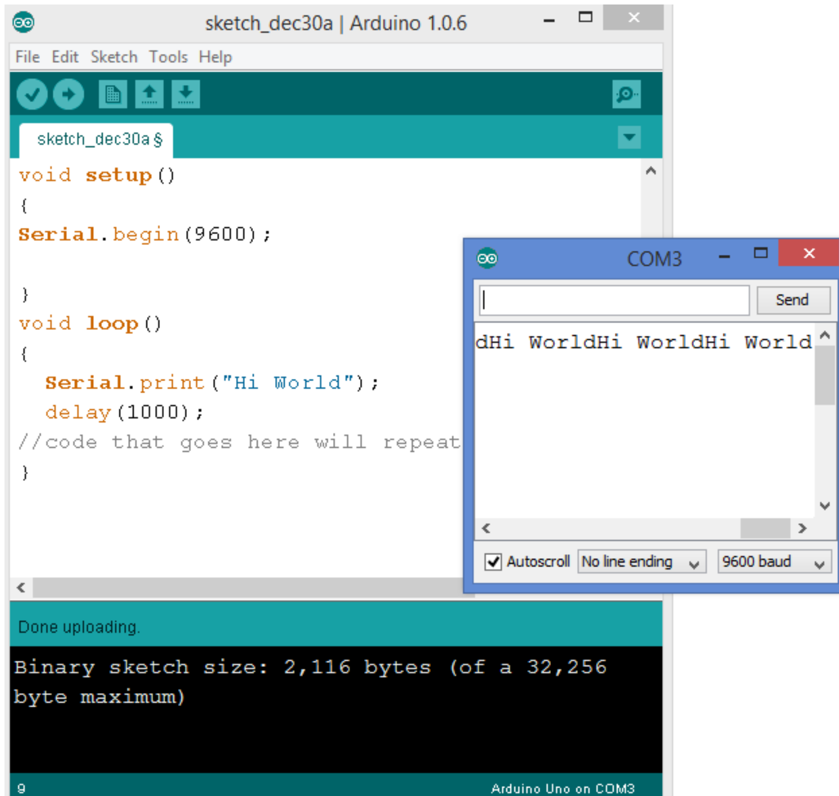
Move **Serial.print("Hi World");** from **setup** to the **loop** function.

Add the line **delay(1000);** on the next line.

Upload the sketch to the Arduino and then open the Serial Monitor again.

The added line **delay(1000)** passes the value 1000 to the **delay** function's millisecond ***ms*** parameter. It's requesting a delay of 1000 milliseconds. 1 ms is 1/1000 of a second. So, **delay(1000)** makes the sketch wait for one second before letting it move on to the next line of code.

Mastering delays is important because later on it will allow you to do precision moves and measurements.



A few more programming samples: Solving Math Problems

Arithmetic operators are useful for doing calculations in your sketch. We will try a few of them.

Assignment (=), addition (+), subtraction (-), multiplication (*) and division(/).

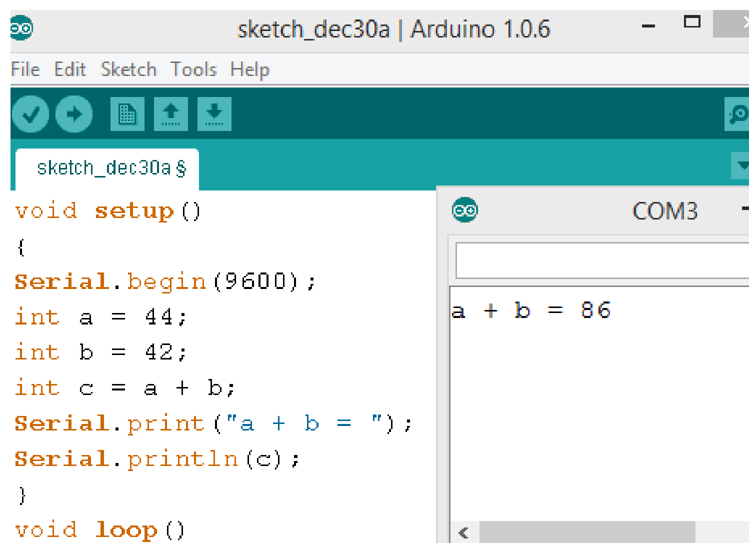
Open up the Arduino IDE and go to the Help menu. Select Reference and take a look at the list of Arithmetic Operators. You can click on each one to get an idea of how they work.

The next example sketch, addMath, adds the variables a and b together and stores the result in c. It also

displays the result in the Serial Monitor. Notice that c is now declared as an int, or integer. Integers are whole numbers (no decimal points or fractions). Another point, int c = a + b uses the assignment operator (=) to copy the result of the addition operation that adds a to b. The figure below shows the expected result of 44 + 42 = 86 in the Serial Monitor.

Type or copy the following program into your Arduino IDE and Upload it.

```
// addMath
void setup()
{
Serial.begin(9600);
int a = 44;
int b = 42;

int c = a + b;

Serial.print("a + b = ");
Serial.println(c);
}
void loop()
{
// Empty, no repeating code.
}
```



Variables, global and local declarations

Variables are ways of naming and storing a value for later use by the program. Variables can have changing values such as results collected from a sensor. So far we have learned how to declare variables with known values such as int a = 40; for example.

Variables can be declared globally or locally in the sketch. Global variables can be accessed in any scope (anywhere in the program) while local variables are good only for the scope where they are declared.

```
// example of declaring  global variable count
int count = 0;
void setup()
{
   Serial.begin(9600);
}

void loop()
{
   Serial.println(count);
   count++; //means add 1 every time it runs

   delay(250);
}
```

```
// example declaring local static variable count
void setup()
{
   Serial.begin(9600);
}

void loop()
{
    static int count = 0;
   Serial.println(count);
   count++;  //means add 1 every time it runs

   delay(250);
}
```

As your programs get larger you may want to declare your variables that can alter its value locally in the scope they are meant to otherwise you will get varied results as it value could change anywhere in your program.

So the decision which to choose come down to following arguments:

Generally, in computer science, it is encouraged to keep your variables as local as possible in terms of scope. This usually results in much clearer code with less side-effects. In the example above the variable count will only be used in the loop scope.

Global and static variables always occupy memory, whereas locals only do when they are in scope.

If you need to work with decimal point values, use **float**.

If you are using integer values (counting numbers), choose **byte**, **int**, or **long**.

If your results will always be an unsigned number from 0 to 255, use **byte**.

If your results will not exceed –32,768 to 32,767, an **int** variable can store your value.

If you need a larger range of values, try a **long** variable instead. It can store values from -2,147,483,648 to 2,147,483,647.

Numeric types	Bytes	Range	Use
int	2	-32768 to 32767	Represents positive and negative integer values.
unsigned int	2	0 to 65535	Represents only positive values; otherwise, similar to int.
long	4	-2147483648 to 2147483647	Represents a very large range of positive and negative values.
unsigned long	4	4294967295	Represents a very large range of positive values.

Numeric types	Bytes	Range	Use
float	4	3.4028235E+38 to -3.4028235E+38	Represents numbers with fractions; use to approximate real-world measurements.
double	4	Same as float	In Arduino, double is just another name for float.
boolean	1	false (0) or true (1)	Represents true and false values.
char	1	-128 to 127	Represents a single character. Can also represent a signed value between -128 and 127.
byte	1	0 to 255	Similar to char, but for unsigned values.
Other types			
string	Represents arrays of chars (characters) typically used to contain text.		
void	Used only in function declarations where no value is returned.		

Other variable types can be:

char b = ‘k’; //declares that character or letter b is k, not that is equal to k

float root5 = sqrt(3.0); //we use float here because we are dealing with decimal numbers

Doing Floating Point Math (dealing with decimal numbers)

- *Enter the Circumference sketch into the Arduino editor and save it.*
- *Make sure to use the values 0.75 and 2.0. Do not try to use 2 instead of 2.0.*
- *Upload your sketch to the Arduino and check the results with the Serial Monitor.*

```
//calculates circumference of a circle using floating point math
void setup()
{
Serial.begin(9600);
float r = 0.75; // this is a value we can change
float c = 2.0 * PI * r; //notice PI is built in to Arduino to it knows the value of 3.14…
Serial.print("circumference = ");
Serial.println(c);
}
void loop()
{
}
```

Making Decisions – using If/Else statements

Your Robot will need to make a lot of navigation decisions based on sensor inputs. Here is a simple sketch that demonstrates decision-making. It compares the value of **a** to **b**, and sends a message to tell you whether or not **a** is greater than **b**, with an **if…else** statement.

If the condition **(a > b)** is true, it executes the **if** statement's code block:

Serial.print("a is greater than b").

If **a** is *not* greater than **b**, it executes else code block instead: **Serial.print("a is not greater than b")**.

- Enter the code into the Arduino editor, save it, and upload it to the Arduino.
- Open the Serial Monitor and test to make sure you got the right message.
- Try swapping the values for a and b.
- Re-upload the sketch and verify that it printed the other message.

```
// decisions
void setup()
{
Serial.begin(9600); //displays values on the Serial Monitor
int a = 89;
int b = 42;

if(a > b)
{
Serial.print("a is greater than b");
}
else
{
Serial.print("a is not greater than b");
}
}
void loop()
{
// Empty, no repeating code.
}
```

More Decisions with if... else if (for multiple use as many else if as needed it: if…else if…else)

Maybe you only need a message when **a** is greater than **b**. If that's the case, you could cut out the **else** statement and its code block. So, all your **setup** function would have is the one **if** statement, like this:

```
void setup()
{
Serial.begin(9600);
int a = 89;
int b = 42;
```

```
if(a > b)
{
Serial.print("a is greater than b");
}
}
```

Maybe your sketch needs to monitor for three conditions: greater than, less than, or equal. Then, you could use an **if…else if…else** statement. Keep this example in mind as we do advanced experiments we will need to check multiple sensors at once like the 3 infrared line following sensors.

```
if(a > b)
{
Serial.print("a is greater than b");
}
else if(a < b)
{
Serial.print("b is greater than a");
}
else
{
Serial.print("a is equal to b");
}
```

So everything is ignored until it finds the correct comparison is looking for.

A sketch can also have multiple conditions with *relational operators* like && and ||.

The && operator means AND; the || operator means OR.

For example, this statement's block will execute only if **a** is greater than 50 AND **b** is less than 50:

```
if((a > 50) && (b < 50))
{
Serial.print("Values in normal range");
}
```

Another example: this one prints the warning message if **a** is greater than 100 OR **b** is less than zero.

```
if((a > 100) || (b < 0))
{
Serial.print("Houston, we have a problem!");
}
```

One last example: if you want to make a comparison to find out if two values are equal, you have to use two equal signs next to each other ==.

```
if(a == b)
```

```
{
Serial.print("a and b are equal");
}
```

= is used to assign a value

== is used to compare values

We will be using these a lot.

Count and Control Repetitions – for and while loops

Many robotic tasks involve repeating an action over and over again. Next, we'll look at two options for repeating code: the **for** loop and **while** loop. The **for** loop is commonly used for repeating a block of code a certain number of times. The **while** loop is used to keep repeat a block of code as long as a condition is true. These two functions will become some of the most important ones to understand.

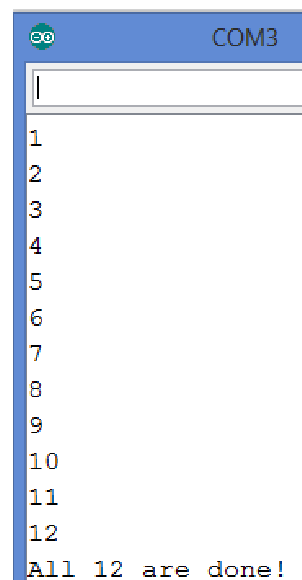
A for Loop is for Counting

A **for** loop is typically used to make the statements in a code block repeat a certain number of times. For example, your Robot will use five different values to make a sensor detect accurate distance, so it needs to repeat a certain code block five times. Or you want to set a value to multiple Arduino ports one at a time using just a couple of lines of code. For this task, we use a **for** loop.

Here is an example that uses a **for** loop to count from 1 to 12 and display the values in the Serial Monitor.

- Copy code and upload to Arduino board
- Open the Serial Monitor and verify that it counted from one to twelve.

```
// Count 1 To 12
void setup()
{
Serial.begin(9600); // setup speed to print on screen
for(int i = 1; i <= 12; i++)
// while I is less than or 12 continue counting
{
Serial.println(i); //print the value of i
delay(500); // wait half a second
}
Serial.println("All 12 are done!");
}
void loop()
{
// Empty, no repeating code.
}
```

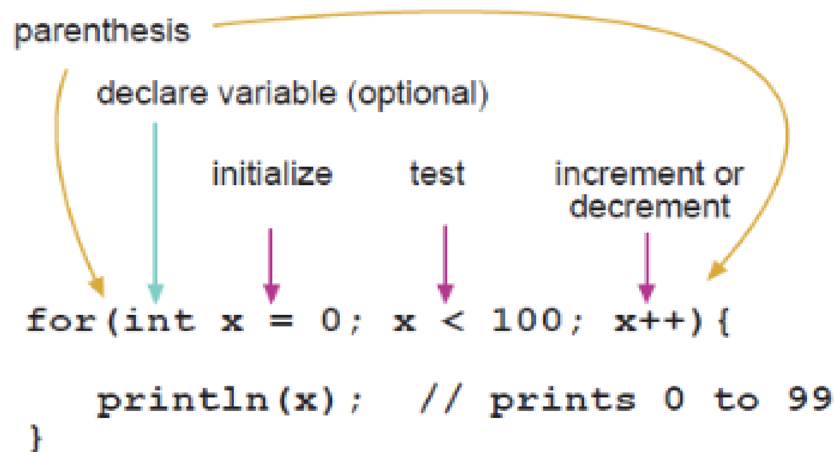


How the for loop Works

The way I like to understand the for loop is by saying that while x has not reached 100 go ahead and keep counting. X=0 means it starts counting from 0 and the x++ means keep adding 1 until 100 has been reached. It prints from 0 to 99 because 0 is still consider a number in computer terms.

There are three parts to the **for** loop header:

```
for (initialization; condition; increment) {
//statement(s);
}
```



If you would have said x<=100 then it would have printed 0-100 on the Serial Monitor. You can also select x- - to decrement or x+=5 will increment by 5 instead of one at a time.

Replace the for statement in the sketch with this:

```
for(int i = 5000; i <= 15000; i+=1000)
```

Upload the modified sketch and watch the output in the Serial Monitor.

A Loop that Repeats While a Condition is True

In robotics, it is useful using a while loop to keep repeating things while a sensor returns a certain value.

```
// Counting to 10 using the While condition
int i = 0;
while(i < 10)
{
Serial.println(++i);
delay(500);
}
```

Another While Example (do not run, for illustration purposed only until we are ready to wire parts on the robot)

This program waits to run until a button is pressed then it starts the blink an LED section in the loop. Keep this program in mind as it will be useful later on when we want to add a switch to our program to start only when the button is pressed. Note the INPUT_PULLUP in the pinMode function allows you to connect a switch reducing the circuit by one resistor. INPUT_PULLUP is like a virtual resistor.

```
int led = 13; //led connected to pin 13
int Button = 2; //push button connected to pin2

void setup() {
// set led as output
  pinMode(led, OUTPUT);
//push button connected to pin2 set as input
  pinMode(Button, INPUT_PULLUP);
  while(digitalRead(Button) == 1){  // wait until button is pushed
  }
}

void loop() {
  digitalWrite(led,HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(500);             // wait
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(500);             // wait

}
```

Let's look at the basics of electricity, electronics and components

Before we do our first experiments go ahead a look at some components. This is how they are represented in wiring or schematic diagrams. An important thing to notice is that some components have polarity, in other words the plus + or positive side of that **component must be connected to the + or positive side of the circuit/power source or else the component will be ruined!**

Electronic Components And Their Symbols.

Capacitor - Unpolarized

Capacitor - Polarized

Diode

LED

Optocoupler

Photoresistor

Pot (Potentiometer)

Relay

Resistor

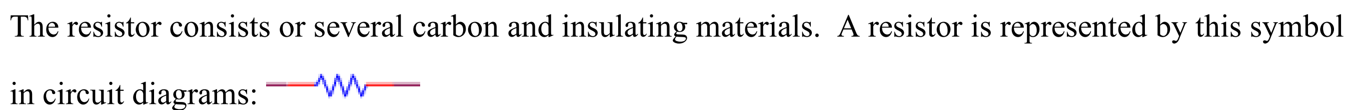
Stepper Motor

Switch

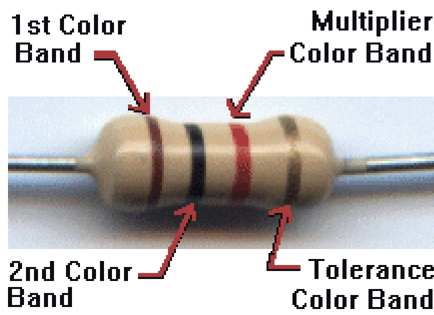
Transistor

Resistors act to reduce current flow or slows down the flow of electrons, and, at the same time, act to lower voltage levels within circuits.

The resistor consists or several carbon and insulating materials. A resistor is represented by this symbol in circuit diagrams:



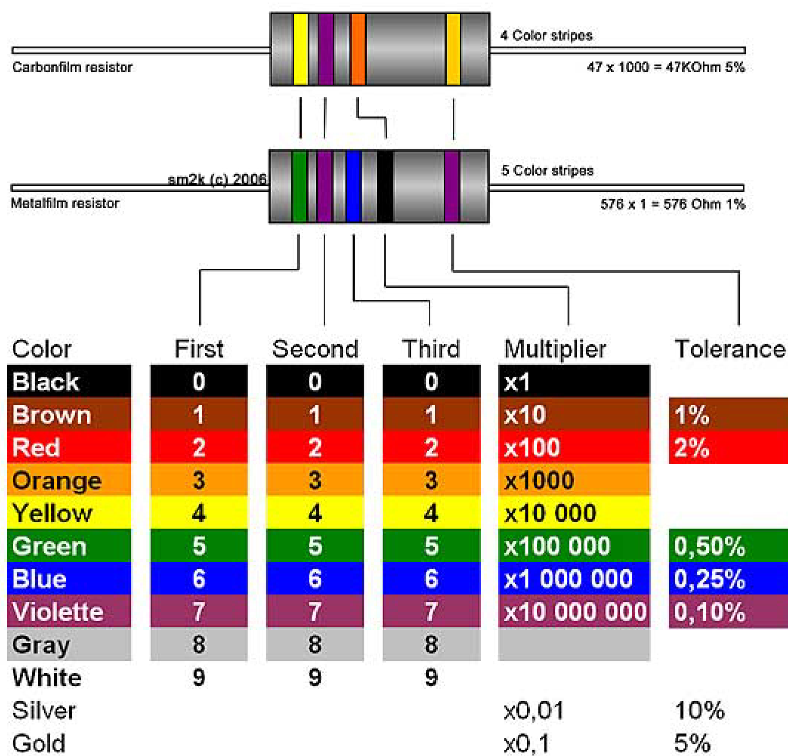
How to read a resistor value? You read the colors on the band from left to right. If the resistor is a 4



color stripes then the first and second band is the resistor value multiplied by the third band. The fourth band is the tolerance or how accurate the resistor value is.

For example, the resistor shown above brown/black/red has a value of 10x100 = 1000 ohms or 1K ohm. The gold colored band means a tolerance of 5% or +/- 5% of 1000. So this means that the resistor's value accuracy is between 950 and 1050 ohms.

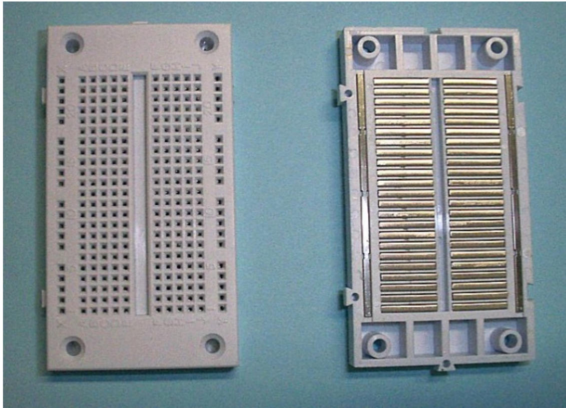
What is the value in ohm for a resistor with colors red/red/brown? **220 Ohms**

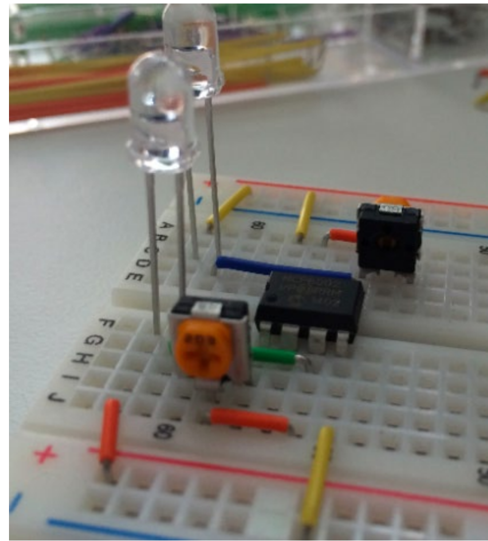


Color	First	Second	Third	Multiplier	Tolerance
Black	0	0	0	x1	
Brown	1	1	1	x10	1%
Red	2	2	2	x100	2%
Orange	3	3	3	x1000	
Yellow	4	4	4	x10 000	
Green	5	5	5	x100 000	0,50%
Blue	6	6	6	x1 000 000	0,25%
Violette	7	7	7	x10 000 000	0,10%
Gray	8	8	8		
White	9	9	9		
Silver				x0,01	10%
Gold				x0,1	5%

Breadboards are boards with metal strips on the bottom and holes on the top so you can temporarily build circuits on them to test before making the final circuit board.

Please note that individual metal strips on the bottom of the breadboards allows your to expand connect your components. Note: Pay attention how you wire them and makes sure you are using the same metal strip for all the legs of the same component!





On breadboards you can do temporary wirings of all your electronic components without creating any permanent connections or soldering. One of the most useful tools in an experimenter's toolkit!

Looking a little closer at the Arduino UNO Board

Digital and analog pins are the small pins on the Arduino module's Atmel microcontroller chip. These pins electrically connect the microcontroller brain to the board.
Atmel is a company that manufactures and sells microcontrollers. Arduino is really a combination of the software created by the open source community and hardware that includes the Atmel microcontroller that then gets installed into a protoboard or development board.

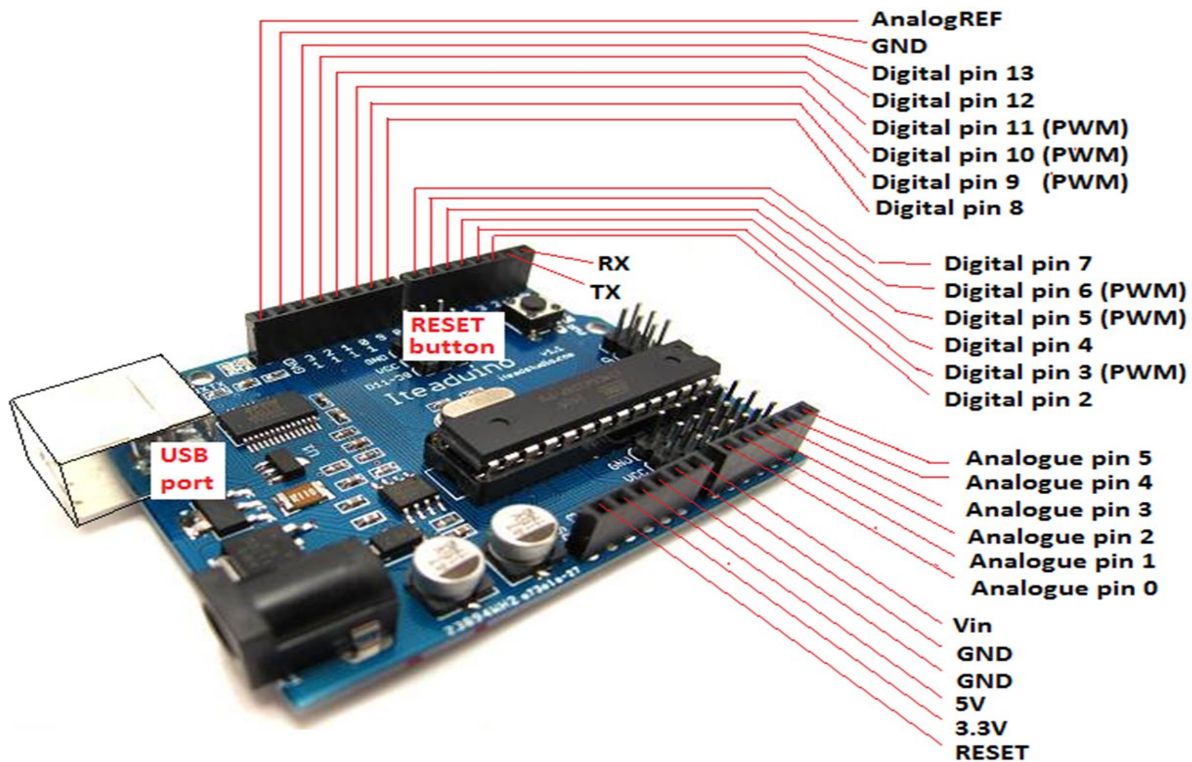


Fig. Arduino UNO development board showing the inputs and output pins.

When writing a sketch we can specify whether each pin will be an input or an output and whether the pin will have a zero (LOW) or a 5 volts or (HIGH) for example.

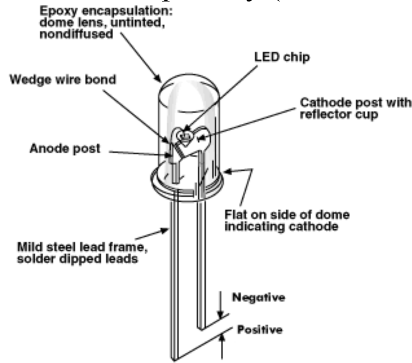
A sketch can also measure the voltages applied to analog pins; we'll do that to measure resistance change from a potentiometer or measure varied sensor values from a line following sensor in another experiment. One thing you will find is that most sensors are basically devices that acts as voltage dividers (meaning you are measuring the voltage difference between some points) that then convert those voltage values into meaningful values like temperature or distance.

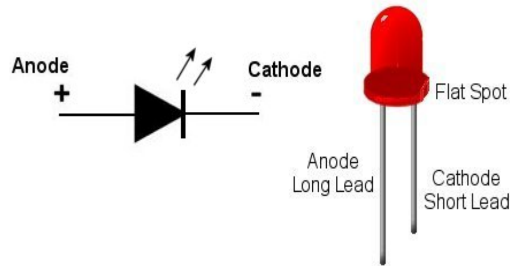
Note: always disconnect power to your board before building or modifying circuits or components might be ruined!

1. Set the Robot's Power switch to off or remove the power cable.
2. Disconnect the programming cable.

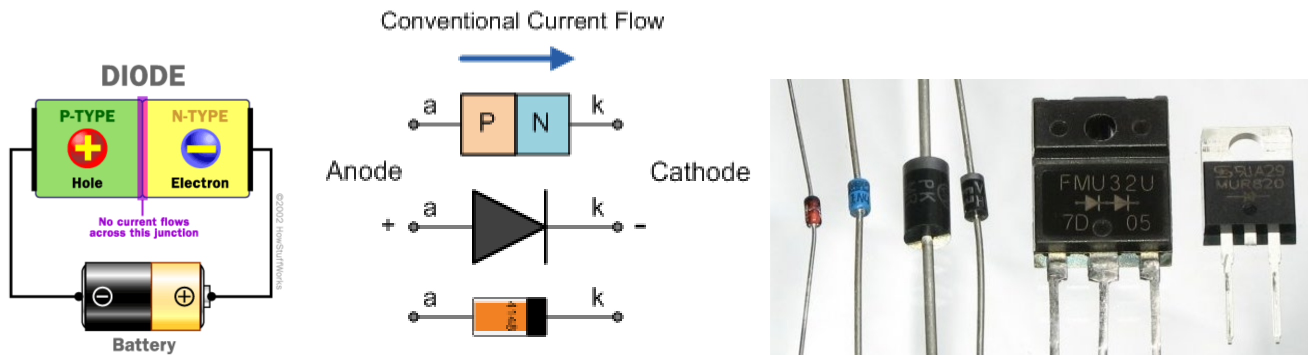
What are Light Emitting Diodes LEDs? When a fitting voltage is applied to the leads, electrons are able to recombine with electron holes within the device, releasing energy in the form of photons. Typically goes along with a resistor to prevent it from burning by too many electrons flowing through it too fast. A diode is a device or part with special material that allows the flow of electrons in only one direction. That is what the LED does, but it also emits light!

Please note the polarity (Anode = + and Cathode = -)





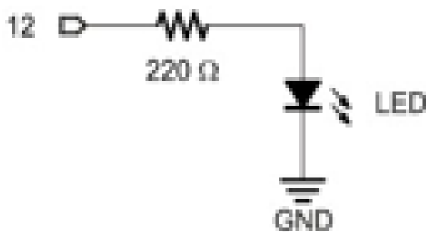
The Diode – allows the flow of electrons in only one direction. Is an electron traffic controller. Note: longer leg is the positive sign. The flat spot as shown on the graphic is the mins.



A good use for it in electronics is when you want to control polarity or divide parts of a circuit.

Building our first circuit and running a program– Blink and LED ON and OFF

Build the circuit shown below.

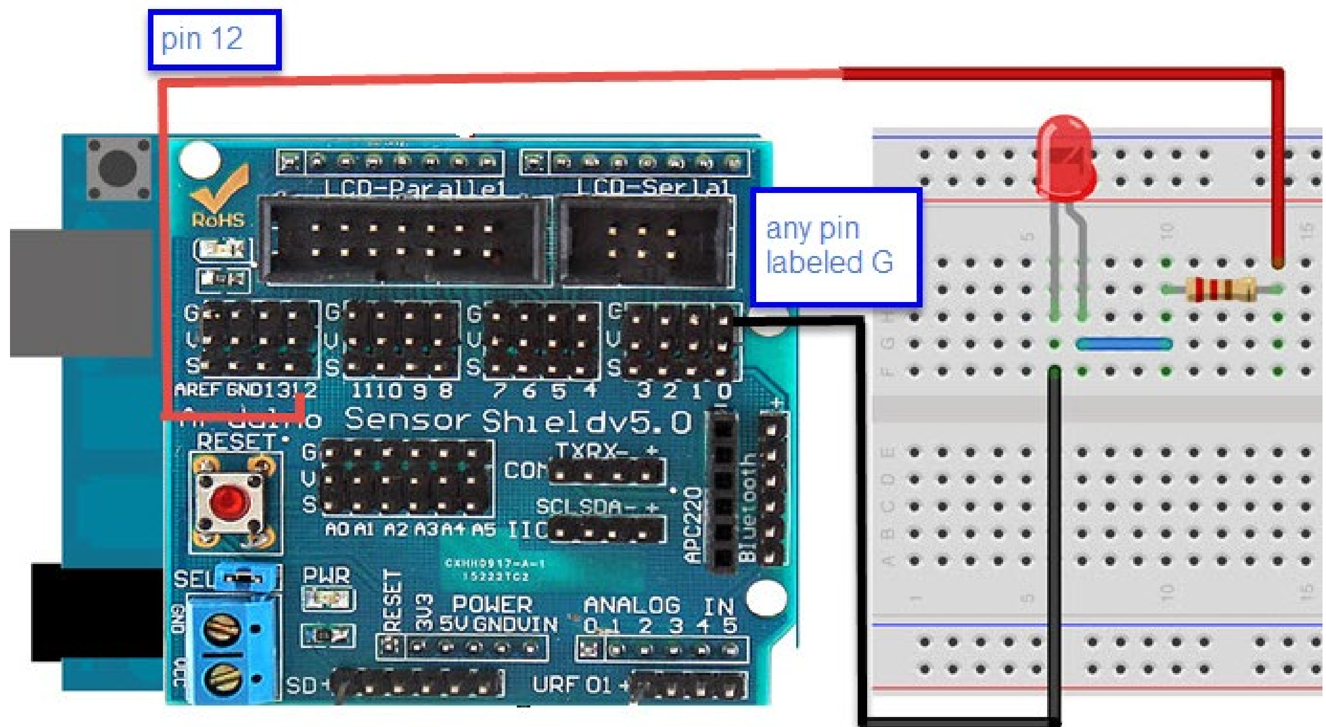


Note GND is Ground and labeled G on the Sensor Shield. You can use any pin labeled G on the Shield since they are all connected to the same place.

Parts:
LED
220 Ohm Resistor
DuPont wires M-F and M-M

Note: Be sure to plug in this wire on pin 12 labeled (S). This is the pin that provides a signal to the LED.

See graphic on next page for a graphical wiring representation.



Copy or type the following code. Plug in your Arduino board and verify or upload. For this part the Arduino is getting power from the USB cable and there is no need to turn the power switch ON

```
void setup()
{
pinMode(12, OUTPUT);
}

void loop()
{
digitalWrite(12, HIGH);          //send 5volts to pin 12
delay(500);                      // ..for 0.5 seconds
digitalWrite(12, LOW);           //send 0 volts to pin 12
delay(500);                      // ..for 0.5 seconds
}
```

Note: One thing to note here is that every time you upload a new sketch or program to your Arduino board the old program will be erased. You can only load and run one sketch at a time.

Verify that the pin 13 LED turns on and off, once every half a second.

Let's redo the program a little differently. Notice the long comment and how we can assign the value of pin 12 to a variable, in this case we named the variable led, but any name will do.

```
/* this is where you can write a nice long description about your program as long as you want to because we
are using an asterisk and forward slash */

int led = 12;
// we are saying that pin 12 is equal to variable named led, any name could be used for a variable

void setup()
{
pinMode(led, OUTPUT);   //pin 12 is now an output pin
}

void loop()
{
digitalWrite(led, HIGH);          //send 5volts to pin 12
delay(500);                       // ..for 0.5 seconds
digitalWrite(led, LOW);           //send 0 volts to pin 12
delay(500);                       // ..for 0.5 seconds
}
```

Note how we used the /* and then we end it with */ , this is the format used in the software to write long descriptions or comments about what the program will do. Anything inside those brackets will be ignored by the compiler. It is useful when documenting complex programs.

What is the advantage of using assigning pin 12 to a variable? Well, let's say you want to change the led to pin 5. You would not be able to do that easily without changing all of the pin 12 to number 5 on the code. By using variables you change the pin number in a single place and it affects it everywhere!
See here for a lot more on variable https://www.arduino.cc/en/Tutorial/Variables

Experimenting with changing time

Can you see how delay **(500)** is controlling the blink rate. This means the Arduino can send pulses out to control devices in a timed fashion. Want to change the blinking rate? Change the delay value to 1000 to blink every second or 6000 so it blinks every 6 seconds.

Digital Input – Getting a value (1 or 0) from outside and doing something with it.

Turning an LED on with a switch

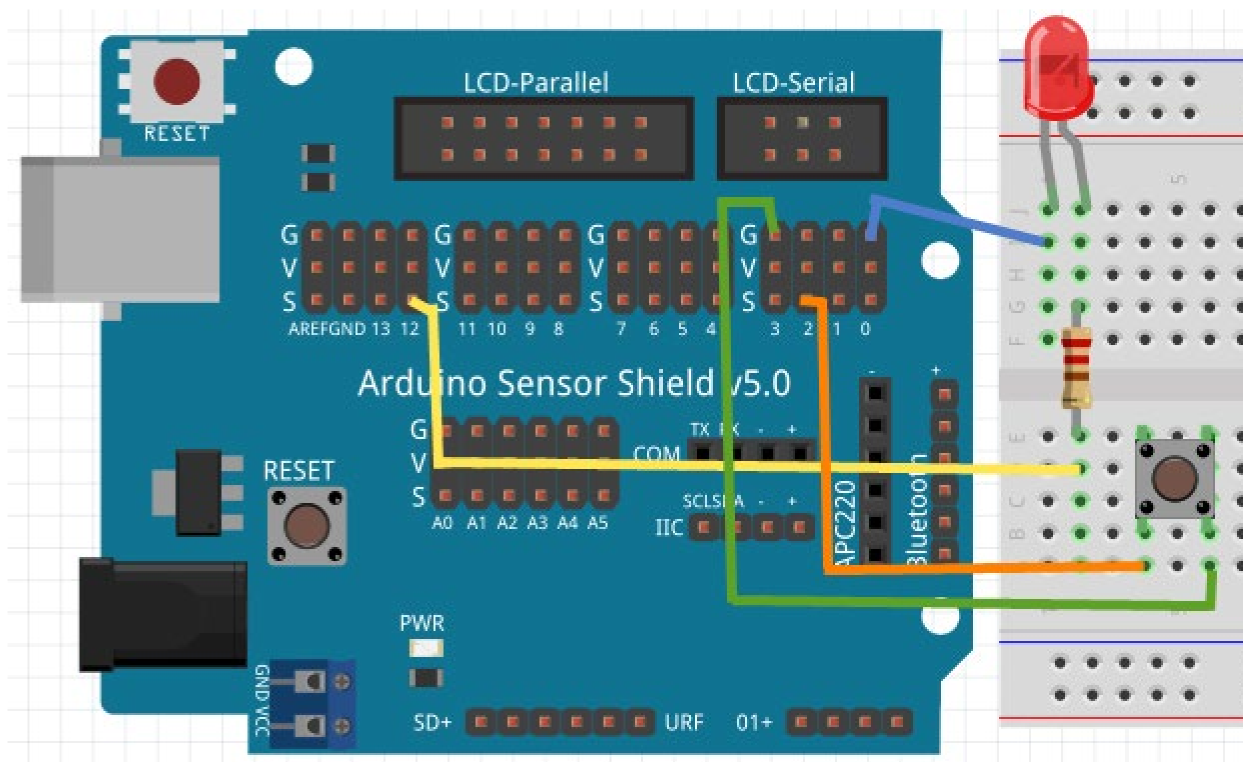
In the following part we will add a switch to our circuit so that the blinking does not start until you press a momentarily ON switch at least once. Connect the circuit as show below and copy and verify plus upload the sketch.

The idea is that once you upload the program you can press the button once and the program will start and will make the LED blink 6 times and will wait 6 seconds to do it again. This is useful when you want your robot to with a delayed start or you want the robot to do something when a simple sensor such as a switch to do something then triggered.

Switches are considered digital sensors since their state is either On or Off they provide a value of 0 or 1. We can look at those values, store them in a variable and perform things like motion or blink, etc.

We are going to use the digitalRead(Button) function.

More information about functions https://www.arduino.cc/en/Reference/FunctionDeclaration



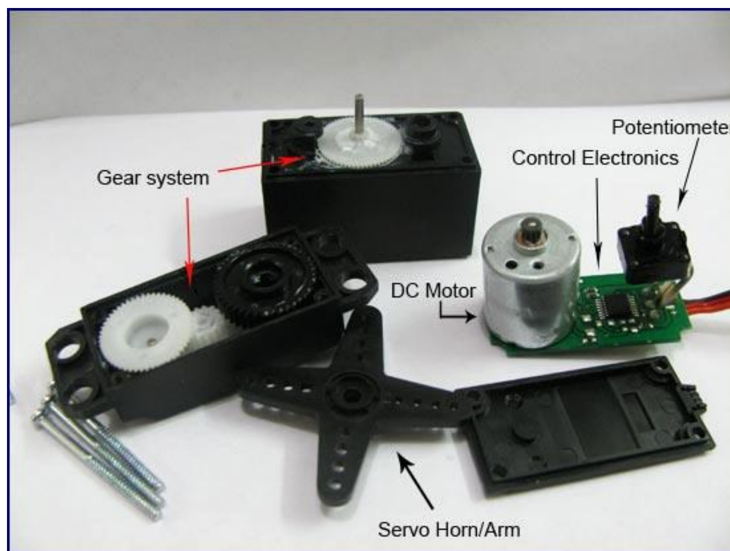
Wire the diagram above and type or copy code below and upload it to your robot.

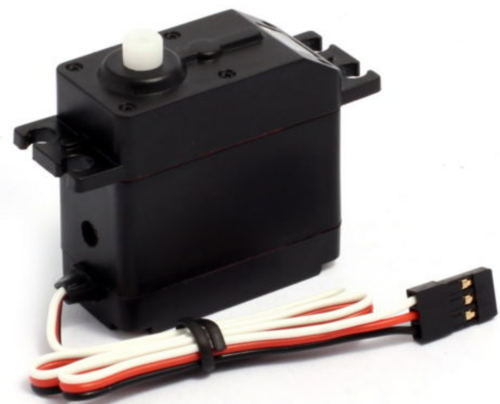
```
// wait for button to be pressed before continuing the program
// we will use a similar function later on when doing the sumobot where we will put the robots on the sumo
ring, but the robot does not start until you press the switch

int led = 12;       // assign pin 12 to LED
int Button = 2;   // push switch connected to pin 2
```

```
void setup()                    {
  pinMode(led, OUTPUT);                     //LED pin is an output pin
  pinMode(Button, INPUT_PULLUP);            //switch pin is an input pin
  while(digitalRead(Button))  {             // wait until button is pushed
  }
}

void loop()
 {
  for (int x=0; x <= 4; x++){                  //run the commands below 5 times (0 to 4)
    digitalWrite(led, HIGH);                   // turn the LED on (HIGH is 5volts to the pin)
    delay(1000);                               // wait for a second
    digitalWrite(led, LOW);                    // turn the LED off by making the voltage 0 volts
    delay(1000);                               // wait for a second
  }
  delay(6000);                                 //wait 6 seconds before running this loop again
}
```

Connecting and Controlling Servo Motors

This figure displays a continuous rotation hobby style servo motor showing the built in motor controller board, gears and potentiometer to help with the calibration.



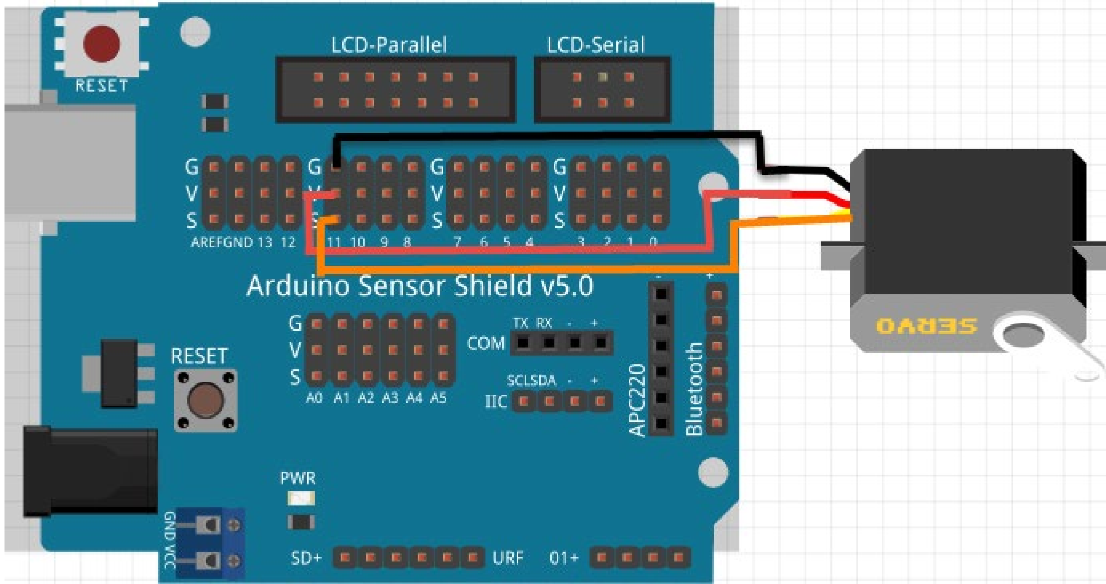


Servos have a circuit board that accepts high and low signals at various times. Depending on that signal the motor will rotate left/right/stop with variable speed. We can send these signals to the servos in a similar we send the signal to an LED.

Note: *Do not wire the following example as is for illustration purposed only.*

In the example below we are sending a high and low to pin 11 where the servo motor is connected to.

Servos only need three wires: +5volts (V), Ground (G) and the Signal (S) wire where the pulses are received. Servos already have gear mechanism inside in them so they provide higher torque making them strong to enough to move your robots. Servos also have a circuit board in them to accept commands for speed and direction.



Sample sketch to move the motor

```
void setup()
{
   pinMode(11, OUTPUT);                    // Set digital pin 13 -> output
}

void loop()                                // Main loop auto-repeats
{
   digitalWrite(11, HIGH);                 // Pin 11 = 5 V – motor turns
   delay(170);                             // ..for 0.17 seconds
   digitalWrite(11, LOW);                  // Pin 11 = 0 V, motor stops
   delay(1830);                            // ..for 1.83 seconds
}
```

How to Use the Arduino Servo Library

A better way to generate servo control signals is to include the Arduino Servo library in your sketch, one of the standard libraries of pre-written code bundled with the Arduino software.

To see a list of Arduino libraries, click the Arduino software's Help menu and select Reference or read more about the Servo library here https://www.arduino.cc/en/Reference/Servo

What are libraries?

Arduino libraries take a complex task and boil it down to simple to use functions. Arduino users have written lots of exciting add-ons for Arduino. For example, we can send the timing necessary to make a motor move by 180 degrees. The Servo library that comes with the Arduino program can already do that for us. Someone took the time to convert all of those timing values for use to just type in a single line of code that we want the servo motor to turn for 180 degrees. Very cool! Lots of companies that make sensors also create Arduino libraries to make it easier for you to write simple code to use their product. They figure it all for you. Of course you can modify the libraries to your liking as well.

Follow the instructions on how to install libraries here:

https://www.arduino.cc/en/Guide/Libraries

Read more about it here: https://learn.sparkfun.com/tutorials/installing-an-arduino-library

The main idea install the library, first quit the Arduino application. Then uncompress the ZIP file containing the library. For example, if you're installing a library called "ArduinoParty", uncompress ArduinoParty.zip. It should contain a folder called ArduinoParty, with files like ArduinoParty.cpp and ArduinoParty.h inside. (If the .cpp and .h files aren't in a folder, you'll need to create one. In this case, you'd make a folder called "ArduinoParty" and move into it all the files that were in the ZIP file, like ArduinoParty.cpp and ArduinoParty.h.)

Drag the ArduinoParty folder into this folder (your libraries folder). Under Windows, it will likely be called "My Documents\Arduino\libraries". For Mac users, it will likely be called "Documents/Arduino/libraries". On Linux, it will be the "libraries" folder in your sketchbook.

Your Arduino library folder should now look like this (on Windows):

```
My Documents\Arduino\libraries\ArduinoParty\ArduinoParty.cpp
My Documents\Arduino\libraries\ArduinoParty\ArduinoParty.h
My Documents\Arduino\libraries\ArduinoParty\examples
```

or like this (on Mac and Linux):

```
Documents/Arduino/libraries/ArduinoParty/ArduinoParty.cpp
Documents/Arduino/libraries/ArduinoParty/ArduinoParty.h
Documents/Arduino/libraries/ArduinoParty/examples
```

There may be more files than just the .cpp and .h files, just make sure they're all there. (The library won't work if you put the .cpp and .h files directly into the libraries folder or if they're nested in an extra folder. For example: Documents\Arduino\libraries\ArduinoParty.cpp and Documents\Arduino\libraries\ArduinoParty\ArduinoParty\ArduinoParty.cpp won't work.)

Restart the Arduino application. Make sure the new library appears in the Sketch->Import Library menu item of the software. That's it! You've installed a library!

You can also Import the Library – they usually come as a .zip file

Libraries are often distributed as a ZIP file or folder. The name of the folder is the name of the library. Inside the folder will be a .cpp file, a .h file and often a keywords.txt file, examples folder, and other files required by the library. Starting with version 1.0.5, you can install 3rd party libraries in the IDE. Do not unzip the downloaded library, leave it as is.

In the Arduino IDE, navigate to Sketch > Include Library. At the top of the drop down list, select the option to "Add .ZIP Library".

You will be prompted to select the library you would like to add. Navigate to the .zip file's location and open it.

Return to the *Sketch* > *Import Library* menu. You should now see the library at the bottom of the drop-down menu. It is ready to be used in your sketch. The zip file will have been expanded in the *libraries* folder in your Arduino sketches directory.

NB: the Library will be available to use in sketches, but examples for the library will not be exposed in the *File* > *Examples* until after the IDE has restarted.

This installing Arduino libraries tutorial based on text by Limor Fried.

Software libraries will provide you with functions that you can use to manage devices like sensors easier. Libraries can provide conversions for example from one type of value to another. In the Servo library for example there are timing issues that are taken care of for you. The Arduino provides pulses at a particular frequency, but servos require a specific frequency so the library takes care of all the conversion for you so that you can send simple human understood commands to make a servo move.

If you want to take a closer look at the Servo library. Find and follow the Servo link at C:\Program Files (x86)\Arduino\reference\Libraries.html

Servos and Signals

Servos have to receive high-pulse control signals at regular intervals to keep turning. If the signal stops, so does the servo. Once your sketch uses the Servo library to set up the signal, it can move on to other code, like delays, checking sensors, etc. Meanwhile, the servo keeps turning because the Servo library keeps running in the background. It regularly interrupts the execution of other code to initiate those high pulses, doing it so quickly that it's practically unnoticeable.

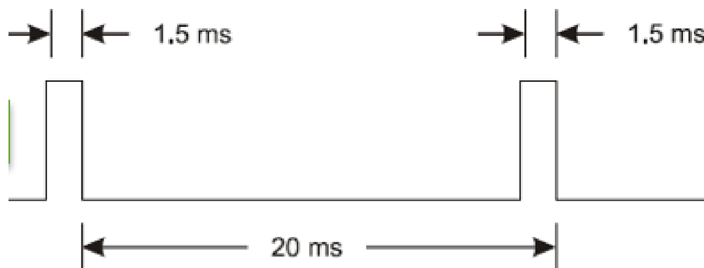
There will be lots of times where you will want to do multiple things "at once". At once really means splitting times at which each function runs at. The delays are so small that it feels like is doing multiple things "at once".

Before making the motors turn let's calibrate the servo motors

If you want all programs to perform the same from robot to robot then we need to ensure we have the servos accepting signals at a specific set standard. You may need to do this for every different project you do.

Servos like to receive a signal between 1300 and 1700 microsecond or 1.3 and 1.7 milliseconds every 20 milliseconds. (These values may vary per manufacturer of servos).

The signal halfway between the 1.7 ms full-speed- counterclockwise and 1.3 ms full-speed-clockwise pulses is 1.5ms and this particular signal causes the servo motors to stop moving.

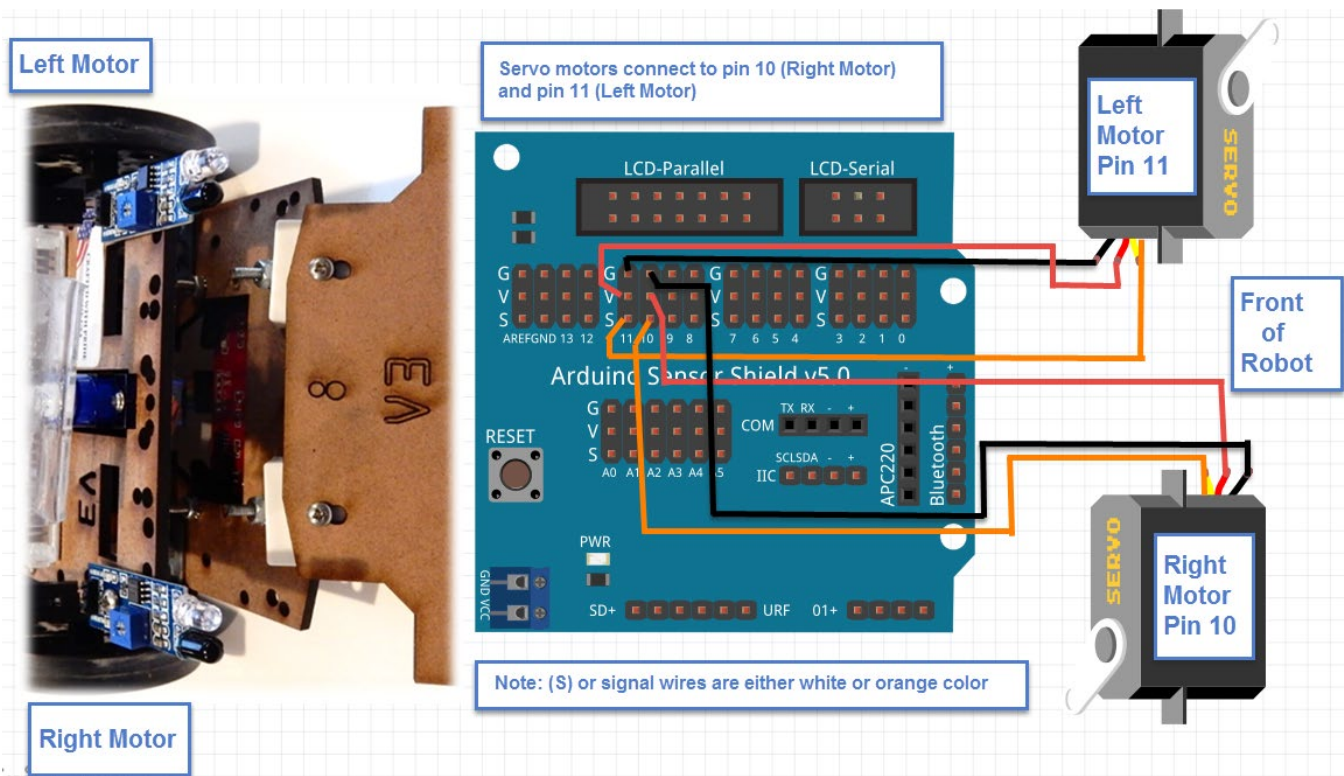


This signal for 1.5ms is universally understood as 'STOP' by the servo motors. What happens when you send a signal before or after 1.5ms varies by manufacturer.

The idea is that a 1.3ms pulse makes the servo turn in one direction, 1.5ms makes it stop and 1.7ms make it go the other direction. Any pulses in between 1.3-1.7 determine speed and direction.

Let's connect the servos to the robot.

Make sure your robot is turned OFF. Connect the servos on pins 10 and 11 of the Sensor Shield as shown on the next page. Disconnect any other parts you have like LEDs or switches.



Connect Copy and paste the following program and upload to robot. **Turn the robot's power switch ON to provide power to the motors from the battery.**

```
 /*   Calibrate_Servos - Generate signals of 1500ms long to make the servos stay still for centering  */
#include <Servo.h>         // Include servo library
Servo servoLeft;           // Declare left servo signal
Servo servoRight;          // Declare right servo signal
void setup()               // Built in initialization block
{
servoLeft.attach(11);               // Attach left signal to pin 11
servoRight.attach(10);              // Attach left signal to pin 10
servoLeft.writeMicroseconds(1500);              // 1.5 ms stay still sig, pin 11
servoRight.writeMicroseconds(1500);             // 1.5 ms stay still sig, pin 10
}
void loop()
{                    // Empty, nothing needs repeating
}
```

Are the robot wheels moving after you uploaded this program and turned the switch ON? If yes we need to Calibrate your motors!

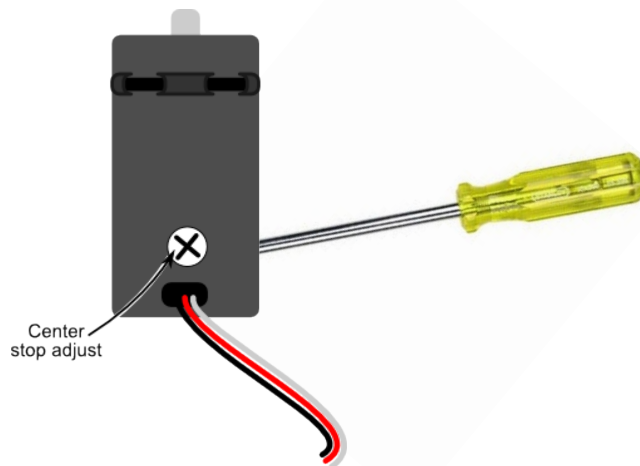
How do we calibrate the servo motors?

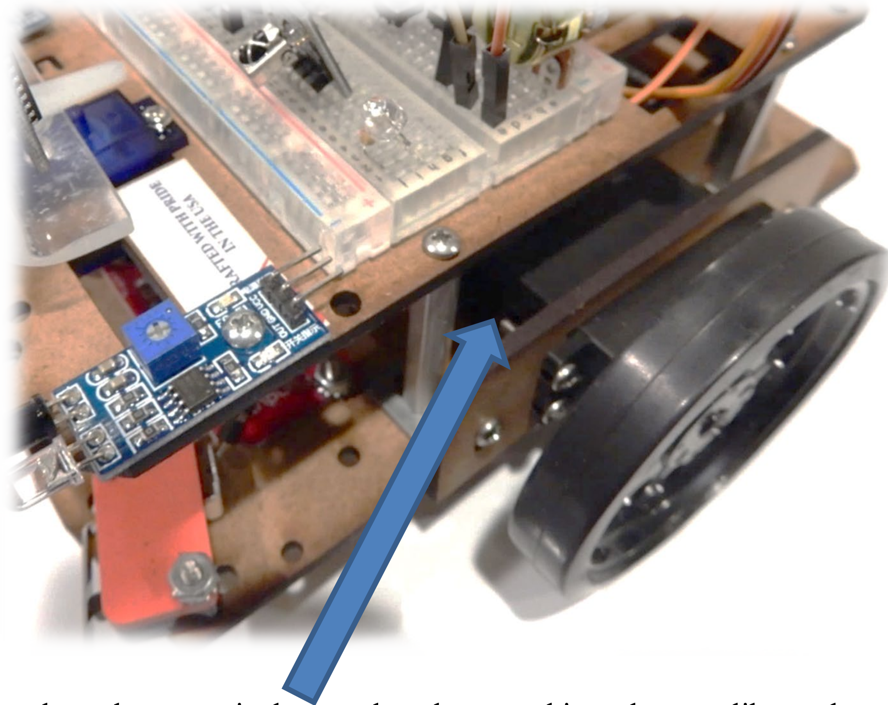
If a servo has not yet been centered it may turn, vibrate, or make a humming noise when it receives the stay still or stop moving signal of 1.5ms. This means the motors need to be calibrated or centered until the motor stops turning.

Make sure you upload the program and disconnect the USB cable from the robot. Turn the robot power **ON**. While holding the robot and you see the motors turning, use a screwdriver to *gently* adjust the potentiometer in the servo as shown below. Don't push too hard or you will **ruin** the potentiometer inside the servo! Adjust the potentiometer slightly until you find the setting that makes the servo stop turning, humming or vibrating. Once servos stop moving they are calibrated. Turn robot OFF and ON again to verify they are calibrated. See video on website on motor calibration for an example of how to do it.

Note: You may have to remove the aluminum spacer to access the potentiometer.

Note: You may have to calibrate every time a different program that uses the motors gets uploaded to the robot.





This where the screw is that needs to be turned in order to calibrate the motors

What's a Potentiometer?
A *potentiometer* is an adjustable resistor with a moving part, such as a knob or a sliding bar, for setting the resistance. This helps center the servo. This resistor adjustment helps you find the center where it will accept 1.5ms pulses to STOP the motor from moving.

Standard Servos vs. Continuous Rotation Servos
Standard servos are designed to receive electronic signals that tell them what position to hold. These servos control the positions of radio controlled airplane flaps, boat rudders, and car steering. Most standard servo motors can only rotate 180 degrees because this allows the ability to add a built-in sensor to determine its position, direction and speed at any one time. Continuous rotation servos receive the same electronic signals, but instead turn at certain speeds and directions continuously 360 degrees. Because continuous rotation servos can rotate all the way around they are handy for controlling wheels of mobile robots and pulleys. You will see later on the smaller servo included in the kit is a standard servo motor and only rotates 0-180 degrees.

Testing both Motors and direction using your computer keyboard
Note: You can also download the following program under Experiment1 at http://roboticscity.com This program will help you make sure your robot's servos are wired properly too.
Upload the program shown below to the robot. When done turn robot ON, but leave it connected to the USB port. We are going to send commands via our computer keyboard to move the servo motors. While the program is running and USB cable is connected open the Serial Monitor window and at the command prompt you can type the letters shown on the next page and hit Enter.

If the motor continues to rotate after you type the letter s and hit Enter then you need to calibrate the servo as we did previously. While holding the robot and you see the motors turning, use a screwdriver to *very gently* adjust the potentiometer in the servo until it stops.

Keyboard commands:
w and Enter for forward
s and Enter to stop
b and Enter for back
a and Enter for left
d and Enter for right

```
// keyboardControl.ino
// great program to test your robot using the keyboard to go forward, left, right, back and stop
#include <Servo.h>

 Servo leftServo;
 Servo rightServo;

 int LEFT_FORWARD_VALUE = 180;
 int LEFT_BACK_VALUE = 0;
 int STOP_VALUE = 90;
 int RIGHT_FORWARD_VALUE = 0;
 int RIGHT_BACK_VALUE = 180;
 int LEFT_SERVO_PIN = 10;
 int RIGHT_SERVO_PIN = 11;

 void setup()
 {
  Serial.begin(9600);
  leftServo.attach(LEFT_SERVO_PIN);
  rightServo.attach(RIGHT_SERVO_PIN);
  stop();
 {
     // you can put a while loop here to look for a switch before continuing
  }
 }
 //========================================
 void forward()
 {
   leftServo.write(LEFT_FORWARD_VALUE);
   rightServo.write(RIGHT_FORWARD_VALUE);
 }
 //========================================
 void stop()
 {
   leftServo.write(STOP_VALUE);
   rightServo.write(STOP_VALUE);
 }
 //========================================
 void back()
 {
   leftServo.write(LEFT_BACK_VALUE);
   rightServo.write(RIGHT_BACK_VALUE);
```

```
}
//=======================================
void left()
{
  leftServo.write(LEFT_BACK_VALUE);
  rightServo.write(RIGHT_FORWARD_VALUE);
}
//=======================================
void right()
{
  leftServo.write(LEFT_FORWARD_VALUE);
  rightServo.write(RIGHT_BACK_VALUE);
}
//=======================================
void loop()
{

 if (Serial.available() > 0) {
  int inByte = Serial.read();

  switch (inByte) {
  case 'w':   forward(); break;
  case 'b':   back(); break;
  case 'a':   left(); break;
  case 'd':   right(); break;
  case 's':   stop(); break;
  }
 }
}
```

How this program will help you

Need to know if you got the motors connected correctly? Type w and enter and see if the robot moves forward. You can test all directions this way and ensure the robot will do what you ask it to in the next parts of this guide. You can also use it to calibrate the servos since 's' is supposed to stop the robot. If it does not then you need to calibrate the servos.

Short Explanation

This part of the code waits for a character to be typed by the keyboard via the USB port.

```
 if (Serial.available() > 0) {
  int inByte = Serial.read();
```

In this part of the code we use the switch () function. When it senses the character we entered it switches over to that function under the case.

```
 switch (inByte) {
  case 'w':   forward(); break;
  case 'b':   back(); break;
  case 'a':   left(); break;
  case 'd':   right(); break;
```

```
    case 's':    stop(); break;
```

Since we are using the Servo.h library we can define the values of motion I degrees like 0, 90 and 180

```
int LEFT_FORWARD_VALUE = 180;
 int LEFT_BACK_VALUE = 0;
 int STOP_VALUE = 90;
 int RIGHT_FORWARD_VALUE = 0;
 int RIGHT_BACK_VALUE = 180;
```

Let' try some individual functions like forward, back, left and right using both values in degrees like the program we ran previously or values in microseconds which can help us set the speed on the servos instead of all ON or all OFF.
Run the program below (make sure you turn the power switch On after program has been uploaded).

Making the robot go forward using microseconds

```
/* Servos attached in OppositeDirections Make robot go forward, this configuration is called
differential steering or tank steering */

#include <Servo.h>     // Include servo library

Servo servoLeft;         // Declare left servo signal
Servo servoRight;        // Declare right servo signal

void setup()
{
servoLeft.attach(11);  // Attach left signal to pin 9
servoRight.attach(10);           // Attach right signal to pin 8

servoLeft.writeMicroseconds(1700);            // 1.7 ms -> counterclockwise
servoRight.writeMicroseconds(1300);           // 1.3 ms -> clockwise
}

void loop()
{         // Empty, nothing needs repeating
}
```

The advantage of using Microseconds is that you can play with the numbers between 1300-1700 to change the speed while using degree as shown in the next example you are moving the motors at full speed.

This opposite-direction control will be important soon. Think about it: when the servos are mounted on either side of a chassis, one will have to rotate clockwise while the other rotates counterclockwise to make the robot roll in a straight line. Does that seem odd? If you can't picture it, try this:

Hold your robot while the sketch is running and you will see this.

Making the robot go forward using degrees instead of microseconds (moves motor at full speed)

```
/* Servos attached in OppositeDirections Make robot go forward, this configuration is called differential steering
or tank steering */

#include <Servo.h>        // Include servo library

Servo servoLeft;          // Declare left servo signal
Servo servoRight;         // Declare right servo signal

void setup()
{
servoLeft.attach(11);     // Attach left signal to pin 9
servoRight.attach(10);    // Attach right signal to pin 8

   servoLeft.write(180);
   servoRight.write(0);
}

void loop()
{
// Empty, nothing needs repeating
}
```

Note: Speed could be adjusted by changing the degree values 0-180, but you have to experiment with the values.

Pulse Width Modulation – a way to vary the voltage output using square waves tuning on and off very rapidly. A very popular method used today to control just about anything from the intensity of light to the precise motion of motors.

Controlling electrical power through a load by means of quickly switching it on and off, and varying the "on" time, is known as pulse-width modulation, or PWM. It is a very efficient means of controlling electrical power because the controlling element (the power transistor) dissipates comparatively little power in switching on and or, especially if compared to the wasted power dissipated of a rheostat in a similar situation. When the transistor is in cutoff, its power dissipation is zero because there is no current through it. When the transistor is saturated, its dissipation is very low because there is little voltage dropped between collector and emitter while it is conducting current. Can also be used to control brightness of DC light bulbs, temperature of heaters and heating elements, etc.

PWM, is a technique for getting analog results with digital means

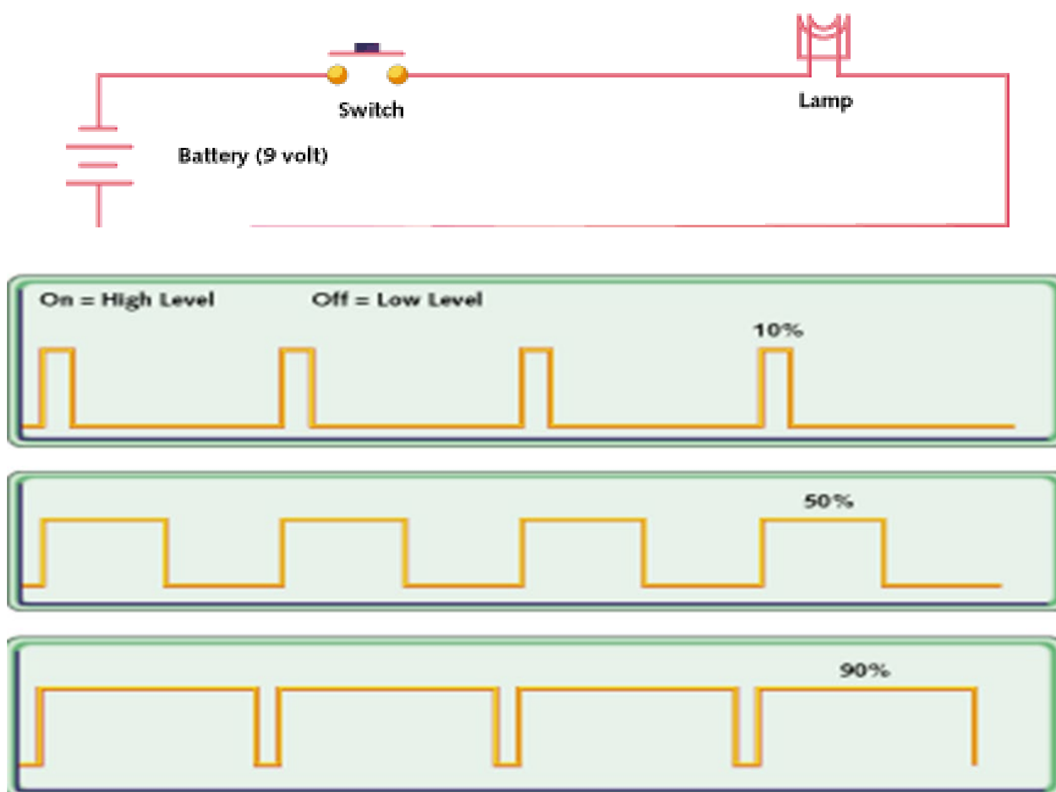
Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the portion of the time the signal spends on versus the time that the signal spends off.

The duration of "on time" is called the pulse width or duty cycle. To get varying analog values, you change, or modulate, that pulse width.

If you repeat this on-off pattern fast enough with an LED for example, the result is as if the signal is a steady voltage between 0 and 5v controlling the brightness of the LED.

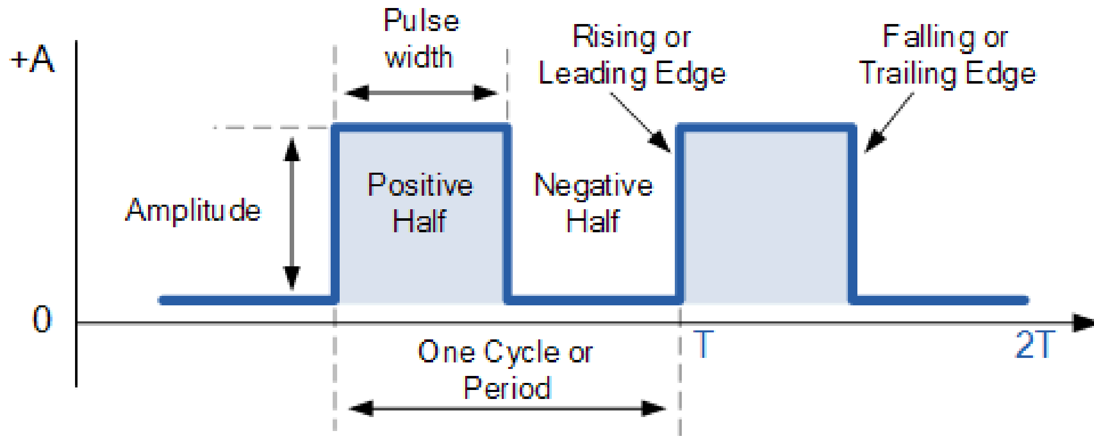
Servo motors have a built-in motor controller circuitry that accepts these fast pulses. Depending on how long the pulse stays on for will determine the speed and direction that the servo motor turns.

Before we had microcontrollers that could put out fast pulses we had manual switches that did not give us the ability to send exact timed pulses so when we wanted to control something it was either on or off.

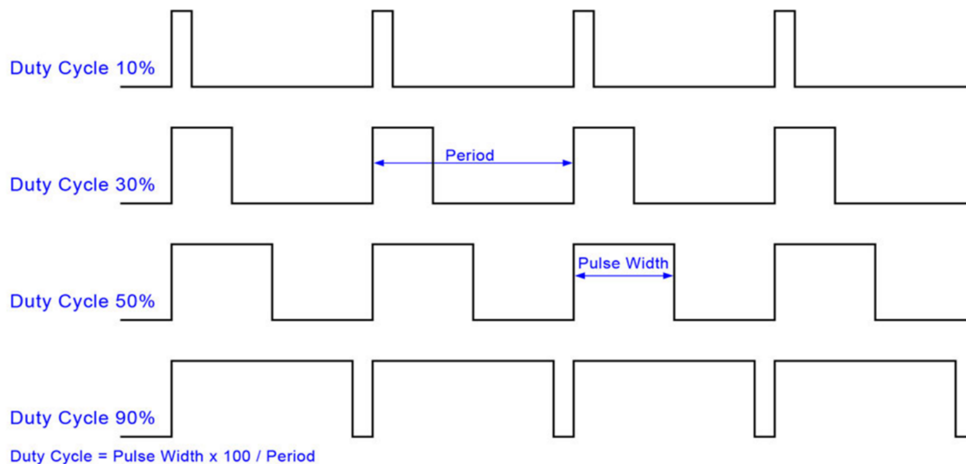


Press button manually really fast to get 10% output from the 9 volts. 10% means is outputting .9 volts if it is coming out of a 9 volt battery or do the same with a microprocessor like the Arduino and you can get an exact 10% output.

Anatomy of a Pulse



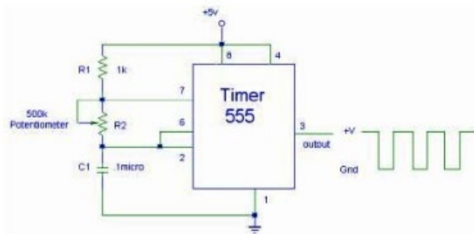
Pulse Width is the Duty Cycle. One complete cycle is called a period. If you have a 50% duty cycle then you are only outputting 50% of the power.



Pulse Width is how much voltage (Amplitude) is being put out.

Period is the total duration of the pulse and also the total amount of voltage that is able to put out.

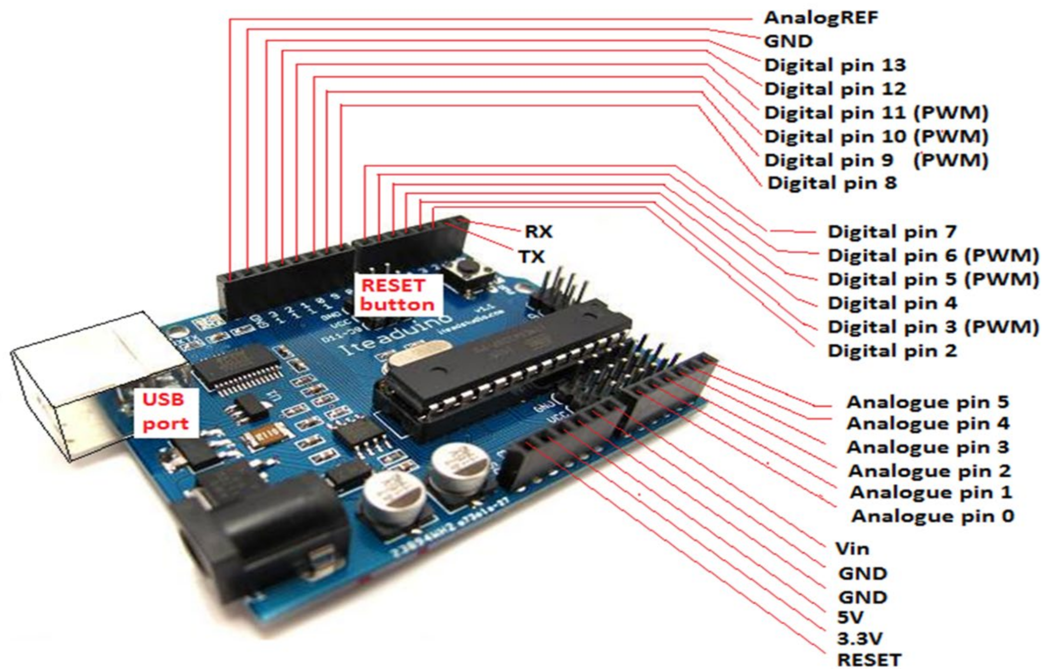
Duty cycle is the total percentage of the voltage you are putting out.



The Arduino has several timers that are capable of controlling the timing of these pulses.

Arduino produces its PWM at about 500 Hz or one cycle every 200 milliseconds

You can send PWM signals directly out of some of the Arduino pins as shown below pins labeled (PWM). You can send a variable voltage to those pins 0-5volts by using values of 0-255.



In this picture you can see how the Arduino UNO board has 6 pins assigned to be able to provide a PWM output. In other words you can output a varied voltage on that pin between 0-5 volts and in code is a value between 0-255.

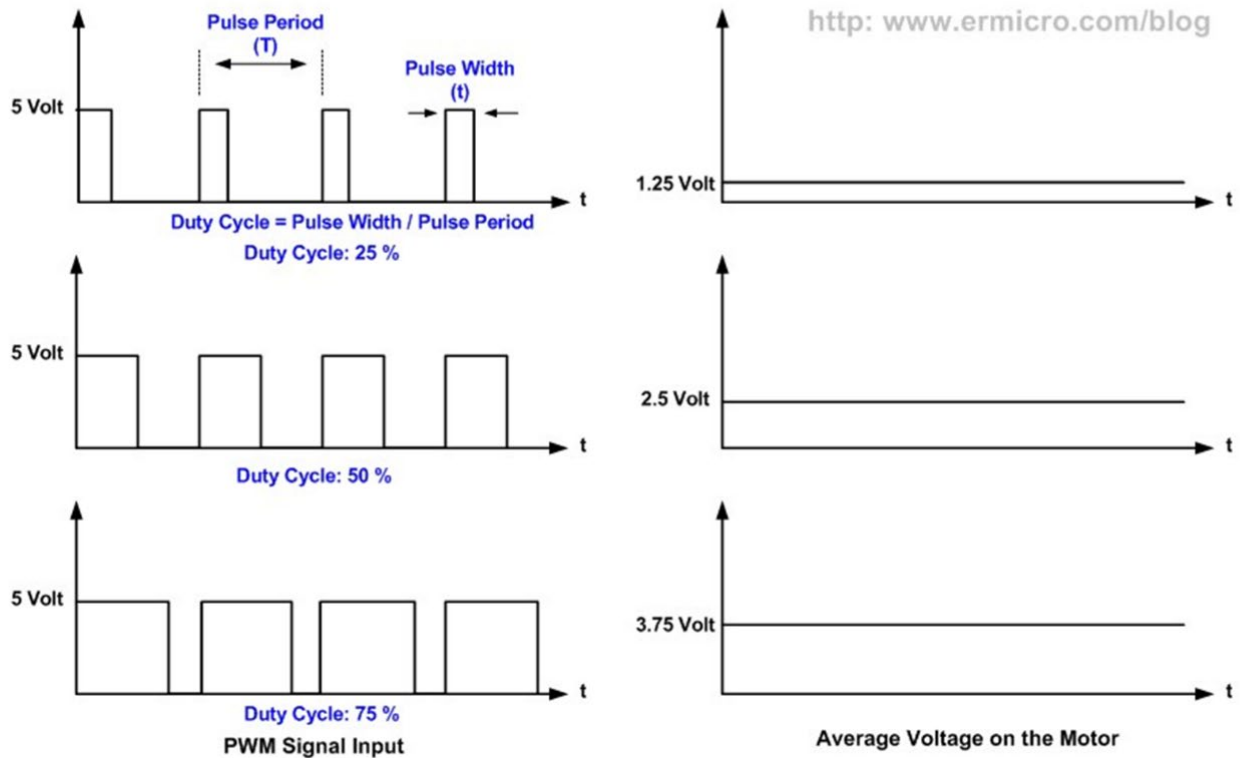
Example for sending a 50% duty cycle pulse using the function analogWrite ()

analogWrite(ledPin, fadeValue or duty cycle);

analogWrite(11, 127); // analogWrite(pin, value) send PWM between 0 and 255

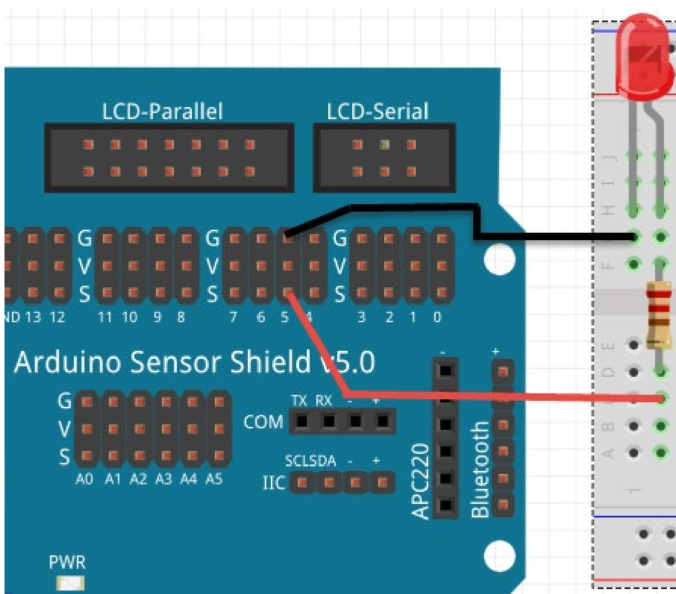
analogWrite() This function invokes the Pulse Width Modulation capabilities of our Arduino board. Pulse Width Modulation basically adjusts the power output at the pin. So you can have a lot of power or a little power applied at the pin, its' your call, you just tell the analogWrite() function which pin to modulate and how much power you want applied. The scale is from 0 to 255 with zero being the lowest power setting and 255 being the highest.

You can utilize pins 3, 5, 6, 10 and 11 with analogWrite() on most Arduino boards (there is a “PWM” or ” ~” next to the pin number on the board). The value range is from 0 to 255 (8bit).



PWM Timing Diagram

Here is a quick example of how to vary a voltage out using PWM to fade an LED. Connect a resistor and an LED to pin 5 of your Sensor Shield. You can leave your motors connected. Turn robot OFF before wiring this sample. *Positive side of the LED connects to pin 5 (S) signal pin.*



```
// fades an LED from min to max then max to min: PWM example
int ledPin = 5;    // LED connected to digital pin 5
void setup()
{
 // nothing happens in setup
}
void loop()
{
 // fade in from min to max in increments of 5 points:
 for(int fadeValue = 0 ; fadeValue <= 255; fadeValue +=5)
 {
  // sets the value (range from 0 to 255):
  analogWrite(ledPin, fadeValue);
  // wait for 30 milliseconds to see the dimming effect
  delay(30);
 }
 // fade out from max to min in increments of 5 points:
 for(int fadeValue = 255 ; fadeValue >= 0; fadeValue -=5) {
  // sets the value (range from 0 to 255):
  analogWrite(ledPin, fadeValue);
  // wait for 30 milliseconds to see the dimming effect
  delay(30);
 }   }
```

Continuing with the Robot Moves and Timing

Making sure your robot is connected upload this program and verify the robot is turning. Make sure to turn the robot's switch to ON so the motors get power from the battery pack. You can remove the USB cable and put it on the floor to test it.

```
/* ServoRunTimes
Generate a servo full speed counterclockwise signal with pin 11 and full speed clockwise
signal with pin 10 for 3 seconds
*/

#include <Servo.h>                          // Include servo library

Servo servoLeft;                            // Declare left servo signal
Servo servoRight;                           // Declare right servo signal

void setup()                                // Built in initialization block
{
servoLeft.attach(11);                       // Attach left signal to pin 11
servoRight.attach(10);                      // Attach right signal to pin 10


servoLeft.writeMicroseconds(1300);
servoRight.writeMicroseconds(1300);
delay(3000);                                // ..for 3 seconds
servoLeft.writeMicroseconds(1700);
servoRight.writeMicroseconds(1700);
```

```
delay(3000);                                 // ..for 3 seconds

servoLeft.writeMicroseconds(1500);           // Pin 11 stay still
servoRight.writeMicroseconds(1500);          // Pin 10 stay still
}

void loop()
{
}
```

Try changing the values of 1300 and 1700 and see what results you get. Every time you make a change you need to upload the program again.

When done be sure to turn the robot OFF

Servo Troubleshooting

Here is a list of some common symptoms and how to fix them.
The first step is to double check your sketch and make sure all the code is correct and that your servos are connected to the right pins.
If the code is correct, find your symptom in the list below and follow the checklist instructions

The servo doesn't turn at all or keeps turning even after is calibrated?

Double-check your servo connections using the wiring diagram as a guide.
Make sure the battery pack has fresh batteries, all oriented properly in the case. Sometimes you have to move the batteries by rolling them in place so they make a good connection.
The left servo turns when the right one is supposed to.
This means that the servos are swapped. The servo that's connected to pin 10 should be connected to pin 11, and the servo that's connected to pin 11 should be connected to pin 10.
Disconnect power—both battery pack and programming cable.
Unplug both servos.
Reconnect power.
Re-run the sketch.
Unplug everything else except the servos to make sure something else is not causing a short.

The wheel does not fully stop; it still turns slowly

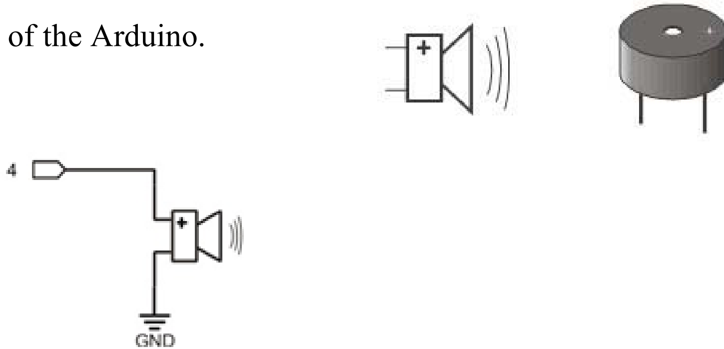
If the wheel keeps turning slowly after the clockwise, stop, counterclockwise sequence, it means that the servo may not be exactly centered.

The wheel never stops, it just keeps turning rapidly

If you are sure the code in your sketch is correct, it probably means your servo is not properly centered. Repeat the exercise in calibrating the Servos.

Making Sounds with the Arduino

We'll use a device called a *piezoelectric* speaker (piezospeaker) that can make different tones depending on the frequency of high/low signals it receives from the Arduino. The schematic symbol and part drawing are shown below. Note: there is a + and a – on the speaker. The + goes to the Signal pin of the Arduino.

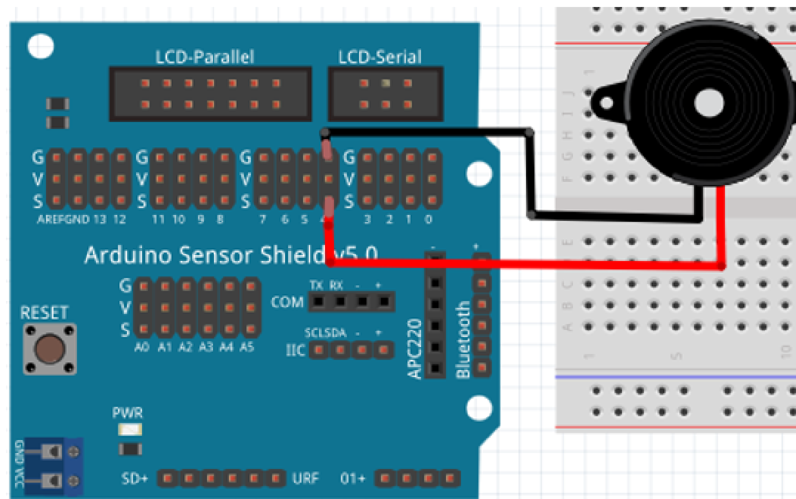


Make sure robot is turned OFF and disconnected then connect your speaker following the figure below using Pin4 for the Signal pin.

Installing the speaker

Parts:
(1) piezo buzzer
(2) M-F DuPont wires

While you can use any color wires you want, you still need to be careful that the positive (+) side is connected to pin 4 signal (S) pin. Connect the other wire to any available ground (G) pin of the sensor shield



13

Frequency is the measurement of how often something occurs in a given amount of time.

A piezoelectric material is a crystal that changes shape slightly when voltage is applied to it. Applying high and low voltages at a rapid rate causes the crystal to rapidly change shape. The resulting vibration in turn vibrates the air around it, and this is what our ear detects as a tone. Every rate of vibration makes a different tone.

Piezoelectric elements have many uses. When force is applied to a piezoelectric element, it can create voltage. Some piezoelectric elements have a frequency at which they naturally vibrate. These can be used to create voltages at frequencies that function as the clock oscillator for many computers and microcontrollers.

Always disconnect power before building or modifying circuits!

The next example sketch tests the piezo speaker using calls to the Arduino's **tone** function. True to its name, this function send signals to speakers to make them play tones.

There are two options for calling the **tone** function. One allows you to specify the *pin* and *frequency* (pitch) of the tone. The other allows you to specify ***pin***, ***frequency***, and ***duration*** (in milliseconds). We'll be using the second option since we don't need the tone to go on indefinitely.

tone(*pin*, *frequency*)
tone(*pin*, *frequency*, *duration*)

Example:
```
tone(4, 3000, 1000);
delay(1000);
```

That will make pin 4 send a series of high/low signals repeating at 3 kHz (3000 times per second). The tone will last for 1000 ms, which is 1 second. The tone function continues in the background while the sketch moves on to the next command.

Frequency can be measured in hertz (Hz) which is the number of times a signal repeats itself in one second. The human ear is able to detect frequencies in a range from very low pitch (20 Hz) to very high pitch (20 kHz or 20,000 Hz). One kilohertz is one-thousand-times-per-second, abbreviated 1 kHz.

Try this sketch to make a sound
```
/*  StartResetIndicator
* Test the piezospeaker circuit */

void setup()
{
tone(4, 3000, 1000);            // Play tone for 1 second
delay(1000);                    // Delay to finish tone
}

void loop()
{
}
```
Remember that a sketch can use the Servo library's **writeMicroseconds** function to control the speed and direction of each servo. Then, it can use the **delay** function to keep the servos running for certain amounts of time before choosing new speeds and directions. Here's an example that will make the Robot roll forward for about three seconds, and then stop.

Moving the robot in a forward and making a beeping sound

Have you ever thought about what direction a car's wheels have to turn to propel it forward? The wheels turn opposite directions on opposite sides of the car. Likewise, to make the Robot go forward, its left wheel has to turn counterclockwise, but its right wheel has to turn clockwise.

Try this example to make the robot go forward for three seconds then stop.

```
//  ForwardThreeSeconds
// Make the Robot roll forward for three seconds, then stop.
#include <Servo.h>        // Include servo library

Servo servoLeft;          // Declare left and right servos
Servo servoRight;

void setup()
{

servoLeft.attach(11);     // Attach left signal to pin 11
servoRight.attach(10);    // Attach right signal to pin 10

tone(4, 3000, 1000);      // Play tone for 1 second delay(1000);

// Full speed forward
servoLeft.writeMicroseconds(1700);      // Left wheel counterclockwise
servoRight.writeMicroseconds(1300);     // Right wheel clockwise
 delay(3000);     // ...for 3 seconds

servoLeft.detach();       // Stop sending servo signals servoRight.detach();
servoRight.detach();   // Stop sending servo signals servoRight.detach();
}

void loop()
{
}
```

As with all motion sketches, the first action **setup** takes is making the piezospeaker beep. The tone function call transmits a signal to digital pin 4 that makes the piezospeaker play a 3 kHz tone that lasts for 1 second. Since the tone function works in the background while the code moves on, delay(1000) prevents the Robot from moving until the tone is done playing.

```
void setup()
{
tone(4, 3000, 1000);                    // Play tone for 1 second
delay(1000);                            // Delay to finish tone
```

Next, the **servoLeft** object instance gets attached to digital pin 9 and the **servoRight** instance gets attached to pin 8. This makes calls to **servoLeft.writeMicroseconds** affect the servo control signals sent on pin 9 sends. Likewise, calls to **servoRight.writeMicroseconds** will affect the signals sent on pin 8.

```
servoLeft.attach(11);                    // Attach left signal to pin 11
servoRight.attach(10);                   // Attach right signal to pin 10
```

Remember that we need the Robot's left and right wheels to turn in opposite directions to drive forward. The function call **servoLeft.writeMicroseconds(1700)** makes the left servo turn full speed counterclockwise, and the function call **servoRight.writeMicroseconds(1300)** makes the right wheel turn full speed clockwise. The result is forward motion. The **delay(3000)** function call keeps the servos running at that speed for three full seconds. After the delay, s**ervoLeft.detach** and s**ervoRight.detach** discontinue the servo signals, which bring the robot to a stop.

```
    // Full speed forward
    servoLeft.writeMicroseconds(1700);     // Left wheel counterclockwise
    servoRight.writeMicroseconds(1300); // Right wheel clockwise delay(3000);
                                           // ...for 3 seconds
     servoLeft.detach();                   // Stop sending servo signals
     servoRight.detach();
  }
```

After the **setup** function runs out of code, the sketch automatically advances to the **loop** function, which repeats itself indefinitely. In this case, we are leaving it empty because the sketch is done, so it repeats nothing, over and over again, indefinitely.

```
  void loop()
  {
  }
```

Let's say that your Robot gradually turns left. That means the right wheel is turning faster than the left. Since the left wheel is already going as fast as it possibly can, the right wheel needs to be slowed down to straighten out the robot's path. To slow it down, change the ***us*** parameter in **servoRight.writeMicroseconds(*us*)** to a value closer to 1500. First, try 1400. Is it still going too fast? Raise it 1410. Keep raising the parameter by 10 until the Robot no longer curves to the left. If any adjustment overshoots 'straight' and your Robot starts curving to the right instead, start decreasing the us parameter by smaller amounts. Keep refining that ***us*** parameter until your Robot goes straight forward. This is called an *iterative process*, meaning that it takes repeated tries and refinements to get to the right value.

If your Robot curved to the right instead of the left, it means you need to slow down the left wheel. You're starting with s**ervoLeft.writeMicroseconds(1700)** so the ***us*** parameter needs to decrease. Try

1600, then reduce by increments of 10 until it either goes straight or starts turning the other direction, and increase by 2 if you overshoot.

Modify ForwardTenSeconds so that it makes your Robot go straight forward.

Use masking tape or a sticker to experiment on each servo with the best ***us*** parameters for your

writeMicroseconds(*us*) function calls.

If your Robot already travels straight forward, try the modifications just discussed anyway, to see the effect. It should cause the Robot to travel in a curve instead of a straight line.

You might find that there's an entirely different situation when you program your Robot to roll backward.

Check out the same program, but this time using degrees instead of microseconds

```
// Make the Robot roll forward for three seconds, then stop.
#include <Servo.h>        // Include servo library

Servo servoLeft;          // Declare left and right servos
Servo servoRight;

void setup()
{

servoLeft.attach(11);     // Attach left signal to pin 11
servoRight.attach(10);    // Attach right signal to pin 10

tone(4, 3000, 1000);      // Play tone for 1 second delay(1000);

// Full speed forward
servoLeft.write (180);    // Left wheel counterclockwise
servoRight.write(0);      // Right wheel clockwise
 delay(3000);             // ...for 3 seconds

servoLeft.detach();       // Stop sending servo signals servoRight.detach();
servoRight.detach();       // Stop sending servo signals servoRight.detach();
}

void loop()
{
}
```

Tuning the Turns

The amount of time the Robot spends rotating in place determines how far it turns. So, to tune a turn, all you need to do is adjust the **delay** function's ***ms*** parameter to make it turn for a different amount of time.

Let's say that the Robot turns just a bit more than 90° (1/4 of a full circle). Try **delay(580)**, or maybe even **delay(560)**. If it doesn't turn far enough, make it run longer by increasing the **delay**function's ***ms*** parameter 20 ms at a time.

The smallest change that actually makes a difference is 20.

Servo control pulses are sent every 20 ms, so adjust your **delay** function call's ***ms*** parameter in multiples of 20.

If you find yourself with one value slightly overshooting 90° and the other slightly undershooting, choose the value that makes it turn a little too far, then slow down the servos slightly. In the case of rotating left, both **writeMicroseconds** ***us*** parameters should be changed from 1300 to something closer to 1500. Start with 1400 and then gradually increase the values to slow both servos. For rotating right, start by changing the ***us*** parameters from 1700 to 1600, and then experiment with reducing in increments of 10 from there.

Creating Functions (like in our keyboard control program)

Let's try putting the **forward**, **turnLeft**, **turnRight**, and **backward** navigation routines inside functions. Here's an example:

```
// MovementsWithSimpleFunctions
// Move forward, left, right, then backward for testing and tuning.

#include <Servo.h>                  // Include servo library
Servo servoLeft;                    // Declare left and right servos
Servo servoRight;
void setup()
{
tone(4, 3000, 1000);                // Play tone for 1 second
delay(1000);                        // Delay to finish tone
servoLeft.attach(11);               // Attach left signal to pin 11
servoRight.attach(10);              // Attach right signal to pin 10
forward(2000);                      // Go forward for 2 seconds
turnLeft(600);                      // Turn left for 0.6 seconds
```

```
turnRight(600);                       // Turn right for 0.6 seconds
backward(2000);                       // go backward for 2 seconds
disableServos();                      // Stay still indefinitely
}

void loop()
{
}

void forward(int time)                // Forward function
{
servoLeft.writeMicroseconds(1700);    // Left wheel counterclockwise
servoRight.writeMicroseconds(1300);   // Right wheel clockwise
delay(time);   // Maneuver for time ms
}

void turnLeft(int time)               // Left turn function
{
servoLeft.writeMicroseconds(1300);    // Left wheel clockwise
servoRight.writeMicroseconds(1300);   // Right wheel clockwise
delay(time);                          // Maneuver for time ms
}

void turnRight(int time)              //Right turn function
{
servoLeft.writeMicroseconds(1700);    //Left wheel counterclockwise
servoRight.writeMicroseconds(1700);   //Right wheel counterclockwise
delay(time);                          //Maneuver for time ms
}

void backward(int time)               // Backward function
{
servoLeft.writeMicroseconds(1300);
servoRight.writeMicroseconds(1700);
delay(time);
                                      // Maneuver for time ms
}

void disableServos()                  // Halt servo signals
{
servoLeft.detach();                   // Stop sending servo signals
servoRight.detach();
}
```

Want to keep performing that set of four maneuvers over and over again? Just move those four maneuvering function calls from the setup function into the loop function.

The .detach() is part of the Servo library that instead of sending the 1.5ms pulse to stop the servo you just say servo.XXX.detach();

Using Degrees to for Robot Movement

A sample program that shows the use of the Servo library

- Goes forward for 2 seconds
- Reverses for 2 seconds
- Turns right for 2 seconds
- Turns left for 2 seconds
- Stops for 2 seconds

```
#include <Servo.h>

Servo servoLeft;         // Define left servo
Servo servoRight;        // Define right servo

void setup() {
  servoLeft.attach(11);  // Set left servo to digital pin 11
  servoRight.attach(10);  // Set right servo to digital pin 10
}

void loop() {           // Loop through motion tests
  forward();            // Example: move forward
  delay(2000);          // Wait 2000 milliseconds (2 seconds)
  reverse();
  delay(2000);
  turnRight();
  delay(2000);
  turnLeft();
  delay(2000);
  stopRobot();
  delay(2000);
}

// Motion routines for forward, reverse, turns, and stop
void forward() {
  servoLeft.write(0);
  servoRight.write(180);
}

void reverse() {
  servoLeft.write(180);
```

```
  servoRight.write(0);
}

void turnRight() {
  servoLeft.write(180);
  servoRight.write(180);
}
void turnLeft() {
  servoLeft.write(0);
  servoRight.write(0);
}

void stopRobot() {
  servoLeft.write(90);
  servoRight.write(90);

}
```

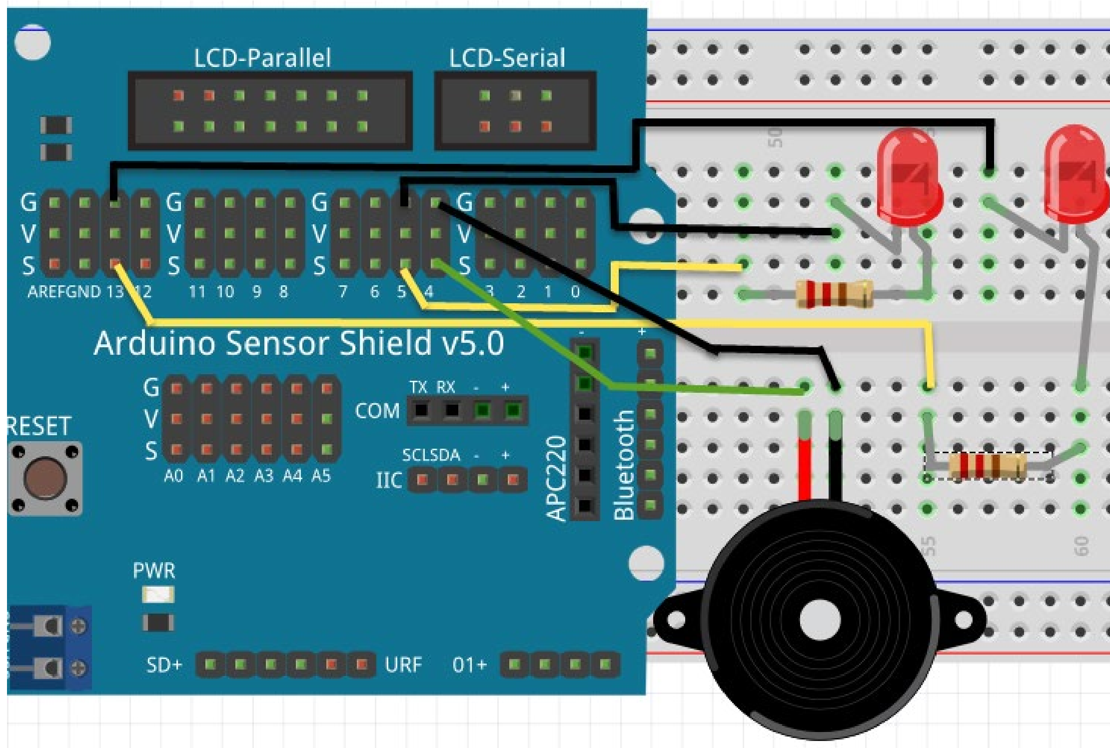
Servo movement is 0-180 degrees. In this case by using the library 0-180 means 13.ms to 1.7ms.
90 degrees is the center so is like sending a 1.5ms stay still signal to the serv.
You can change how long a robot moves for just by changing the delay value.

```
forward();          // Example: move forward
delay(2000);         // Wait 2000 milliseconds (2 seconds)
```

Here is a sample calibration program using this method

```
#include <Servo.h>

Servo servoLeft;        // Define left servo
Servo servoRight;        // Define right servo

void setup() {
 servoLeft.attach(11);  // Set left servo to digital pin 9
 servoRight.attach(10);  // Set right servo to digital pin 8
}

void loop()
{          // Loop through motion tests
 servoLeft.write(90);
 servoRight.write(90);

}
```

Now you are ready to complete the Experiment 1 challenge.
Advanced Sounds
Want more advanced sound? How about the Darth Vader entrance theme song or Mario Bros song? You can download the programs from the website.
If you look at the Darth Vader song you will see that you can also use some LEDs as special effect so they turn on and off as the song plays. LEDs are connected to pins 5 and 13. You probably already have an LED on pin 5. You can connect another to pin 13 the same way the one on pin 5 is connected.
Remember to turn robot OFF before rewiring anything on it.



For the Mario Bros song you will need to include a file called pitches.h and is also on the website under Experiment 1. Place this pitches.h file in the same folder as your MarioBros sketch.
If that does not work the follow the following.
This sketch needs the Library "pitches" which must be in the same folder as the sketch. In current versions of the Arduino IDE there is a "Down-Symbol" at the far right. Click that and select "New Tab" and name it "pitches.h". Paste the contents of the downloaded file into that window and save it along with the sketch.

Sample Solution to this experiment. You can also download from the website

```
#include <Servo.h>
// Robot waits for you to press the button to start the program. Robot rolls in a 3ft square making a sound at every turn
Servo leftServo;
Servo rightServo;
int button=2;
```

```
int speaker=4;
int count=0;
void setup()
{
leftServo.attach(11);
rightServo.attach(10);
pinMode(button, INPUT_PULLUP);

while(digitalRead(button)==1)
{
}
tone(speaker,1000,2000);
delay(2000);
do
{

forward();
delay(3560);
left_signal();
stop_move();
tone(speaker,1000,1000);
delay(500);
left_turn();
delay(850);
stop_move();
count++;
}
while(count<4);
right_turn();
delay(3000);
leftServo.detach();
rightServo.detach();
tone(speaker, 2500,3000);
}
void loop() {
}

void forward()
{
leftServo.writeMicroseconds(1300);
rightServo.writeMicroseconds(1325);
}
```

```
void right_turn()
{
leftServo.writeMicroseconds(1300);
rightServo.writeMicroseconds(1700);
}

void left_turn()
{
leftServo.writeMicroseconds(1700);
rightServo.writeMicroseconds(1300);
}

void reverse()
{
leftServo.writeMicroseconds(1700);
rightServo.writeMicroseconds(1700);
}

void stop_move()
{
leftServo.writeMicroseconds(1500);
rightServo.writeMicroseconds(1500);
}
void left_signal(){
for(int n=0; n<8; n++)
{
digitalWrite(5,HIGH);
delay(250);
digitalWrite(5,LOW);
delay(250);
}
}
void right_signal(){
for(int n=0; n<8; n++)
{
digitalWrite(6,HIGH);
delay(250);
digitalWrite(6,LOW);
delay(250);
}
}
```

End of Experiment

References

https://learn.rollingrobots.com/book/export/html/148
http://learn.parallax.com/tutorials/board-education-shield-arduino
https://www.arduino.cc/en/Tutorial/Foundations