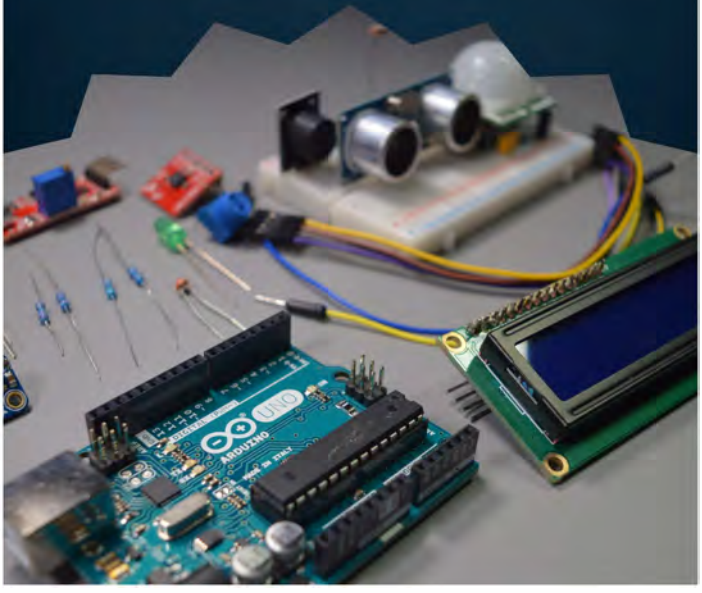
Tech Explorations

ARDUINO MOTORS & DISPLAYS

GET THE MOST OUT OF YOUR ARDUINO WITH ARTICLES FROM THE TECH EXPLORATIONS BLOG



Peter Dalmaris, PhD

Arduino Motors and Displays

Get the most out of your Arduino with articles from the Tech Explorations Blog

Welcome to this special collection of articles, meticulously curated from the Tech Explorations blog and guides. As a token of appreciation for joining our email list, we offer these documents for you to download at no cost. Our aim is to provide you with valuable insights and knowledge in a convenient format. You can read these PDFs on your device, or print.

Please note that these PDFs are derived from our blog posts and articles with limited editing. We prioritize updating content and ensuring all links are functional, striving to enhance quality continually. However, the editing level does not match the comprehensive standards applied to our Tech Explorations books and courses.

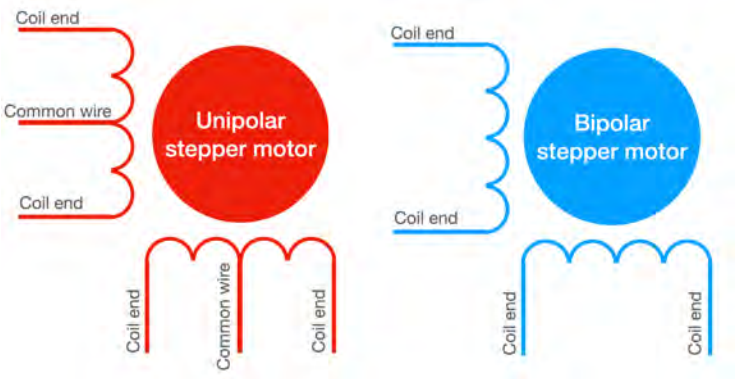
We regularly update these documents to include the latest content from our website, ensuring you have access to fresh and relevant information.

License statement for the PDF documents on this page

Motors guide series

Unipolar vs bipolar stepper motors

What is the difference between unipolar and bipolar stepper motors? It's in the coils.

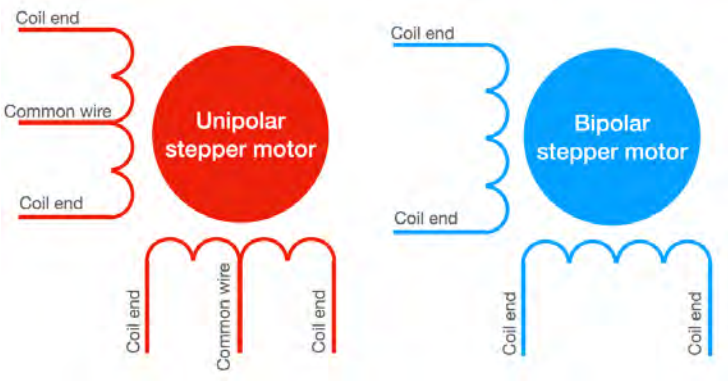


Bipolar stepper motors are generally able to produce more torque than unipolar stepper motors, and are more efficient.

However, they are more complicated to drive (="operate"). The main difference between the two types of stepper motors has to do with the way that the wire winding is constructed.

Stepper motor wire winding

Here's a simplified depiction of the wire winding for the two types of stepper motors.



The unipolar motor has a central common tap per phase. The bipolar motor does not.

In the schematic above, you can see a bipolar stepper motor and a unipolar stepper motor with two phases each.

A wire winding arrangement is refered to as a “phase”

The unipolar stepper motors, has one winding per phase, with a center tap. This allows the controlling circuit to operate the motor with current that flows always in the same direction. Therefore, there is no need to generate reverse current. Each time the phase is activated, only half of its coil is energized.

The bipolar stepper motor also has a single winding per phase. However, there is no center tap. This means that when the phase is activated, the entire coil is energized. The result is that the bipolar motor is able to produce much more torque compared to the unipolar motor. But, there is a cost: the controlling circuit must be able to generate current that can move both ways through the coil, i.e. “regular” current and “reverse” current. As a result, the controlling circuit for a bipolar stepper motor is more complicated then that of a unipolar stepper motor.

Bipolar motors have multiple (at least two) independent windings. A wire comes out of each of the winding’s ends, so

you get two wires per winding.

Unipolar motors may also have multiple (more than two) windings. However, in addition to the ends of each winding are connected to wires, the middle attaches to a third wire.

The absence of this third (common) wire means that bipolar motors are slightly easier to make.

Stepper motor drivers

When it comes to driving stepper motors, the simpler bipolar motor requires a more complex driver; this is because, to precisely control its motion, we need to be able to drive current in each winding in both directions.

On the other hand, in a unipolar motor, we can get away with a current that flows only in a single direction; this means that the driver electronics can be made simpler. The trade-off is that we use only half of each winding coil at a given time, and this translates to lower torque and efficiency.

However today, with easy access to motor drivers like H-bridges, it is easy to drive bipolar motors with alternating current. Unipolar motors advantage of not needing the reverse current is not a big deal anymore so it is possible to get all of their operational advantages with minimal cost.

Stepper motor drivers

To drive a stepper motor with your Arduino, you can consider these drivers (I have included on a few common examples):

- A4988, can control one bipolar motor with up to 2A of current per coil.
- DRV8825, can control one bipolar motor with up to 2.2A of current per coil.

- L298N, a classic driver, can control one bipolar motor with up to 2A of current per coil.
- TB6600, can control one large bipolar motor with up to 4.5A of current per coil.
- ULN2003, can control one small 5V unipolar stepper motor.

Learn more

If you would like to learn how to use bipolar and unipolar stepper motors (such as the NEMA17) with drivers such as the L298N, Easydriver or the ULN2003, consider enrolling to Arduino Step by Step Getting Serious.

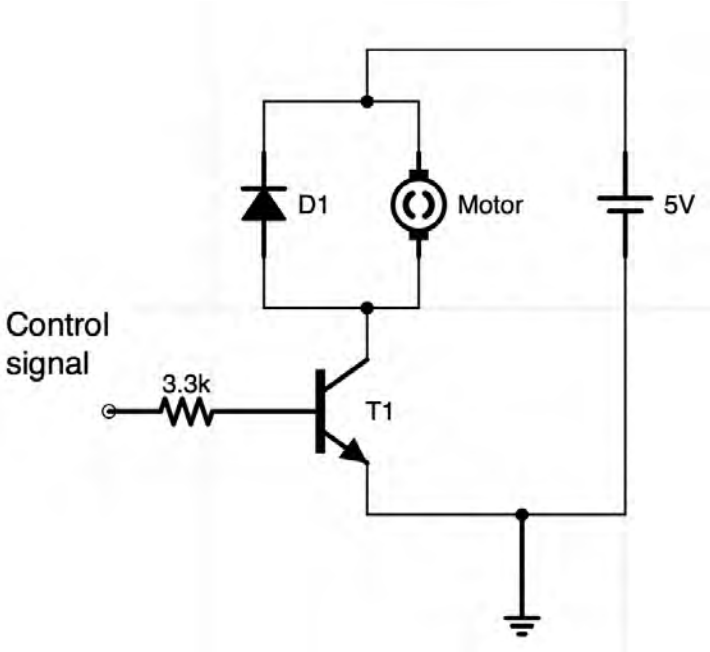
We cover stepper motors in a dedicated section (Section 18) that contains 18 lectures.

2. How to drive a DC motor without a motor driver module

Motors guide series

How to drive a DC motor without a motor driver module

DC motors draw currents that can be beyond the ability of the Arduino to supply. Transistors can be used as very simple at fast on/off switches and are an excellent option for designing simple motor controllers.



Any DC motor can be driven with PWM simple signals that can be generated by the Arduino Uno and virtually any other microcontroller. Just like you can control the intensity of an LED, you can use PWM to control the rotational speed of a DC motor.

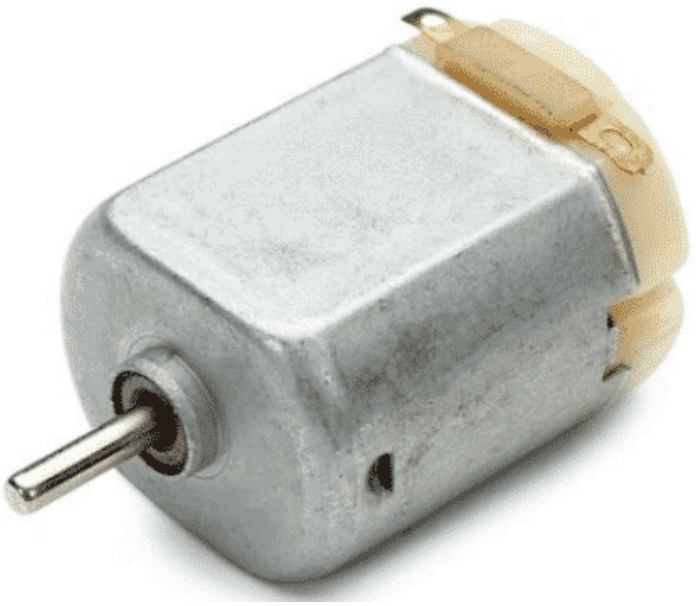
Whether it is a miniature 3V motor for toys, or a large 12V or 24V motor for your lawnmower, the principle of operation is the same.

DC motors and current requirements

The amount of current that a DC motor requires depends on the size of the motor. How large it is, the length of the wire in the motor coils, and the load that is attached to the motor. Because the Arduino Uno can only supply a few tens of milliamps (20mA, to be exact) of current through its digital pins, you should assume that it will not be able to safely power

even the smallest DC motor.

A motor draws the most current when its rotor is stationary. This is true when it starts, or when it is unable to move the attached load.



A 3V to 5V DC motor used in hobby applications.

Things get worse for larger motors. A 12V DC motor with nominal resistance in its coil of 15 will draw around 0.8A of current when it’s starting its rotation. That’s way too much, and it can damage destroy your Arduino.

For example, a tiny 3V DC motor with a 15 total resistance in its coil (like the one in the photo above) will draw 0.2A of current.

This is just within the Arduino’s I/O pin current limit. However, but if your motor’s resistance is slightly smaller, the current can easily increase more than the 0.4A which exceeds the

Arduino’s safe operating limits. If this happens, your Arduino will be damaged.

DC motor driver hardware

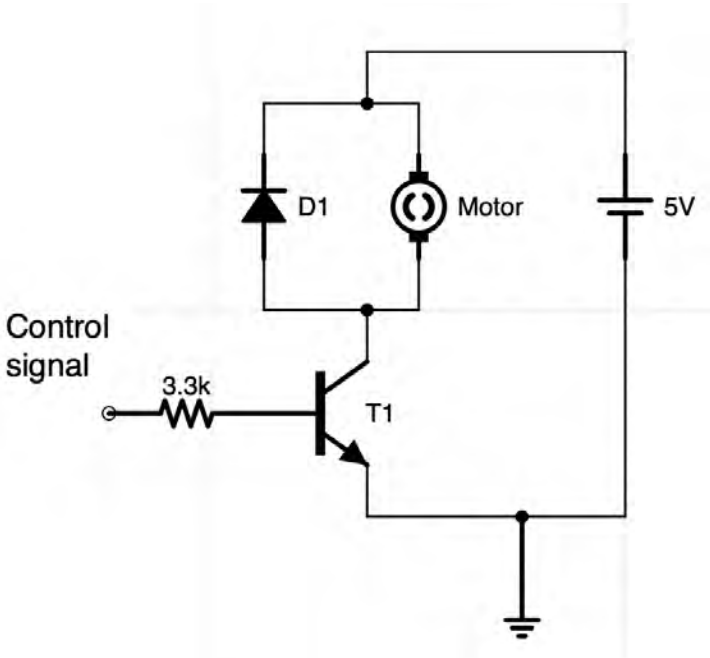
This is why we use specialized motor driver hardware to power and control the motor.

The L298N motor driver is easy to use and cheap, with a peak current capability of 3A. This amount of current is sufficient for more regular applications, like controlling a small fan or a robot.

A transistor as a simple DC motor controller

If you are looking for the simplest possible way to control a DC motor, then you will need a single transistor. You can choose a transistor that is appropriate for the current requirements of the motor that you want to control.

A DarlingtonTIP122transistor is a common device used in DC motor control applications.



A Dalrington transistor used to control a DC motor.

The Darlington TIP122 can provide 5A of continuous current through its collector and 15A of peak current, which can be drawn when a large motor starts. You can see it marked as “T1” in the schematic above.

Your Arduino can easily control the transistor, since it only needs 2.5V in its base to switch on (notice the label “Control signal” on the left of the current limiting resistor in the schematic above).

You can add a resistor (~3.3k) to protect the Arduino across the base of the transistor, and a diode (like the 1N4004) to block back-currents from the motor, and you have your motor driver, capable of regulating the rotational speed using PWM. If your motor is brushed, also add a small capacitor (~1uF) across the terminals of the motor to help with the electrical noise.

The power that drives the motor can come from an external power supply or a large batter. For example, you can drive a 12V motor from a mains (walled) 12V power supply that you recycle from an old appliance. In the example schematic, I am using a 5V power source for the motor.

In your Arduino sketch, you can use a simple PWM sketch like this:

[tcb-script src="https://cdnjs.cloudflare.com/ajax/libs/prism/9000.0.1/prism.min.js"][/tcb-script]int motor = 9; int speed = 0; int speedAmount = 5; void setup() { pinMode(motor, OUTPUT); } void loop() { analogWrite(motor, speed); speed = speed + speedAmount; if (speed <= 0 || speed >= 255) { speedAmount = -speedAmount; } delay(30); }

According to the sketch, the base of the transistor, via the resistor, is connected to Arduino's pin 9. In setup(), we configure the pin to be an output. In the loop(), we use analogWrite() to get the motor to gradually increase its rotational speed, and then to gradually decrease it.

Learn more

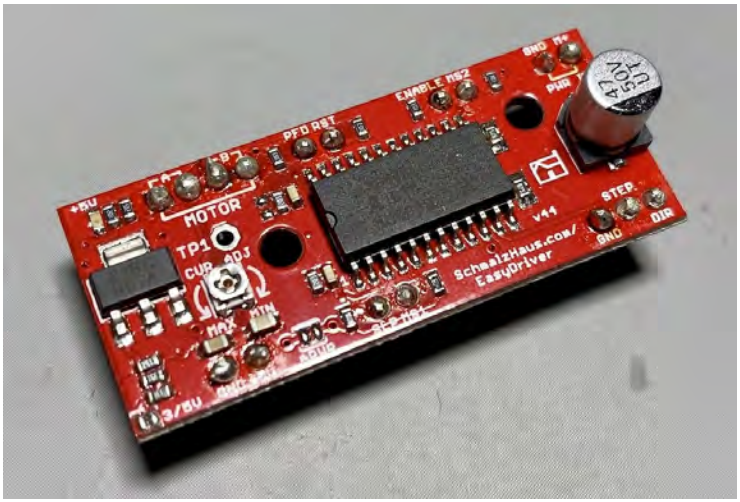
An excellent discussion of the use of discreet transistors to control a DC motor with schematics is here.

If you would like to learn how to use DC motors with the Arduino, consider enrolling to Arduino Step by Step Getting Serious.

We cover stepper motors in a dedicated section (Section 16) that contains 10 lectures.

What is “microstepping”?

Stepper motors move in steps. The width of a step is determined by the physical characteristics of the motor’s rotor. With a clever technique called “microstepping”, your stepper motor can double, quintuple or even octuple its precision. Read this article to also learn new words.

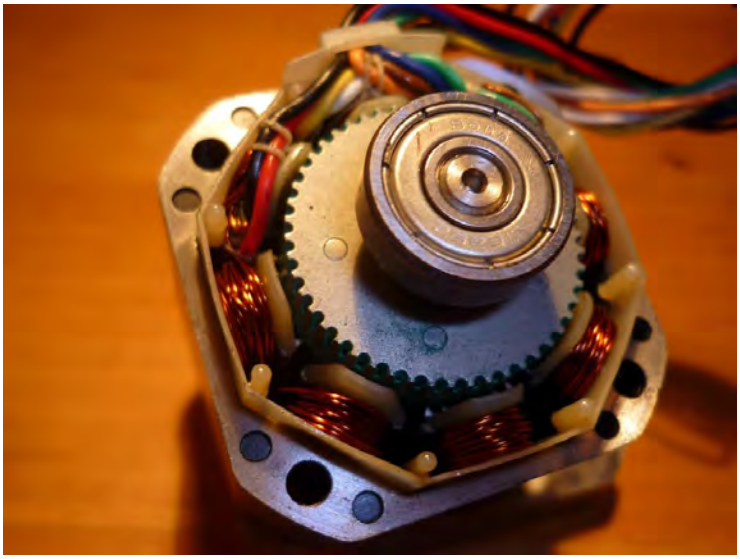


Stepper motors are capable of moving their rotor on specific positions along a 360° arc. The exact positions are defined by the positions of the toothed electromagnets positioned on the central rotor.

What is a stepper motor “step”?

In the photograph below, you can see the exposed central rotor of a stepper motor. Along the perimeter of the rotor, you can see the “teeth”, arranged in equal distances from their neighbors.

Because to the magnetic properties of these teeth, and the energizing pattern of the coils positioned on the inside of the motor’s casing, the rotor can be aligned precisely.



A stepper motor is designed to move at one step at a time by energizing its coils at the exact right times. When the motor moves like this (1 step at a time) it also moves within its torque specifications.

Standard stepper motors, like the one pictured above, have rotors with 200 teeth. There is a 1.8° separation between each tooth, which means that the rotor can move in steps of 1.8°, and do a full rotation by completing 200 steps.

What is a “microstep”?

If a precision of 1.8° is not enough, it is possible to increase it through the use of “magic” created in electronics hardware and microcontroller firmware. Essentially, we can have stepper motors that can operate at far higher precisions then their manufactured specifications.

Stepper motor drivers such as the EasyDriver can do microstepping. Microstepping is a technique that makes it possible to reduce the size of a step at the expense of torque. In other words, you get more accuracy and smoother motion with less turning force.

Microstepping is an elegant trick; it is a bit of a juggling act.

Microstepping with Easydriver

The EasyDriver energizes the two coils so finely that it manages to keep the motor in-between its regular resting positions. That is why when you do microstepping, both the motor and the EasyDriver can get very hot. They get hot because microstepping is hard work.

You can read about the EasyDriver microstepping modes in the documentation. Look for Q12. I am copying the relevant part here for convenience:

EasyDriver microstepping

The Easy Driver is able to operate in 1/8th, 1/4, half, and full step (2 phase) modes.

These four modes are selected by the logic levels on the MS1 and MS2 input pins.

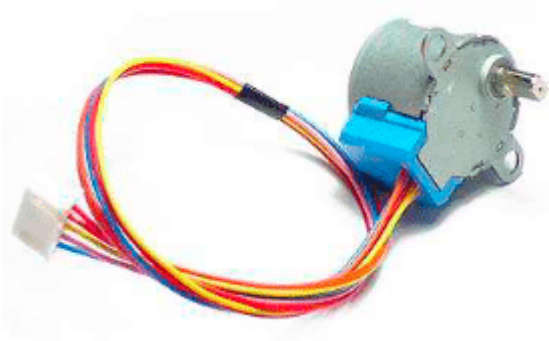
Normally, the pull-up resistors on the Easy Driver hold MS1 and MS2 high, which results in a default setting of 1/8th microstep mode.

You can pull either or both to ground to select the other 3 modes if you want. See the table below:

MS1	MS2	Resolution
low	low	Full Step (2 phase)
high	low	Half Step
low	high	Quarter Step
high	high	Eighth Step

For a motor like the common 28BYJ-48 (pictured below), one step is 5.625° (specs).

With the EasyDriver, depending on the state of the MS1 and MS2 pins, you can get it to move at 5.625° /2 = 2.8125° (half step), 5.625° /4 = 1.4062° (quarter step) and 5.625° /8 = 0.7031° (eight step).

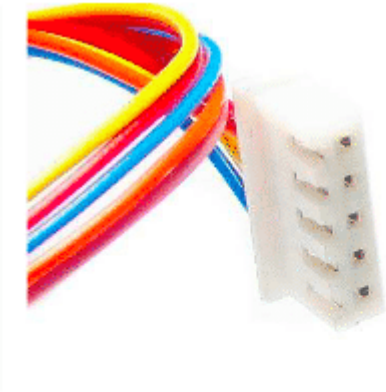


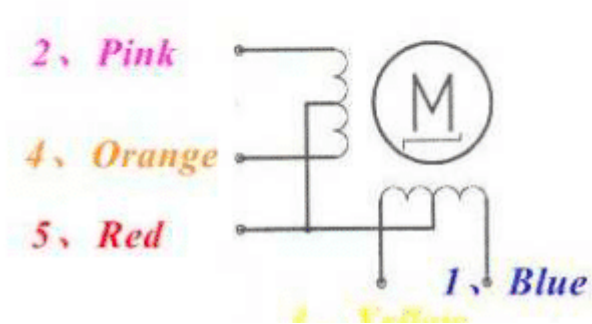
The 28BYJ-48 – 5V Stepper Motor

You can control MS1 and MS2 with wires connected to GND or 5V, dip switches, or just connected to the Arduino and control them in your sketch.

How to figure out the stepper motor coils

To find which wires correspond to a coil, look at the diagram in the datasheet.





It will be a bit tedious because of the middle connection but think about this: when you connect the pink and blue wires with your multimeter, you will get the largest resistance.

Only one pair can do this because these two pins wire contain the entire coil wire.

If you connect pink-red or orange-red, or red-yellow or red-blue, you will get the smallest resistance because they contain only part of the coil wire.

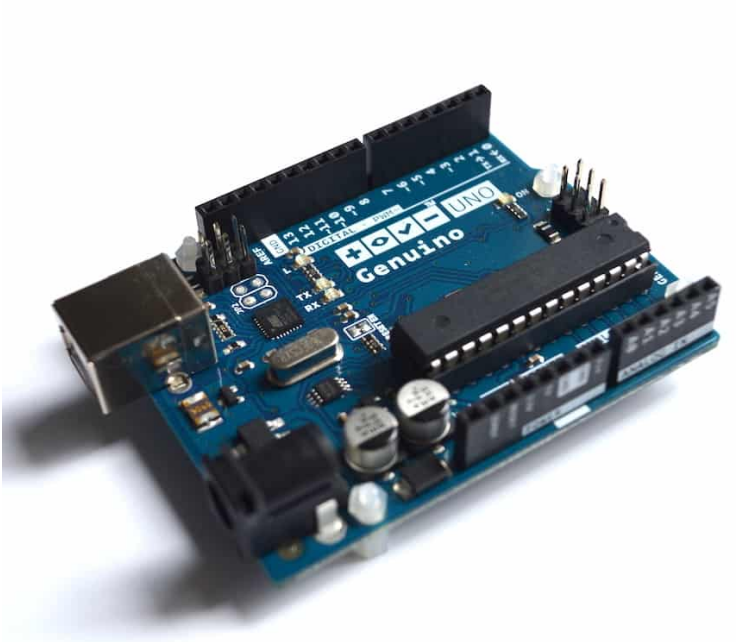
Now you can know the red (common wire). And when you connect pink-orange and yellow-blue, you will get the middle resistance.

With a bit of patience and a notebook, you can work out the coils.

Learn more

If you would like to learn how to use Easydriver with your Arduino, consider enrolling to Arduino Step by Step Getting Serious.

We have 5 dedicated lectures in Section 18.



Motors guide series

Direct current motor

Direct current motors represent the easiest way to add movement to your projects. A direct current or DC motor can be a very cost effective and flexible solution for all sorts of projects: robots, cars, boats, toy helicopters, home automation, and many more.

In this article, you will learn how to use a DC motor through a series of three projects.

To implement the projects, you will need two 5V DC motors and a L289N motor driver module.

Here's the project outcomes:

- Project 1: connect the motors to the Arduino via the motor driver module, and make them spin in a single direction.
- Project 2: use a potentiometer to control the direction of rotation and speed of the motors.
- Project 3: use an ultrasonic sensor to control the rotational speed the motors.

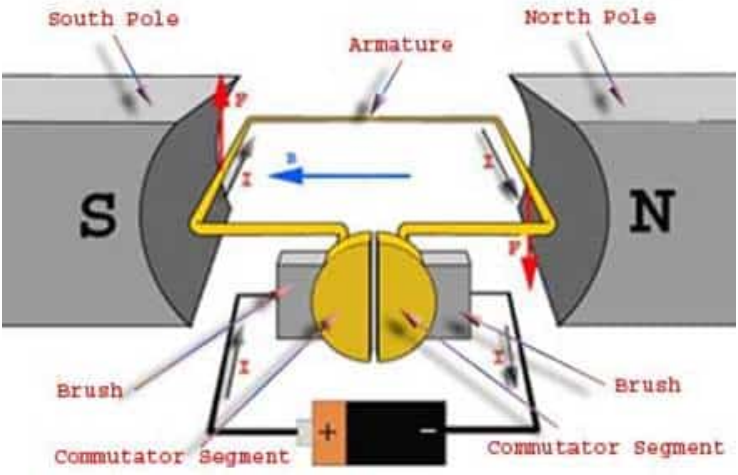
Before we get into the projects, I want do a quick introduction to motors, and specifically to discuss the kinds of motors you can use in your projects.

Types of motors

DC motors

The most common type of motor is the DC (Direct Current) motor. DC motors are low cost devices that work by connecting two wires to positive and negative voltage. Once you connect them to power, the rotor will turn. The direction of the rotor spin depends on the polarity of the connection. To reverse the direction of the spin, just switch the electrical connections.

All electrical motors work by taking advantage of the forces generated by a magnet (permanent or temporary electromagnet) and the electric field generated by electrical current as it travels through a coil. (Here is the source of the schematic below).



The DC motor can spin very quickly, which often is not what we want in our applications. For example, if you are building mechanism that opens the shutters of a window, you would like the shutter to open slowly to avoid damaging it. A DC motor on its own would try to get to full speed immediately. So, in most cases people attach a gear box to the motor. The gear box will convert high rotational spin from the shaft of the motor to lower rotational speed but higher torque so that large loads can be moved.

Stepper motors

Another common type of motor is the stepper motor.

A stepper motor operates by moving the motor rotor in steps, hence the name. I have written a dedicated article on stepper motors.

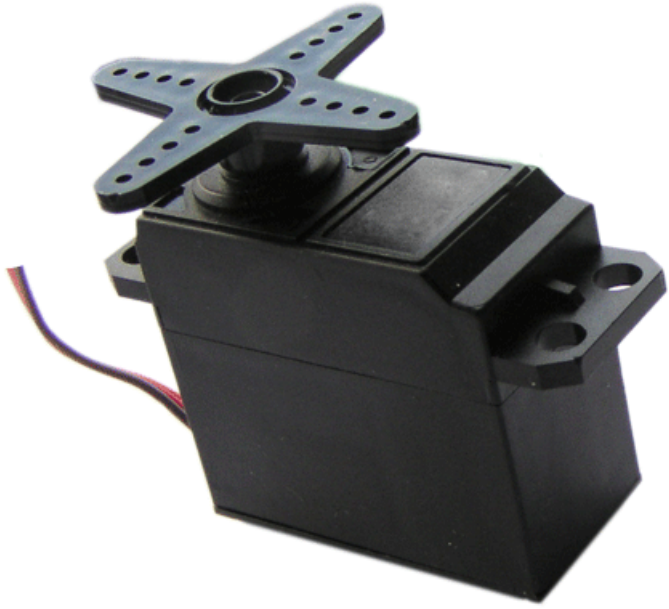
While the DC motor will move continuously as long as there is power, a stepper motor will move by one step when an appropriate signal is send via a wire, and then stop. Additional signals are needed for the rotor to move further. The size of each step (measured in degrees) depends on the mechanical characteristics of the motor as well as the characteristics of the electronic signals it receives on its coils.

For applications where you need fine control and small movements, a stepper motor is a good choice. For example, they are used in 3-D printers, where milli-meter accuracy is needed.

Stepper motors tend to be bulkier and more expensive then DC motors. They contain multiple coils and more complicated mechanical components, and that is reflected in their cost. The electronics needed to control stepper motors are also more complicated compared to DC motor controllers.

Servo motors

A common problem with stepper motors is that in certain situations (usually under high load), there is a mismatch between the signal that the motor controller is sending and the actual mechanical movement that the motor produces. In simple words, the rotor may “miss” steps and the controller will never know because usually there is no feedback mechanism to track and confirm movement.



A servo motor is designed to remedy this problem. A servo motor contains a DC motor, a gear box, and a position feedback mechanism. This mechanism can convert a simple DC motor into an a motor that can achieve precision movement with the ability to report its position to its controller circuit. Servo motors are used extensively in robotics, manufacturing automation systems, and high-end toys.

Now that you have a better understanding of motors, time to roll up your sleves and work on Project 1.

Learn more

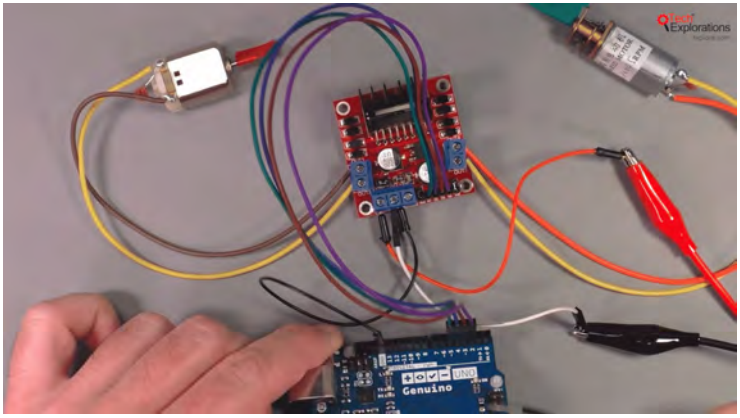
If you would like to learn how to use DC motors with your Arduino, consider enrolling to Arduino Step by Step Getting Serious.

We have a full section (Section 16) with 10 lectures dedicated to this topic.

Motors guide series

Project 1: Control two DC motors with your Arduino and the L298N controller

The L298N motor controller is a low cost and simple way to control two DC motors at the same time. It works well with the Arduino, and once you learn how to use it, you will be able to apply it on a wide range of DC motors.



In this article you will learn how to control two 5V DC motors with your Arduino.

To enable the Arduino to safely control the motors, you will use the L298N motor controller module.

I will be using two geared DC motors from a toy tank that I stole from my kids (they originally stole it from me).

The DC motor controller module

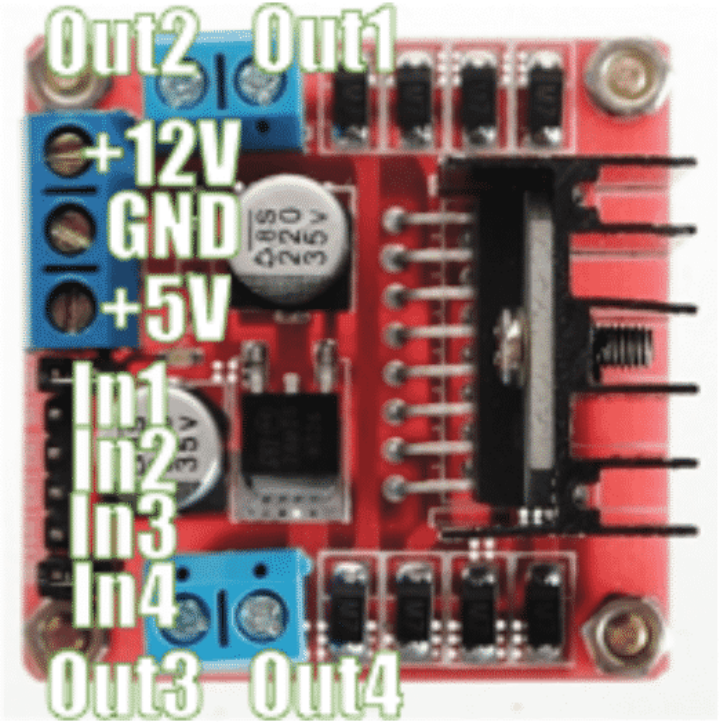
Even though you could connect a small DC motor directly to the Arduino, it is not a good idea.

The Arduino can only provide a very small amount of current to external devices. Only the tiniest of motors would be satisfied with that.

In virtually all real-life applications, you should provide external power to the motor.

To do this, I am going to use a popular motor driver module that contains the L298N motor driver "bridge". This integrated circuit (IC) can provide enough current to our motors, so we can use them in power-hungry applications. The L298N IC can't be used on its own. It requires several other components to operate properly, such as capacitors and diodes, a heatk sink etc.

Therefore, it is convenient to use a module that contains the L298N and all of the required components on a single board, like as the one in this image below.



Project parts list

You will need the following components for this circuit:

1. Two 5V DC motors.
2. An Arduino Uno.
3. A motor controller module with the L298N chip.
4. One AA-battery pack with 4 batteries.
5. A bunch of jumper wires.

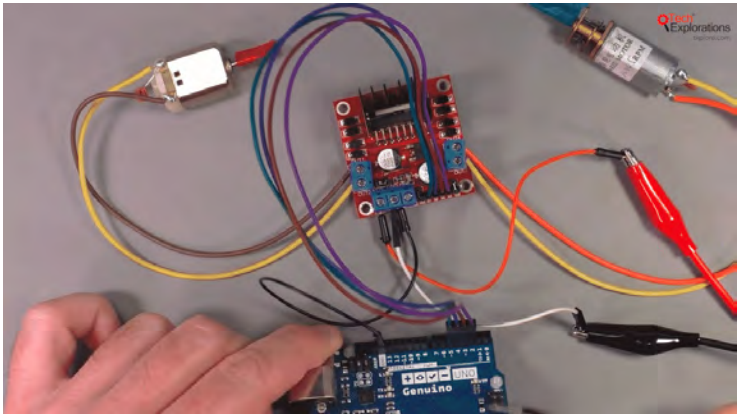
Circuit assembly process

Follow these instructions to make the connections and build your circuit. Refer to the annotated photograph of the L298N module above for the pin names.

Start by unplugging the Arduino from your computer so that it is not in operation.

1. Connect the **first motor** to motor controller module **Out1** and **Out2**. The order does not matter.
2. Connect the **first motor** to motor controller module **Out3** and **Out4**. The order does not matter.
3. Connect the **positive wire** from the battery pack to **pin +12V** on the module.
4. Connect the **negative wire** from the battery pack to **pin GND** on the module.
5. Connect the **GND pin** of the module to the **GND pin** of the Arduino.
6. Connect **Arduino pin 5** to **module pin In1**.
7. Connect **Arduino pin 4** to **module pin In2.**
8. Connect **Arduino pin 3** to **module pin In3.**
9. Connect **Arduino pin 2**to **module pin In4.**
10. **Inspect the wiring and ensure they are correct.**

And here’s what the assembled circuit looks like:



Let's look at the role of each pin in the motor controller module.

Out1 and **Out2** control the speed and direction of motor 1.

Out3 and **Out4** control motor 2.

As we are using DC motors, the higher the voltage differential between the pins in those pairs, the faster the motors will spin. If you want to reverse the motors, you must change the polarity of these pins. This is something that the L298N will do for you with the appropriate signal in the input pins.

In1 and **In3** control the direction of spin for motor 1 and motor 2 respectively.

In2 and **In4** control the speed for each motor.

Pin "+12V" receives power from an external power source. In this experiment, the safest option is to use an AA battery pack. If you use alkaline batteries, you will have 4 x 1.5V = 6V. This is sufficient to drive our two motors.

Finally, pin "GND" is for ground, and should be connected to one of Arduino's GND pins so that both the Arduino and the module share the same ground voltage level.

Sketch for project 1

Here's the sketch for Demo 1:

```
//Arduino PWM Speed Control

int E1 = 5; int M1 = 4;int E2 = 6; int M2 = 7;

void setup(){ pinMode(M1, OUTPUT); pinMode(M2, OUTPUT);}

void loop(){ int value; for(value = 0 ; value <= 255;
value+=1) { digitalWrite(M1,HIGH); digitalWrite(M2,HIGH);
analogWrite(E1, value); //PWM Speed Control analogWrite(E2,
value); //PWM Speed Control delay(30); } }
```

We start by setting the digital pins for the motor 1 and motor 2 control. We need two pins to control each motor. The "M" pins control the direction of rotation for the motor shafts, and the "E" pins control the speed.

In the **setup()** function we set the mode for pins M1 and M2 to be output (so we can control the direction of the spin).

In the **loop()** function, we have a "for" loop that cycles from 0 to 255, so it covers all the possible PWM values that can be written to pins E1 and E2 (Arduino pins 5 and 6 respectively).

Every time that the block in this loop runs, we first set the direction of spin for both motors by writing a HIGH value to pins M1 and M2 (Arduino pins 4 and 7 respectively), and then we set the spin by using the PWM function **analogWrite**. The value we write to these pins is controlled by the "for" loop.

Once you connect this circuit and run the loop, you will have your motors starting from zero speed, speeding up gradually until they reach their maximum rotational speed, before they suddenly stop to a halt and then repeating the same processes again, until power is switched off.

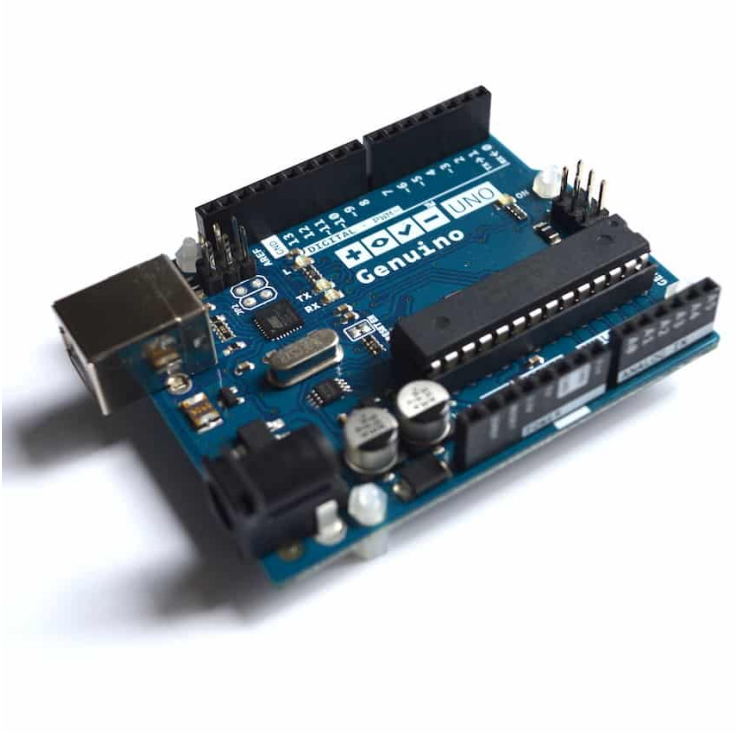
To change the direction of rotation, just change the value you

write to either M1 or M2 pins to LOW, and upload the sketch. You will see the corresponding motor shaft moving the opposite way compared to the original sketch.

Learn more

If you would like to learn how to use DC motors with your Arduino, consider enrolling to Arduino Step by Step Getting Serious.

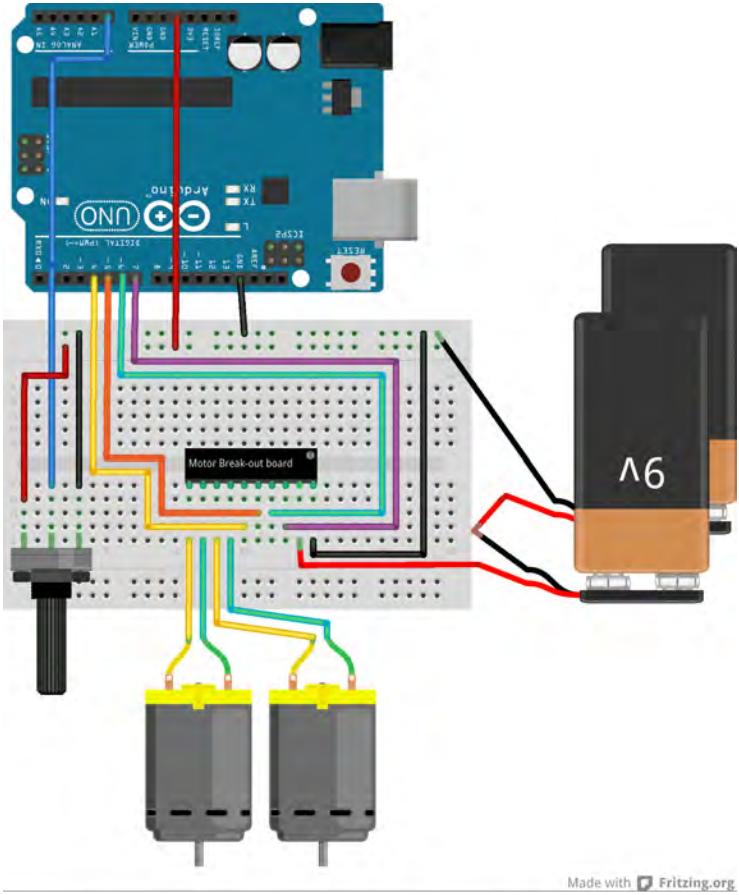
We have a full section (Section 16) with 10 lectures dedicated to this topic.



Motors guide series

Project 2: DC motor speed and direction control with a potentiometer

In this project, you will learn how to control the speed and direction of spin of the DC motor's rotor. You will use a potentiometer to provide input to the Arduino, and the map() and analogWrite() functions in your sketch to make this work.



In Project 1 you learned how to control two DC motors with an Arduino Uno and an L298N motor driver. In this experiment you will take the next step. You will learn how to control the speed and the direction of the two motors. In the role of the user interfaceyou will use a 10 k potentiometer.

You will use the potentiometer in two ways:

- First, you will use the potentiometer to control the speed of the motors, but not the direction.

- Second, you will use the potentiometer to control both the speed and the direction o the motors.

The project 2 components

You will only need to add a 10kOhm potentiometer to the circuit of Project 1. To save you from jumping to the Project 1 page, I'm copying the full list of components here.

The new item is #6:

1. Two 5V DC motors.
2. An Arduino Uno.
3. A motor controller module with the L298N chip.
4. One AA-battery pack with 4 batteries.
5. A bunch of jumper wires.
6. A 10 k potentiometer.

The project 2 circuit

Let's do the wiring. You can add the potentiometer to the circuit from Project 1. I am copying here the complete instructions so that you don't have to jump back and forth between the two articles.

The new items in the list are #10 and #11.

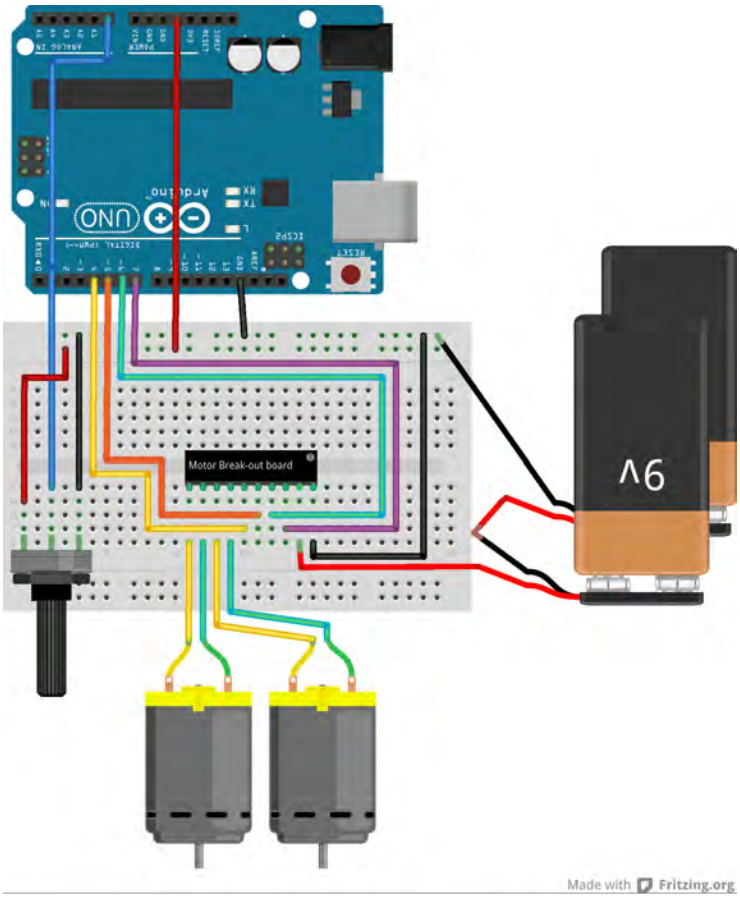
Start by unplugging the Arduino from your computer so that it is not in operation.

1. Connect the **first motor** to motor controller module **Out1** and **Out2**. The order does not

matter.

2. Connect the **first motor** to motor controller module **Out3** and **Out4**. The order does not matter.
3. Connect the **positive wire** from the battery pack to **pin +12V** on the module.
4. Connect the **negative wire** from the battery pack to **pin GND** on the module.
5. Connect the **GND pin** of the module to the **GND pin** of the Arduino.
6. Connect **Arduino pin 5** to **module pin In1**.
7. Connect **Arduino pin 4** to **module pin In2.**
8. Connect **Arduino pin 3** to **module pin In3.**
9. Connect **Arduino pin 2** to **module pin In4.**
10. Connect the **middle pin** of the potentiometer to **Arduino pin A0**.
11. Connect the **other two pins** of the potentiometer to **Arduino pins 5V and GND**.
12. **Inspect the wiring and ensure they are correct.**

Here’s the demo 2 circuit:



The only difference to the Project 1 circuit is the addition of the rotary potentiometer. It is connected to the 5V and GND columns on the breadboard, and to analog pin 0 on the Arduino. The 9V batteries depict the motor battery power supply. The black module in the middle of the breadboard depicts the L298N motor driver module with the connections as described earlier.

Sketch for Project 2, version 1: Control speed with a potentiometer

Here's the first sketch for Project 2. This sketch allows the control of only the rotational speed of the two motors:

```
int E1 = 5; int M1 = 4;int E2 = 6; int M2 = 7;

void setup(){ Serial.begin (9600); pinMode(M1, OUTPUT);
pinMode(M2, OUTPUT);}

void loop(){ int potentiometerVal = analogRead(A0);
Serial.println(potentiometerVal);
move_motors(potentiometerVal);}

void move_motors(int potValue){ int mappedVal =
map(potValue,0,1023,0,255); digitalWrite(M1,HIGH);
digitalWrite(M2, HIGH); analogWrite(E1, mappedVal); //PWM
Speed Control analogWrite(E2, mappedVal); //PWM Speed
Control delay(30); }
```

I have highlighted the changes and additions from the sketch in Project 1 in red.

In the **loop()** function, we take a reading from analog pin 0 where the middle pin of the potentiometer is connected. We then call the function **move_motors** and pass the potentiometer reading to it.

Inside the **move_motors**() function, we map the potentiometer reading (which comes in the range of 0 to 1023) to a value between 0 and 255 (which is the range of valid values for the PWM pins), set the direction of rotation for the two motors using the **digitalWrite** functions, and set the speed using the PWM function **analogWrite()** to be **mappedVal** (which in turn is relevant to the value we set by turning the potentiometer).

Upload this sketch and turn the knob of the potentiometer

back and forth. See how the speed of the motors change accordingly?

Sketch for Project 2, version 2: control speed and direction with a potentiometer

There's one more thing to do, and that is to also be able to control the direction of rotation of the motors, not just the speed. What I'd like to do is to be able to accelerate the motor towards one direction when I turn the knob of the potentiometer towards the left, and to the opposite direction when I turn the knob towards the right.

Schematically, this is what I would like to achieve: The red circle represents the potentiometer knob, and the black shows movement of the knob towards the left or the right.

Motors move anti-clockwise

Neutral – motors don't move

Motors move clockwise

We are only going to make an adjustment to the sketch to make this happen, no need to modify the circuit at all.

Here is the modified sketch:

```
//Arduino PWM Speed Control

int E1 = 5;

int M1 = 4;

int E2 = 6;

int M2 = 7;

void setup()

{

Serial.begin (9600);

pinMode(M1, OUTPUT);

pinMode(M2, OUTPUT);

}

void loop()

{

int potentiometerVal = analogRead(A0);

Serial.println(potentiometerVal);

move_motors(potentiometerVal);

}

void move_motors(int potValue)

{

if (potValue< 512)
```

```
{
int mappedVal = map(potValue,0,512,0,255);
//Going forward
digitalWrite(M1,HIGH);
digitalWrite(M2, HIGH);
analogWrite(E1, mappedVal); //PWM Speed Control
analogWrite(E2, mappedVal); //PWM Speed Control
delay(30);
} else
{
//Going backward
int mappedVal = map(potValue-512,0,512,0,255);
digitalWrite(M1,LOW);
digitalWrite(M2, LOW);
analogWrite(E1, mappedVal); //PWM Speed Control
analogWrite(E2, mappedVal); //PWM Speed Control
delay(30);
}
}
```

I have highlighted in red the part of the sketch that has changed from the script in demo 2.

In the **move_motors(int pot_value)** function, **potValue**, the value read from analog pin 0 where the potentiometer is connected, is tested using the **if** function. If **potValue** is less than 512, then the first block is executed, if not then the second block is executed.

In block 1 and 2, the motor control functionality from demo 2 is repeated with the difference that now we now need to recognize that the potentiometer's range of output values is divided to two parts. The first part, from 0 to 512, is used for moving the motors towards one direction, while the second one, 513 to 1023, moves the motors towards the other direction.

The only other difference between these two blocks is that in Block 1, a HIGH is written to M1 and M2, while in Block 2 we write a LOW, therefore spinning the motors towards the opposite direction.

Go ahead, try it.

If you are having any difficulty understanding what is going on in Blocks 1 and 2, experiment with the sketch. Try changing the map function values to something different.

For example, in Block 2, instead of

map(potValue-512,0,512,0,255)

try

map(potValue,0,512,0,255)

and upload the sketch.

What happens to the rotational speed of the motors?

Perhaps you could try this too:

map(potValue,512,1023,0,255).

Upload and compare to the previous two versions.

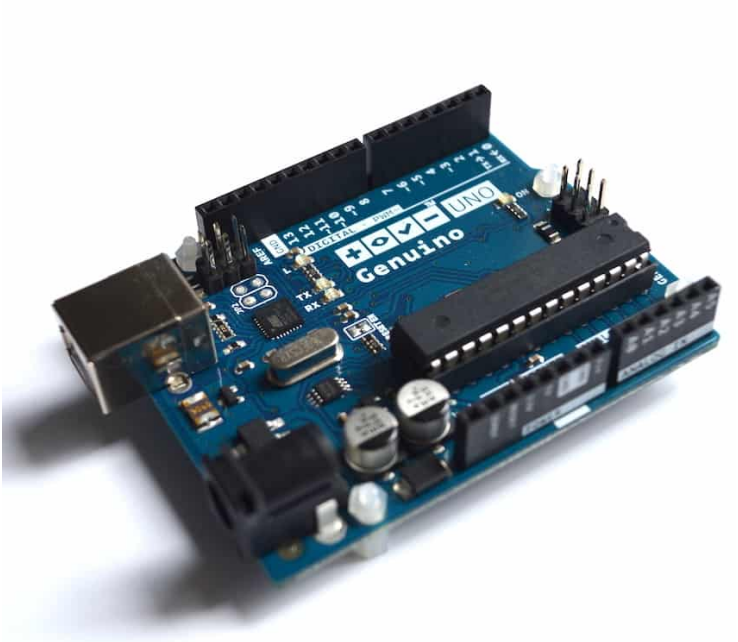
Can you notice any difference in behavior?

Can you explain any differences or similarities?

Learn more

If you would like to learn how to use DC motors with your Arduino, consider enrolling to Arduino Step by Step Getting Serious.

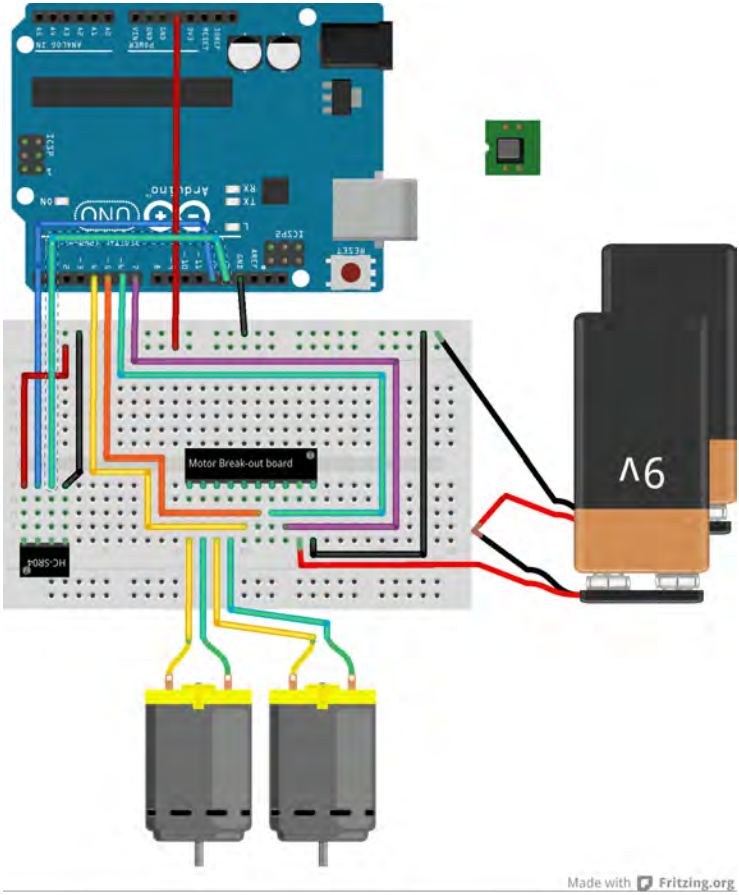
We have a full section (Section 16) with 10 lectures dedicated to this topic.



Motors guide series

Project 3: DC motor speed control with a distance sensor

Let’s try a variation of the Project 2 experiment: control the speed of the DC motor with an ultrasonic distance sensor. Of course, we’ll use an Arduino and the L298N motor driver.



After completing Project 1 and Project 2, you have learned how to control a motor with your Arduino and the L298N driver module. You can replace the potentiometer of Project 2 with a joystick, without many modifications to the sketch. You can see how, from a humble beginning, you can start branching out and experimenting with different types of hardware, to gradually reach more interesting configurations.

In this article, I will describe one last experiment that involves a DC motor. We will connect an ultrasonic distance sensor in the place of the potentiometer, and use that to control the rotational speed of the motors. The idea is this: The distance

sensor will tell the Arduino how close it is to an obstacle. If it gets too close, it will start slowing down the motors, effectively slowing down our prototype vehicle. Otherwise, it will forge ahead full speed.

The project 3 components

To complete this project, you will built on the circuit from Project 2. If you have completed Project 2, you will only need one additional component: the HCSR04 ultrasonic distance sensor. This sensor will replace the potentiometer.

The HC-SR04 ultrasonic distance sensor.

To save you from jumping to the Project 2 page, I'm copying the full list of components here.

The new item is #6:

1. Two 5V DC motors.
2. An Arduino Uno.
3. A motor controller module with the L298N chip.

4. One AA-battery pack with 4 batteries.
5. A bunch of jumper wires.
6. A HCSR04 ultrasonic distance sensor.

The project 3 circuit

Time to do the wiring. You can remove the potentiometer to the circuit from Project 2 and replace it with the ultrasonic distance sensor. I am copying here the complete instructions so that you don't have to jump back and forth between the two articles.

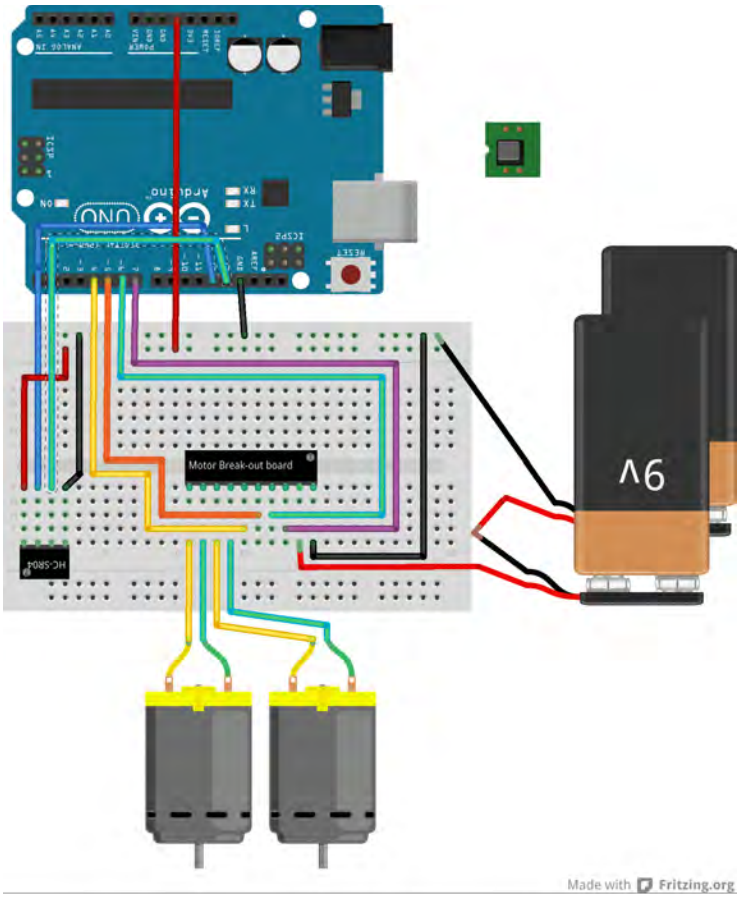
The new connections that relate to the ultrasonic distance sensor are #10, #11, #12 and #13.

Start by unplugging the Arduino from your computer so that it is not in operation.

1. Connect the **first motor** to motor controller module **Out1** and **Out2**. The order does not matter.
2. Connect the **first motor** to motor controller module **Out3** and **Out4**. The order does not matter.
3. Connect the **positive wire** from the battery pack to **pin +12V** on the module.
4. Connect the **negative wire** from the battery pack to **pin GND** on the module.
5. Connect the **GND pin** of the module to the **GND pin** of the Arduino.
6. Connect **Arduino pin 5** to **module pin In1**.
7. Connect **Arduino pin 4** to **module pin In2.**
8. Connect **Arduino pin 3** to **module pin In3.**

9. Connect **Arduino pin 2** to **module pin In4.**
10. Connect the **sensor pin Vcc** to **Arduino pin 5V**.
11. Connect the **sensor pin GND** to **Arduino pin GND**.
12. Connect the **sensor pin Trig** to **Arduino pin 13**.
13. Connect the **sensor pin Echo** to **Arduino pin 12**.
14. **Inspect the wiring and ensure they are correct.**

Here’s the schematic:



As with Project 2, the 9V batteries depict the motor battery power supply. The black module in the middle of the breadboard depicts the L298N motor driver module with the connections as described earlier.

The ultrasonic distance sensor uses 4 wires, two for power (5V and GND), one for the Trigger pin, and one for Echo. Connect those wires as per the instructions

Sketch for Project 3

Here’s the sketch for this project.

```
//Arduino PWM Speed Control

int E1 = 5;

int M1 = 4;

int E2 = 6;

int M2 = 7;


#define trigPin 13

#define echoPin 12


void setup()

{

Serial.begin (9600);

pinMode(M1, OUTPUT);

pinMode(M2, OUTPUT);

pinMode(trigPin, OUTPUT);

pinMode(echoPin, INPUT);

}

void loop()

{
```

```
long duration, distance;
digitalWrite(trigPin, LOW);
delayMicroseconds(2);
digitalWrite(trigPin, HIGH);
delayMicroseconds(10);
digitalWrite(trigPin, LOW);
duration = pulseIn(echoPin, HIGH);
distance = (duration/2) / 29.1;

move_motors(distance);
}
void move_motors(int distance)
{
if (distance >= 50 ){
Serial.print(distance);
Serial.println(" cm – 255);
digitalWrite(M1,HIGH);
digitalWrite(M2, HIGH);
analogWrite(E1, 0); //PWM Speed Control
analogWrite(E2, 0);
delay(30);
```

```
}

else {

int mappedVal = map(distance,0,50,0,255);

Serial.print(distance);

Serial.print(" cm – ");

Serial.println(mappedVal);

digitalWrite(M1,HIGH);

digitalWrite(M2, HIGH);

analogWrite(E1, 255-mappedVal); //PWM Speed Control

analogWrite(E2, 255-mappedVal); //PWM Speed Control

delay(30);

}

}
```

Much of this code, at least the part in the **loop()** function, has been taken from the sketch in Lecture 11. It generates the ping pulse that is emitted by the sensor, and then calculates the distance of an object from the time it takes for the echo to return.

Once the distance is calculated, the **move_motors** function is called, and the distance is passed as an integer parameter. Just like in Demo 2 Version 2, motors are moved in one of two ways: if distance is 50cm or less, the motor speed is proportional to the distance. The smaller the distance is, the smaller the speed. Otherwise, if the distance is over 50cm, then motors will move at their maximum speed.

If these motors were part of a fully assembled vehicle, the effect of this sketch would be that the vehicle would be capable of avoiding hitting an object by slowing down on approach, until it came to a full stop.

Upload the sketch and play with it to get a sense of it in “eal life”. Try to imagine how it would behave in a real situation. Can you foresee any limitations? Can you imagine any improvements?

Improvements

For example, in the current version of the sketch, the vehicle would come to a complete stop only when the distance to a wall was 0cm. Is this sufficient? Perhaps it should come to complete stop a bit earlier, maybe at 5cm? Or perhaps, to be even more cautious, the vehicle should backup slightly away from the obstacle in order to provide room for a turns?

Can you make changes to the Demo 3 sketch and implement these scenarios (or whatever other improvement you can think of)?

Learn more

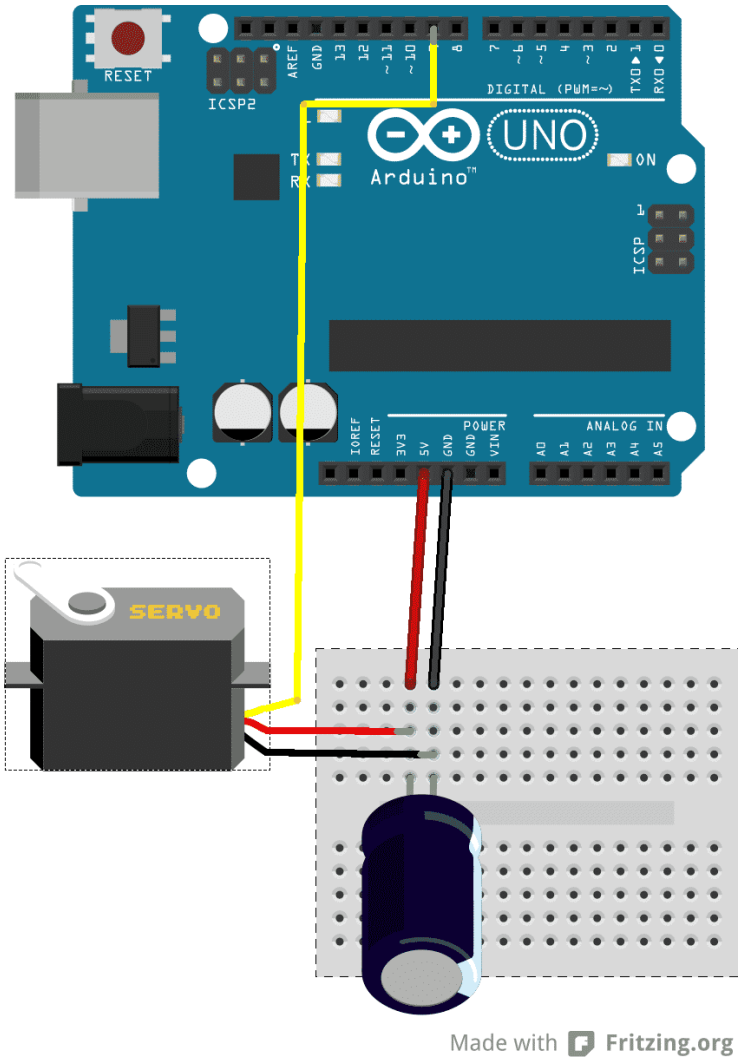
If you would like to learn how to use DC motors with your Arduino, consider enrolling to Arduino Step by Step Getting Serious.

We have a full section (Section 16) with 10 lectures dedicated to this topic.

Motors guide series

Project 1: Control a servo motor with a potentiometer

Servo motors are used in applications where precision movement is required, such as in robotics. It is very easy to control one or more servo motors with the Arduino. Learn how with this article.

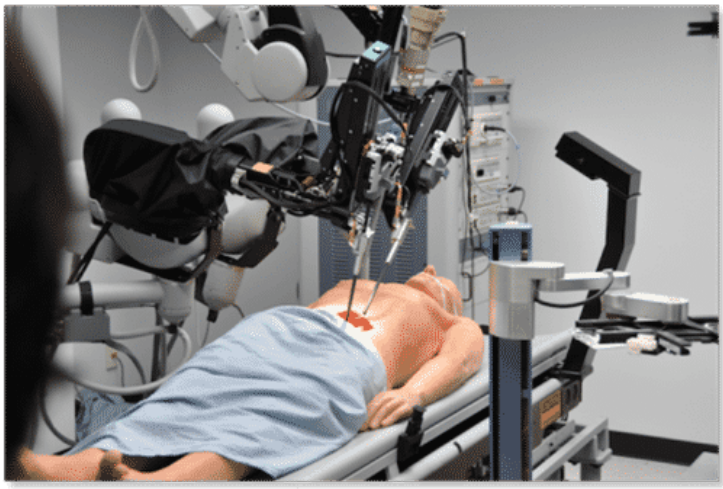


In other articles in the series (see here and here), you learned about the DC motor. The DC motor is a versatile, low-priced solution for providing motion to your projects. A “weakness” (or characteristic) of the plain vanilla DC motor is that when you apply voltage to its coil, it will spin as fast as its mechanics and load allows. Without additional electronics, we have no way of knowing or controlling the rotor position and speed.

This is not a problem for applications were continuous movement is needed, like for propelling a vehicle. However, it is a problem in application where precision is needed.

Example application of precision movement

Imagine a robot that can perform surgery on humans. Some of you may work on something like this in the not so distant future. In such application, precision and control feedback is of paramount importance. You don't just want to tell the motor to rotate its rotor by 90 degrees; you also want confirmation that it did.



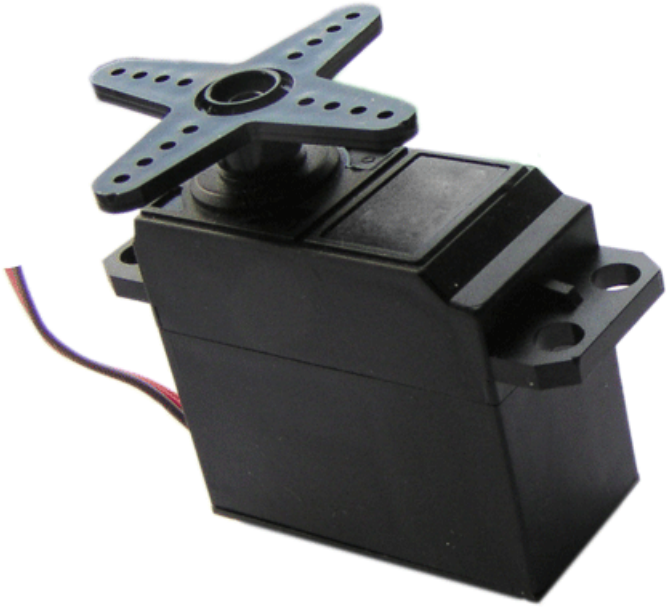
The servo motor

In a previous article in the series, I made a passing mention of the servo motor. The servo motor is an enhanced DC motor that includes circuitry for fine movement control and feedback.

In this and the next article, you will learn how to use a servo

motor with your Arduino.

To make programming easy, you will use the servo motor library that comes standard with the Arduino IDE, as well as a third party library that adds a bunch of very useful capabilities.



Let’s go straight to the assembly and sketches.

A simple servo motor project

In this project, you will use the servo motor library that comes with the Arduino IDE to start playing with your mini servo motor. Start by assembling the circuit. Then you will work on the sketch.

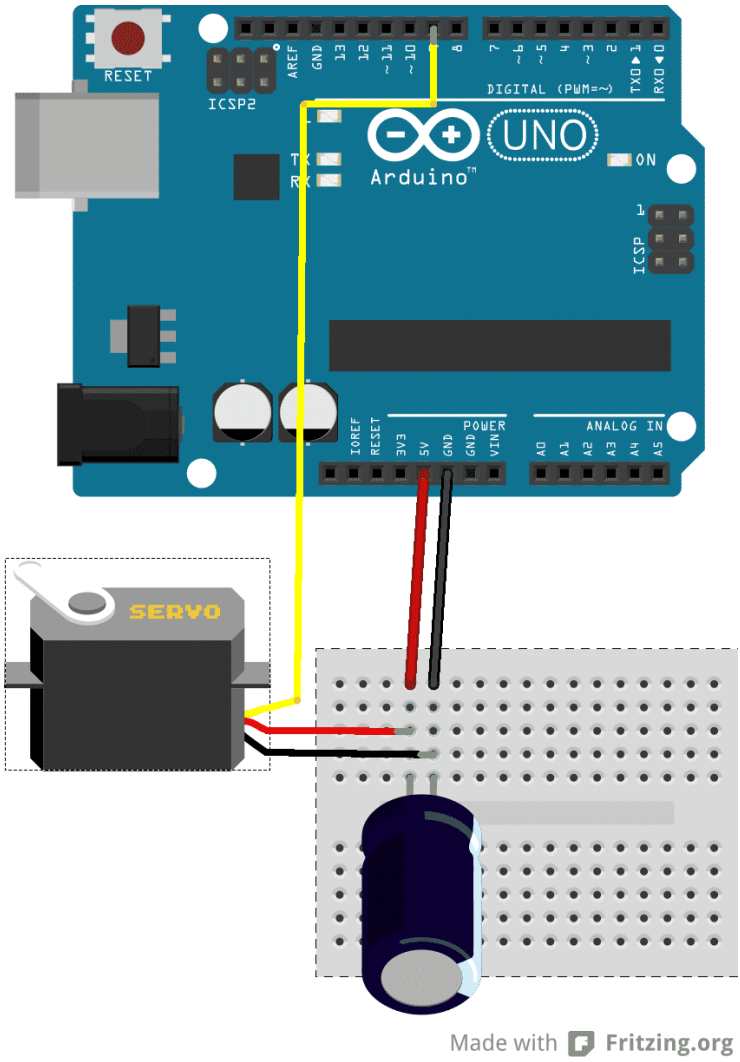
Project parts list

You will need the following components for this circuit:

- An Arduino Uno.
- A mini servo motor.
- A 300F capacitor.

Servo motor circuit

This is what you are going to assemble:



There are a couple of things to notice here.

First, there is no motor break-out board like the one we used in the DC motor lecture. There, I wrote that because the Arduino cannot provide enough current for the motor, it is best to always use an external power source to power our motors. This is why you used the L298N motor controller module.

However, the mini servo motor that you are using in this circuit, is small enough to not stress the Arduino too much. Therefore, you can safely plug it straight into the Arduino in order to keep things simple.

Notice the big round tube at the lower part of the breadboard? That is a capacitor. Even though my servo motor is small, occasionally it may draw more power than what the Arduino can provide. This spike in power demands happens when the motor begins to move.

To assist the battery or other power supply to provide this additional power, you can use a small capacitor. Connect this capacitor between the GND and +5V pins of the motor. A capacitor works as store of energy, a bit like a battery that can charge and discharge very quickly.

When the servo motor starts, the capacitor will assist the main power supply by discharging, ensuring that the motor has all the power it needs.

A capacitor that is around 300F or more is a good choice for a mini servo motor.

The motor itself has three wires coming out of it.

Two are for power (+5V and GND), and one for signal.

Usually, the red wire is for +5V, and black or brown for GND.

The yellow wire is for the signal.

Plug the **signal wire to digital pin 9**, the **red to +5V** and the **black to GND**.

Sketch

You could try to control the servo motor through the Arduino digitalWrite() functions but that would require us figuring out

the right values to write, and the timing for writing those values.

That's too much work.

We are lucky, though, because with the Arduino IDE we get the Servo library, which contains functions that allow us to easily work with servo motors. First, we'll write the sketch using the Servo library.

Later, I'll show you an alternative library that provides additional functionality.

Here's the sketch:

```
#include <Servo.h>


Servo myservo; // create servo object to control a servo

// a maximum of eight servo objects can be created

int pos = 0; // variable to store the servo position


void setup()

{

// attaches the servo on pin 9 to the servo object

myservo.attach(9);

}


void loop()

{
```

```
// goes from 0 degrees to 180 degrees

for(pos = 0; pos < 180; pos += 1)

{ // in steps of 1 degree

// tell servo to go to position in variable ‘pos’

myservo.write(pos);

// waits 15ms for the servo to reach the position

delay(15);

}

// goes from 180 degrees to 0 degrees

for(pos = 180; pos>=1; pos-=1)

{

// tell servo to go to position in variable ‘pos’

myservo.write(pos);

// waits 15ms for the servo to reach the position

delay(15);

}}
```

This is one of the demo sketches that also come with the Arduino IDE. You can find it by clicking on **File —> Examples —> Servo —> Sweep**.

You first include the Servo library (“**#include <Servo.h>**”), and create the variable **myservo** that you can use as a handle to the Servo object (“**Servo myservo;**”).

In the setup function, you tell the Arduino that the control wire from our servo motor is attached to digital pin 9 ("**myservo.attach(9);**").

The work is done in the loop() function, where you use the "for loop" to count from 0 to 180, and another one to count backwards, from 180 to 0. This has the effect of the rotor of the servo motor traveling 180 degrees to one side, and 180 degrees to the other side, 1 degree at a time, constantly.

Inside each for block, you first write a value to the motor using **myservo.write(d)**, where "d" is a number representing the degree to which the shaft should turn. If we want to turn it by 15 degrees, we write **myservo.write(15)**.

Simple, right?

In Block 1, you get the rotor to turn from 0 to 180 degrees, and in Block 2 to travel all the way back to 0 degrees.

Finally, notice how you set a delay of 15 milliseconds inside each block, after a movement has been written? We need this because it takes a bit of time for the motor to move, and you want to make sure that any previous instruction has been completed before sending through the next move instruction.

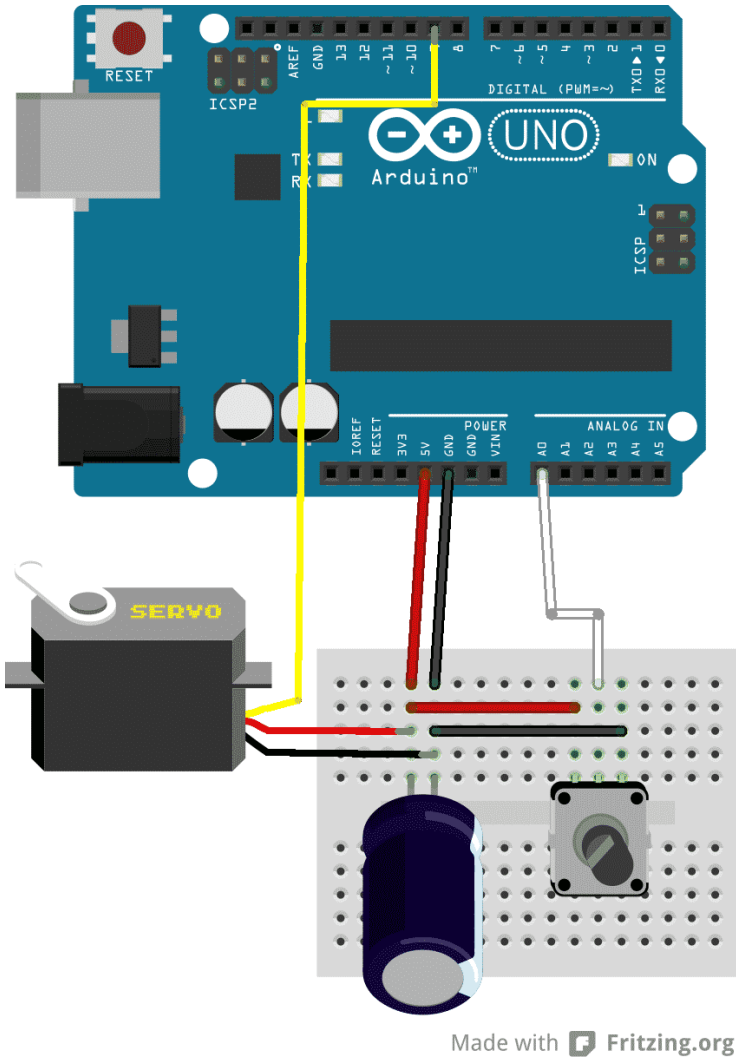
That was easy but not satisfying enough. I want to be able to control the servo motor myself, instead of the Arduino being in charge.

How about you try to connect a potentiometer and use it as a controller for the motor?

Control the servo with a potentiometer

Let's attach a rotary 10k potentiometer, and adjust our sketch to enable us control of the motor by turning the knob.

Here's the new circuit:



Connect the middle of the potentiometer pin to analog pin 0 (A0) on the Arduino. The other two pins connect to +5V and GND.

The new sketch looks is this:

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

```
// create servo object to control a servoServo myservo;

// analog pin used to connect the potentiometer
int potpin = 0;
// variable to read the value from the analog pin
int val;

void setup()
{
// attaches the servo on pin 9 to the servo object
myservo.attach(9);
}

void loop()
{
// reads the value of the potentiometer (0 to 1023)
val = analogRead(potpin);
// scale it to use it with the servo (0 to 180)
val = map(val, 0, 1023, 0, 179);
// sets servo position according to the scaled value
myservo.write(val);
// waits for the servo to get there
```

```
delay(15);

}
```

Just like in the first part of this project, you include the Servo library and set pin 9 as the servo pin.

In the loop() function, the Arduino constantly takes readings from analog pin 0 (A0) where the potentiometer is attached.

Because the range of values read in A0 is not the same as the values we can send to the servo, we use the map() function to scale appropriately.

Finally, we use the **myservo.write(val);** function to send the scaled value to the potentiometer.

Learn more

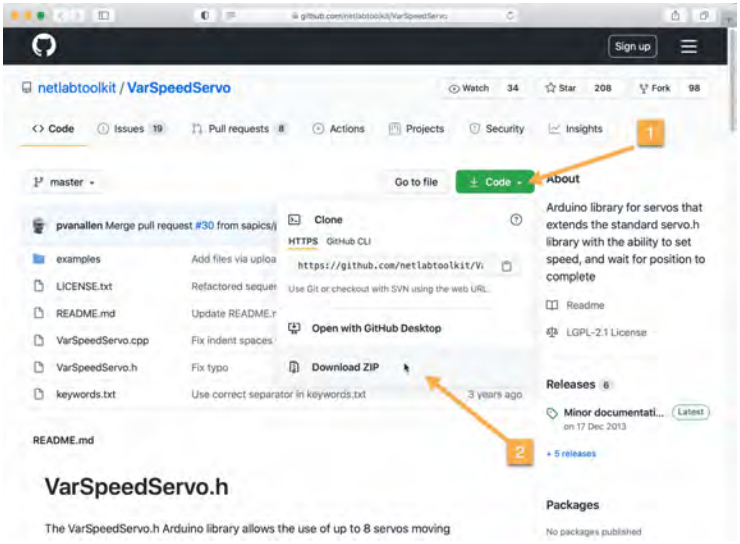
If you would like to learn how to use servo motors with your Arduino, consider enrolling to Arduino Step by Step Getting Serious.

We have a full section (Section 17) with 10 lectures dedicated to this topic.

Motors guide series

Project 2: Servo motor control with VarSpeedServo

Well-crafted libraries allow us to improve the quality of our gadgets with very little additional effort. In this article, you will learn how to use a simple library to improve your servo motor controller sketches with minimal changes in your code.



The build in Servo library is good and very simple to use. However, there’s more we can do with the hardware than it

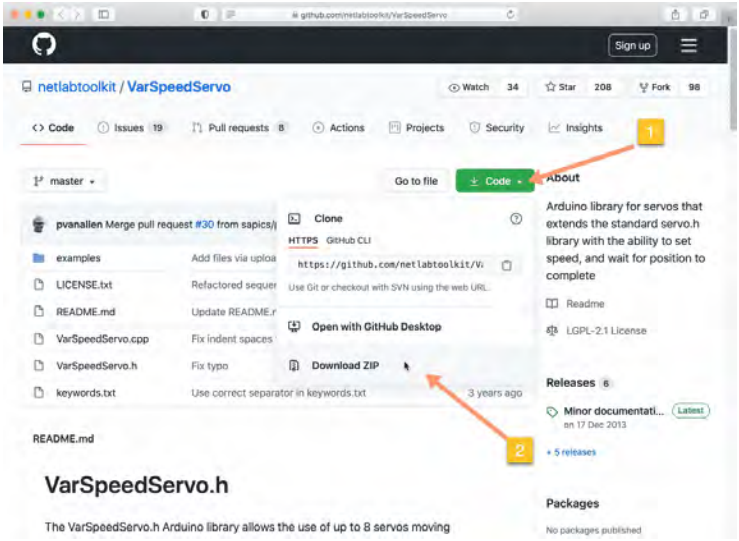
allows us.

For example, how fast should the motor move from one position to the next? What about the ability to define a set of movements for the motor to perform, and then send them with a single instruction?

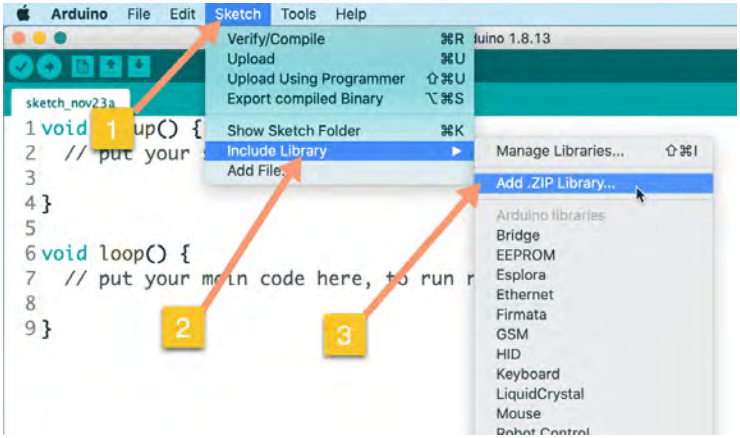
To achieve functionality like that, we need to use an external library. I am going to show you how to use the VarSpeedServo library, written by Korman and updated by Philip van Allen. Thank you to both for their work and contribution!

Download & install the VarSpeedServo library

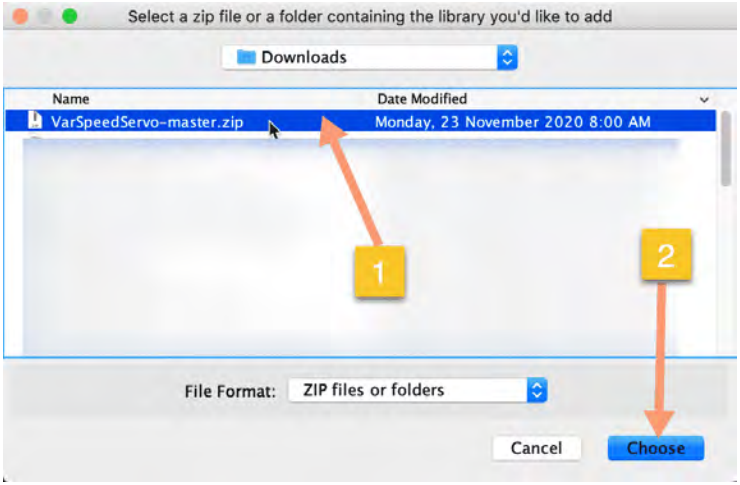
First, you will need to get this library. Go to the library Github repository to download it.Click on the green “Code” button to expand the drop-down menu (“1”), and then click on “Download ZIP” (“2”) as I show in the screenshot below:



Download it somewhere on your computer, and then import it to your Arduino IDE's libraries folder. The easiest way to do this is to use the Arduino IDE's Include Library option. Start your Arduino IDE, click on Sketch ("1") –> Include Library ("2") –> Add .ZIP Library ("3"), as in the screenshot below:

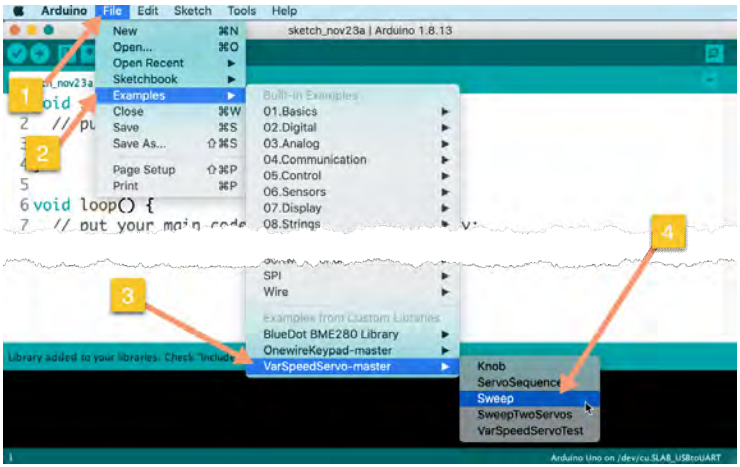


In the dialog box that appears, navigate to the location where the ZIP file is stored on your disk, select it, and click "Choose".



Load the “Sweep” demo sketch

The VarSpeedServo library is now installed and ready to use. The library comes with several examples. One of them is “Sweep”, which is perfect for an introduction to the library. Let’s open it. Go to the Arduino IDE and click on File (“1”) –> Examples (“2”) –> VarSpeedServo-master (“3”) –> Sweep (“4”).



Here’s the Sweep sketch:

#include <VarSpeedServo.h>

// create servo object to control a servo// a maximum of eight servo objects can be createdVarSpeedServo myservo;

// the digital pin used for the servoconst int servoPin = 9;

void setup() {

// attaches the servo on pin 9 to the servo object myservo.attach(servoPin);

```
// set the initial position of the servo, as fast as possible, wait
until done myservo.write(0,255,true);

}

void loop() {

// move the servo to 180, max speed, wait until done //
write(degrees 0-180, speed 1-255, wait to complete true-false)
myservo.write(180,255,true);

// move the servo to 180, slow speed, wait until done
myservo.write(0,30,true);

}
```

You can probably see the similarities and differences between the original Sweep sketch and this one.

In the header of this sketch, you include the *VarSpeedServo* library and create the ***myservo*** object. You set the servo signal pin to 9.

In the loop() function, you attached the servo at pin 9 to the ***myservo*** object.

You then send the first command to the servo by using the ***myservo.write()*** function. This looks the same as in the original Sweep sketch, except that is takes three arguments, not just one.

It looks like this: **myservo.write(position, speed, wait)**;

The function accepts three parameters:

- first, an *integer* which represents the degree we'd like the servo to turn to. For example, 15 degrees.
- Second, the *speed argument* which controls

how fast the movement should be. Top speed is 255, stopped is 0, and anything is between is allowed.

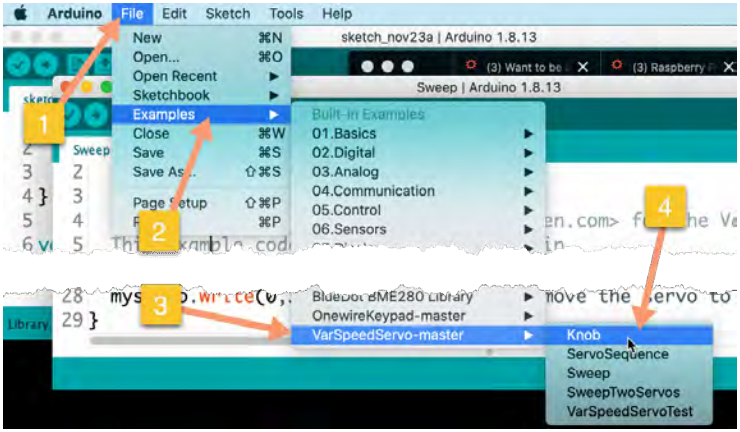
- And third, the *wait argument* that can block the program at this line and wait for the motor to finish its movement. To enable blocking set this argument to “true”. If you want to allow the sketch continue instead of block and let the servo finish its movement on its own, set this argument to “false”.

Nice, isn’t it?

What do you think happens in the **loop()** function?

Add a potentiometer

Let’s now connect the potentiometer, like we did in the second part of the first project, and load the VarSpeedServo example sketch titled **Knob** into the Arduino IDE. You will find this sketch under File (“1”) –> Examples (“2”) –> VarSpeedServo-master (“3”) –> Knob (“4”).



Here it is:

```
#include <VarSpeedServo.h>

// create servo object to control a servo

VarSpeedServo myservo;

// analog pin used to connect the potentiometer

const int potPin = 0;

// the digital pin used for the servo

const int servoPin = 9;

// variable to read the value from the analog pin

int val;

void setup() {

// attaches the servo on pin 9 to the servo object
```

```
myservo.attach(servoPin);

}


void loop() {

// reads the value of the potentiometer (0 to 1023)

val = analogRead(potPin);

// scale it to use it with the servo (0 to 180)

val = map(val, 0, 1023, 0, 180);

// sets the servo position according to the scaled value

myservo.write(val);

// waits a bit before the next value is read and written

delay(15);

}
```

Compare this against the sketch from the second part of the first project.

Isn't it almost identical?

Notice how the sketch above also makes use of the **myservo.write(val)** function, with just a single argument? The VarSpeedServo library allows you to use the familiar functions from the build-in Servo library, so that any existing sketches you may have lying around will still work. You get the additional functionality, as we saw in the Sweep sketch, for free!

An exercise

Can you build a circuit that contains a servo motor and three buttons?

- When you press button 1, the motor moves to 60 degrees.
- When you press button 2, it moves to 90 degrees.
- When you press button 3, it moves to 180 degrees.

Learn more

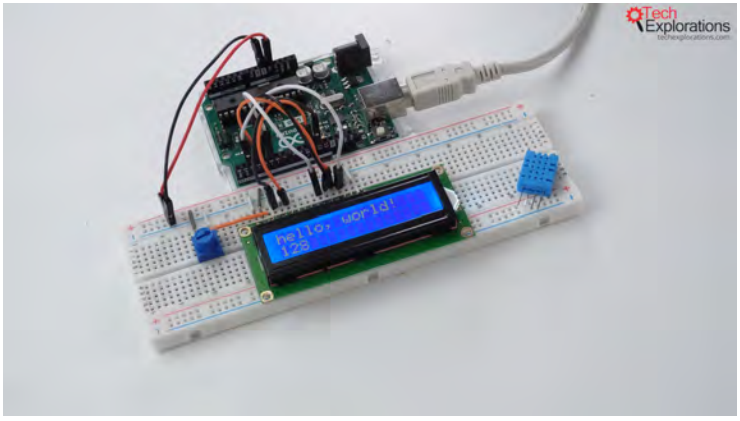
If you would like to learn how to use servo motors with your Arduino, consider enrolling to Arduino Step by Step Getting Serious.

We have a full section (Section 17) with 10 lectures dedicated to this topic.

The 2×16 Liquid Crystal Display (LCD)

In this guide, I'll show you how to incorporate a 2×16 Liquid Crystal Display into your Arduino-based projects. I'll go over the required wiring configurations as well as the sketch.

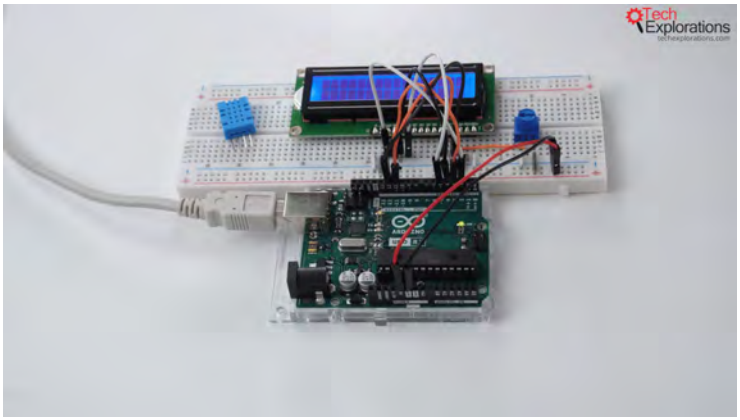
I recommend connecting the Arduino to the LCD module in 4-bit parallel mode, and I will provide instructions on how to upload the sketch and test the LCD.



Introduction

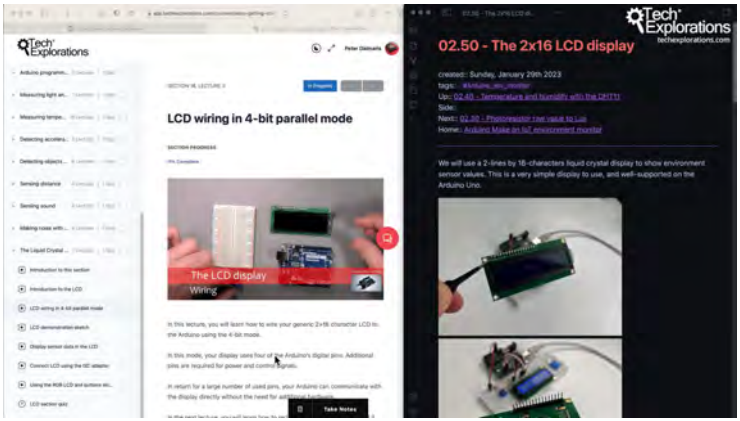
In this article, you'll learn how to use the **2×16 (two-line, 16-character each) Liquid Crystal Display (LCD)**. It's a common type of display used in numerous Arduino-based projects. We'll use this LCD in our project to display the

temperature and humidity readings from the **DHT11 sensor**. Our course Arduino IoT Environment Monitor Project teaches you how to implement this sensor.



The wired-up 2×16 LCD

As you can see in the image above, I have already wired up the LCD. I'll explain the setup, and after covering the various methods of wiring, I will discuss the sketch and display some demonstration text on the LCD to make sure everything is set up correctly.



LCD wiring in 4-bit parallel mode

Wiring configurations and the sketch

Let's take a look at the wiring and programming of the 2×16 Liquid Crystal Display. Outlined in this section is the simplest method for connecting this display to the Arduino used in our project. It uses more wires to transmit data from the Arduino to the LCD. However, there is no need for any additional hardware, making this the simplest wiring method.

Overview of displays

I want to take a moment to discuss displays more broadly. I thought it would be helpful to let you know that, as is the case with most electronics, the Arduino is compatible with a wide variety of displays. The Liquid Crystal Display (LCD) is probably the simplest way to show text characters, whether it be numbers, letters, or symbols. The LCD allows for that and more.

In addition to the standard LCD screen, the Arduino can also output data to the user via LEDs, LED alphanumeric displays, and, of course, dot-matrix displays, which excel at displaying graphics with very small, tiny dot pixels.

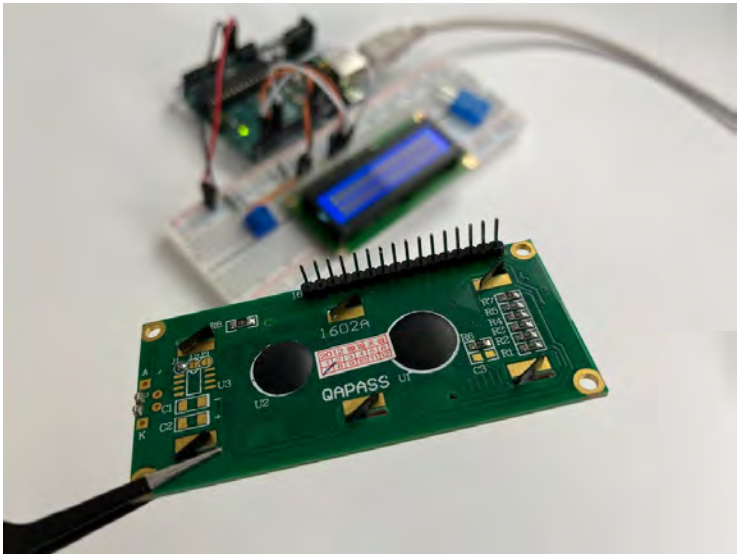
If you want to learn more about some of those displays, you can refer to my other course, Arduino Step-by-Step: Getting Started, where I have a dedicated section on the liquid crystal display. Also, on Arduino Step-by-Step: Getting Serious I demonstrate various types of displays, including dot-matrix and LED-type displays.

The 2×16 LCD screen

Back to the current topic, if you turn the 2×16 LCD that comes with the **3 in 1 IoT/Smart Car/Learning Kit** for Arduino by **SunFounder** around, it will look like the image below.

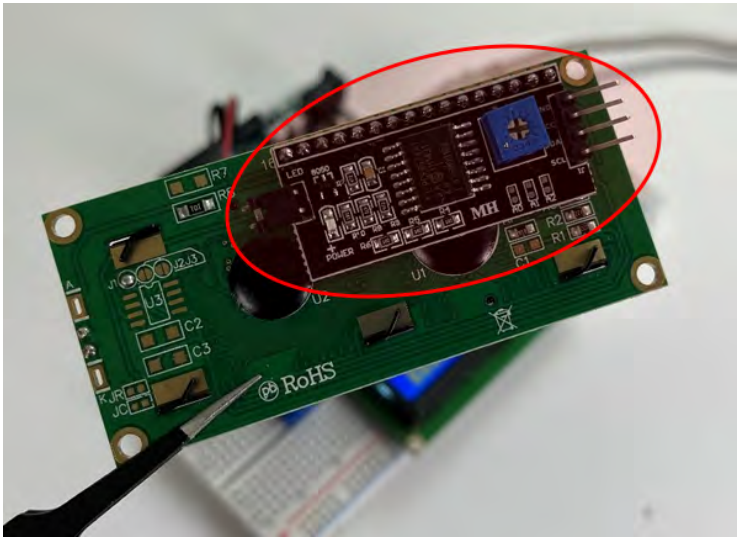
The 2×16 LCD included in the 3 in 1 IoT/Smart Car/Learning Kit for Arduino by SunFounder

If you turn the LCD we're going to use around, you can't see any parts other than the pins and bits and pieces that are mounted on the PCB.



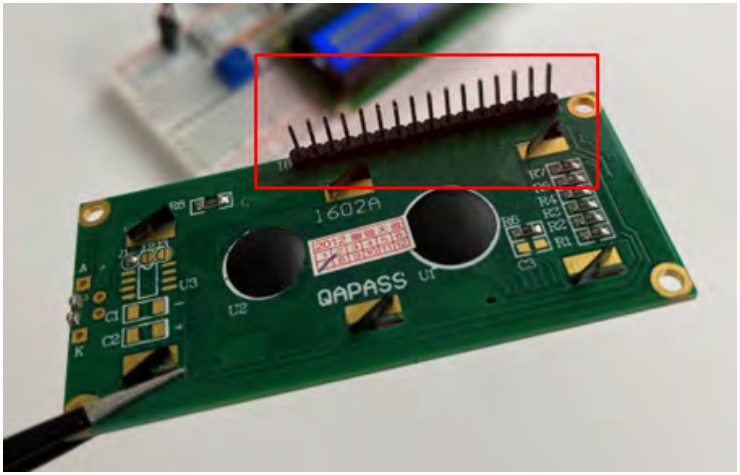
The LDC we'll be using in this project

The black module that you see in the image below is an I2C backpack.

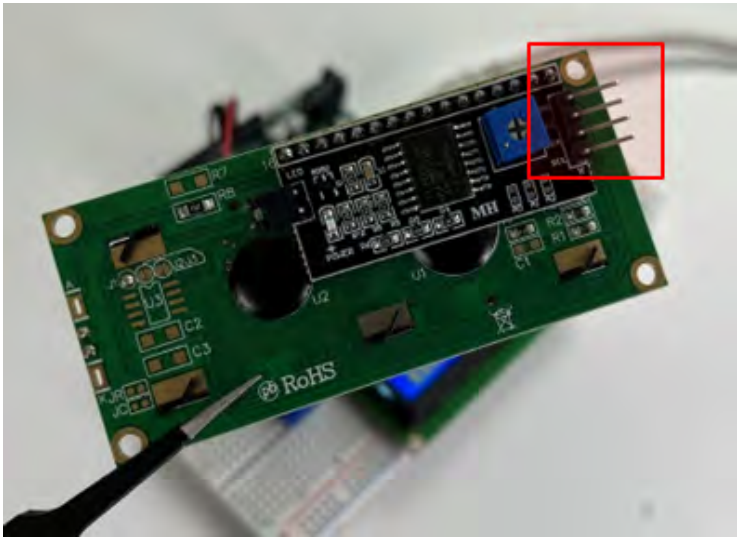


The I2C backpack

The I2C backpack converts the 16 pins visible on the base LCD module, to four pins on the converted module, which is the LCD with the black backpack module.



The 16 pins on the base LCD module



The 4 pins on the I2C LCD module

This allows you to simplify the wiring by reducing the number of wires required.

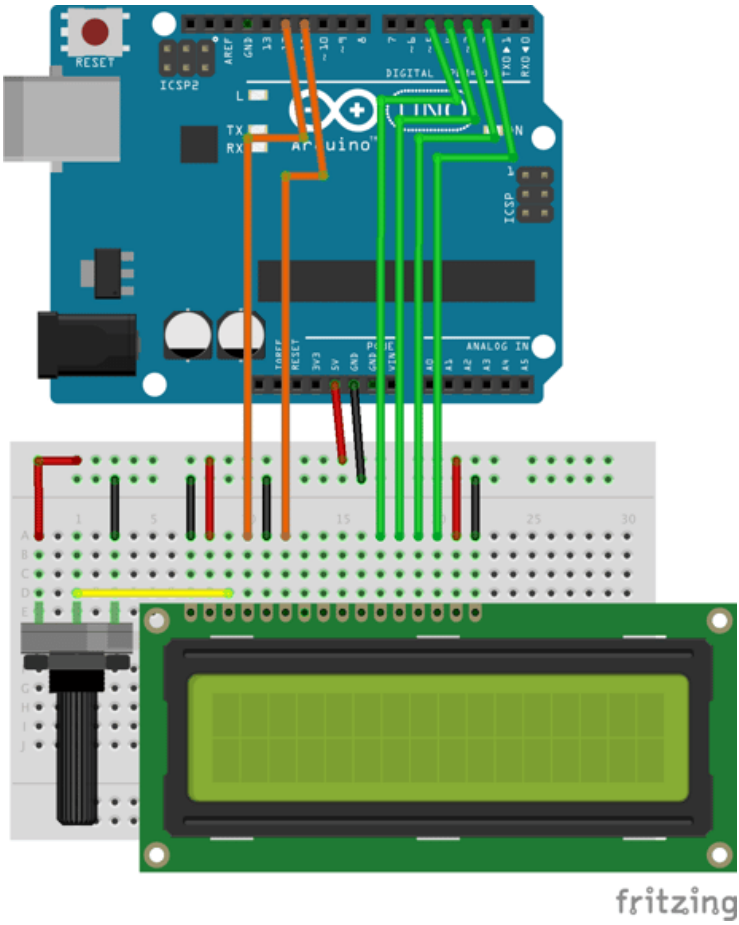
In our example, and I'll explain why in a moment, we'll be using the majority of the LCD module's back panel pins. When using the black backpack, you only need four pins, two for power and the other two for I2C data and the clock. Therefore, this is the preferred method of connecting the LCD to the Arduino, from a wiring perspective.

The issue here is that using the I2C backpack increases the **footprint** of your sketch. In my experiments, I found that the Arduino Uno's memory was too low to accommodate both the I2C-enhanced LCD and the other features we require in our sketch. The Arduino Uno's storage and RAM were taxed by the tasks of connecting to the internet and the **Blynk** platform and reading data from the DHT sensor. After turning on the LCD screen's I2C backpack functionality, the sketch stopped running because it was too much for the Arduino to handle.

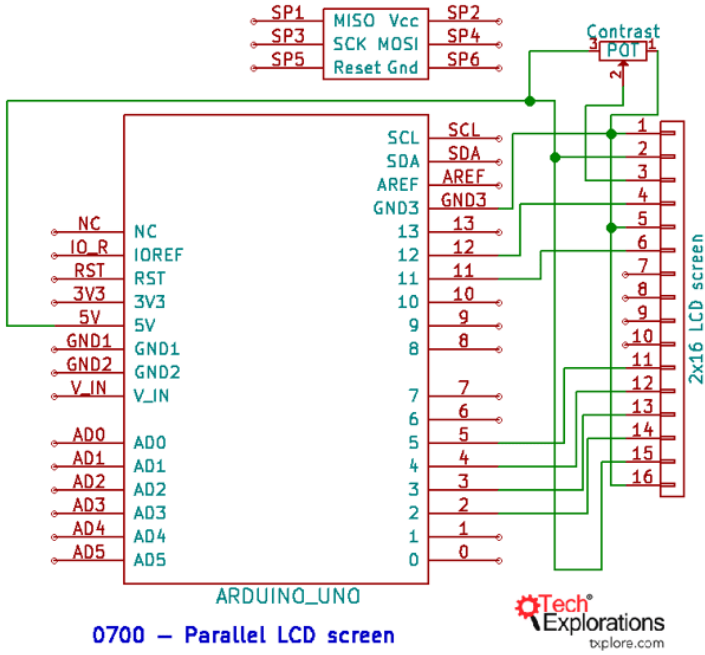
Therefore, I used the "standard" **4-bit parallel mode** to connect the Arduino to the LCD module. This particular LCD screen supports two parallel modes: 4-bit and 8-bit. If we use the 4-bit mode, we'll need four data pins. And if we use the **8-bit parallel mode**, we'll need to use eight data pins.

Wiring

In terms of the wiring, it looks like this:



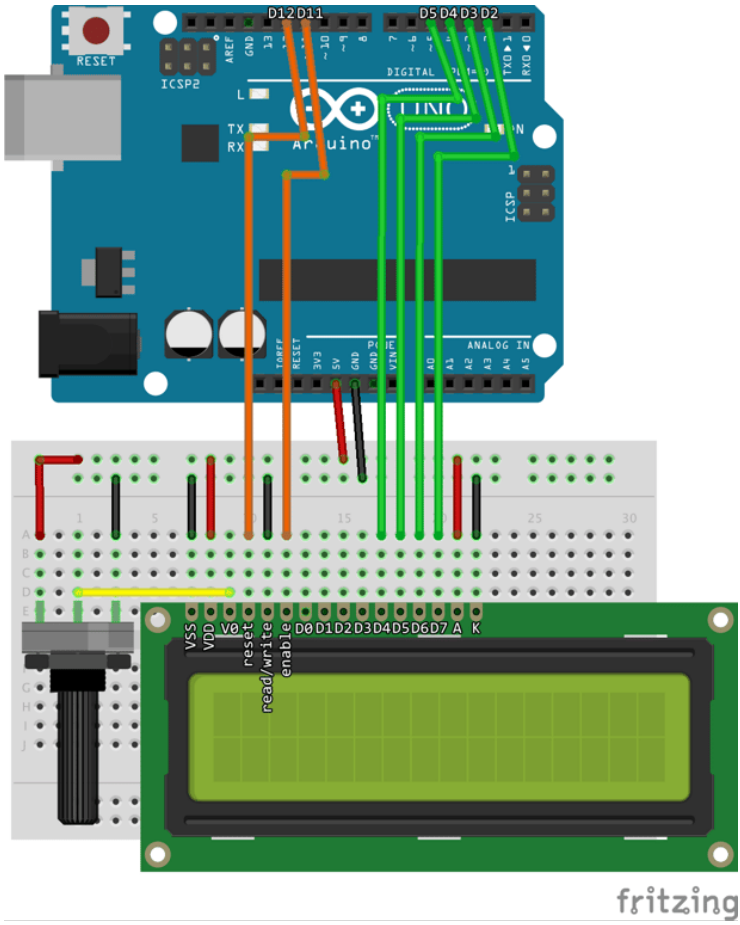
The Fritzing version of the sketch



The KiCad version of the sketch

The same wiring is shown above from two different design versions. For the left one, I'm using **Fritzing**, which is more graphical and representative. I'm using a **KiCad** schematic for the right one.

However, in either case, the green wires are used to transfer data in 4-bit parallel mode. As you can see, I'm using four of them. If I had used 8-bit parallel mode, I would have needed to use four more of those pins as well as four more Arduino pins to make the connection. Obviously, we wouldn't be able to connect anything else because we would have used up nearly all of the Arduino Uno's digital pins.



The 4-bit parallel connection mode

So we'll use a **4-bit parallel connection mode** for this LCD. Starting with the data pins, we'll connect them as follows: The rightmost green cable, as seen in the image above, will connect **pin D7** on the LCD module to **digital pin 2** on the Arduino. Because we are using a four-pin parallel mode connection, after connecting pins 6, 5, and 4, pins 3, 2, 1, and 0 will remain unconnected. The LCD module's four pins will connect to the Arduino's digital pins 2, 3, 4, and 5.

In addition to the data pins, we'll need the **reset** pin, which will be connected to Arduino **digital pin 12**. There is also the **enable** pin, which we'll connect to **digital pin 11** on the Arduino.

Then there's a **potentiometer**, which is discussed in more detail in the course. We'll use the potentiometer to control the **contrast** of the LCD and connect it so that **5-volt power** is applied to the leftmost pin. The other one on the other side will connect to **ground (GND)**. The middle pin will be connected to the third from the left pin on the LCD module, which is **V0**. V0 controls the voltage, which in turn controls the contrast of the LCD.

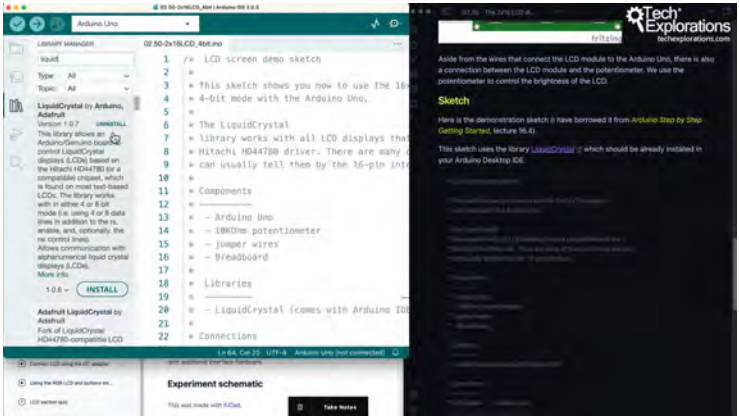
Back on the LCD, on the left side as seen in the image above, there's the **VSS pin**, which we'll simply connect to ground (GND). The **VDD pin** power will be connected to 5 volts. The **read/write pin** is permanently connected to ground.

Finally, on the other side, for the backlit, there is the **A (anode) pin**, which we'll connect to the breadboard's 5-volt power rail, and **K (cathode) pin**, which we'll connect to the breadboard's ground power rail. The last step is to connect the two power rails to the **5V** and **GND pins** on the Arduino.

The wiring process is now finished.

The Sketch

The sketch is very simple and relies on the LiquidCrystal **library**, which is already included in the Arduino IDE, so you won't have to install it separately. If you search in your library manager, you will find the LiquidCrystal library as shown below. You can access the library's documentation by clicking **More info**.



The LiquidCrystal library included in the Arduino IDE

See this sketch on GitHub.

I have simplified this sketch so that it only contains the absolutely essential components to drive the screen, but you can find the other available functions in the LiquidCrystal documentation.

LiquidCrystal Functions

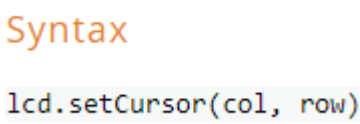
For example, I'll be using the constructor LiquidCrystal() for the liquid crystal, which is shown in this code block:

[tcb-script src="https://emgithub.com/embed-v2.js?target=https%3A%2F%2Fgithub.com%2Ffutureshocked%2FArduino-environment-

monitor%2Fblob%2Fmain%2F02.50-2x16LCD_4bit%2F02.50-2x16LCD_4bit.ino%23L59-L75&style=night-owl&type=code&showBorder=on&showCopy=on”][/tcb-script]

The code block above also includes begin(), which I’ll use to start the LCD, and print(), which I’ll use to print something out, in this case a bit of text. I’ll also use setCursor() to move the cursor to a specified position on the LCD.

You will get more information about these functions by exploring the documentation in detail. To use setCursor(), for instance, the parameters you would need to specify are the **column (**col**)** and the row.



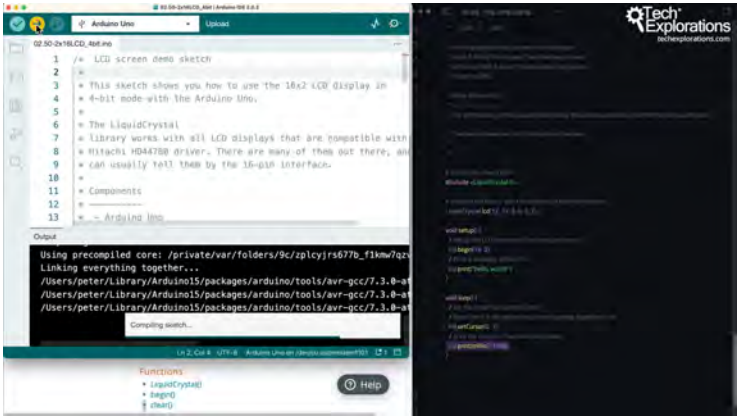
In this example, I specified column zero, which is the leftmost column, and row one, which is the bottom row in the display because we only have two rows. This will move the cursor to the desired location. Following that, we’ll print some numbers to represent the number of seconds since the Arduino started.

However, there are other functions that you can explore and experiment with. For example, you can use scrollDisplayLeft() to make text scroll to the left, scrollDisplayRight() to make it scroll to the right, or, if your text is longer than 16 characters, you can use autoscroll() to have it scroll automatically across the screen.

You can either read up on the subject in the documentation or head back to Arduino Step-by-Step: Getting Started to learn more about this LCD, including information on how to use the I2C adapter and communicate with the display using the I2C protocol, which requires far fewer wires.

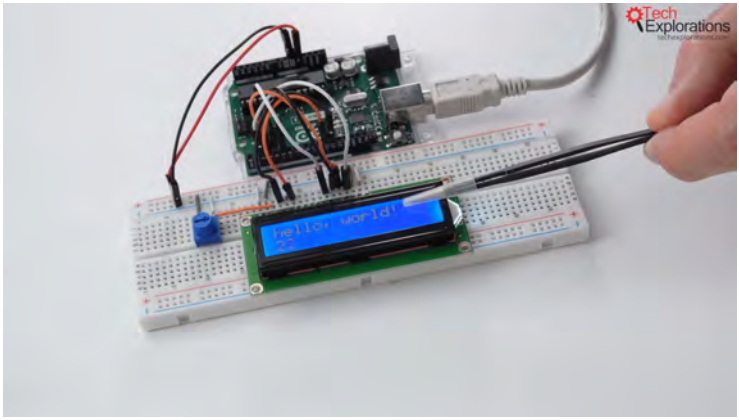
Sketch uploading and testing

I've assembled the hardware, and the sketch is ready to go, so I'll go ahead and upload it to the Arduino.



Uploading the sketch to the Arduino

So there you go!

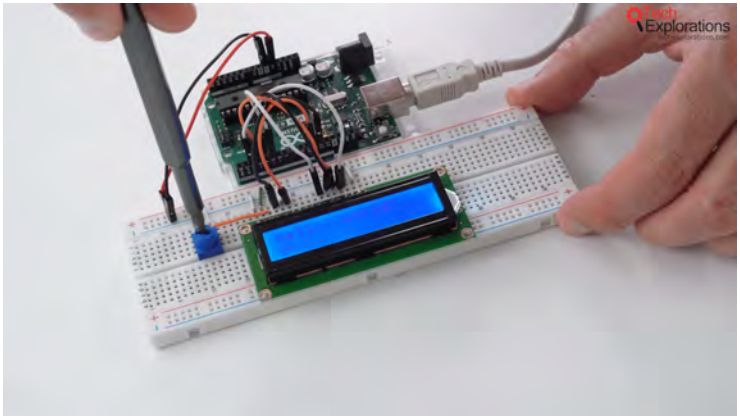


hello, world! is displayed in the first row of the LCD

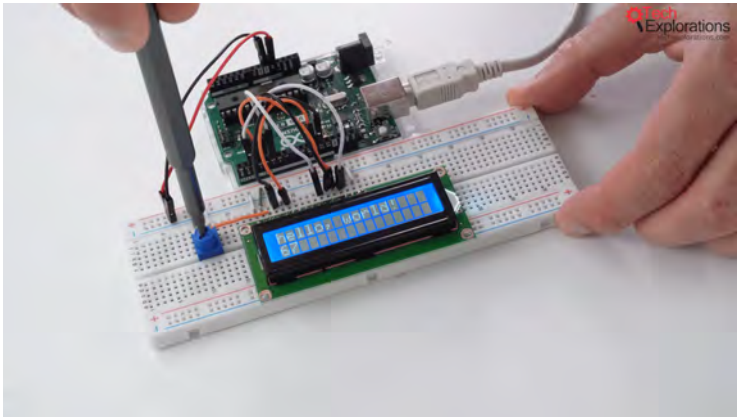
So, hello, world! is the fixed text displayed in the first row of

the LCD. In the second row I have a number that counts the seconds since the Arduino began its most recent operation.

I can simply use the potentiometer to adjust the display's contrast, and we can see the effect on the image quality.

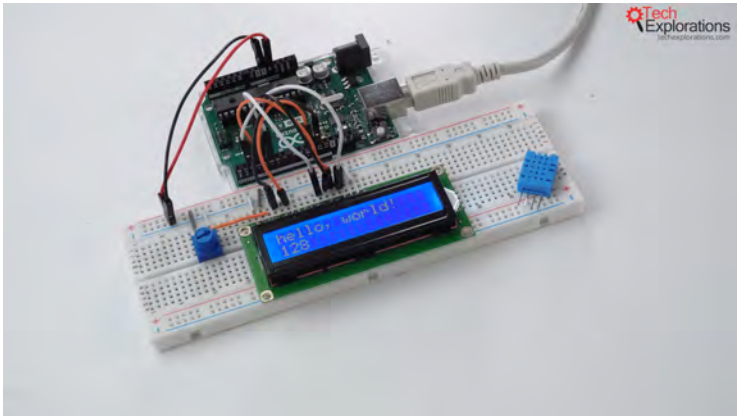


Low contrast

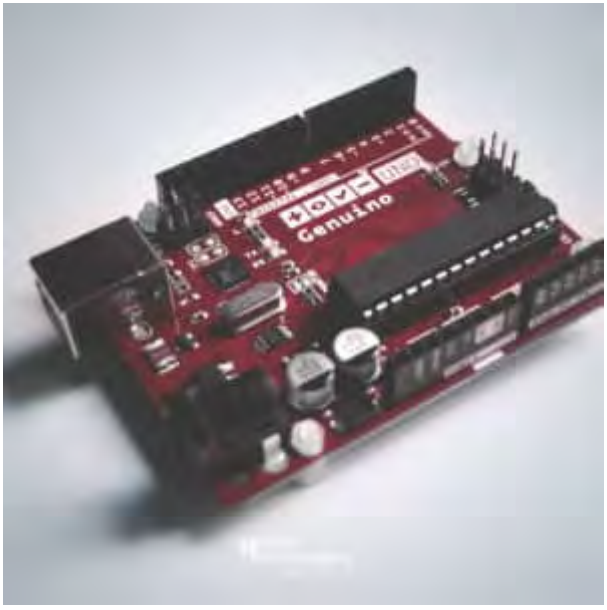


High contrast

And that's all there is to the LCD. You now have everything you need to use this liquid crystal display to monitor environment data from the DHT11 sensor.



The liquid crystal display is ready to monitor environment data from the DHT11 sensor



Done with the basics? Looking for more

advanced topics?

Arduino Step by Step Getting Serious is our comprehensive Arduino course for people ready to go to the next level.

Learn about Wi-Fi, BLE and radio, motors (servo, DC and stepper motors with various controllers), LCD, OLED and TFT screens with buttons and touch interfaces, control large loads like relays and lights, and much much MUCH more.

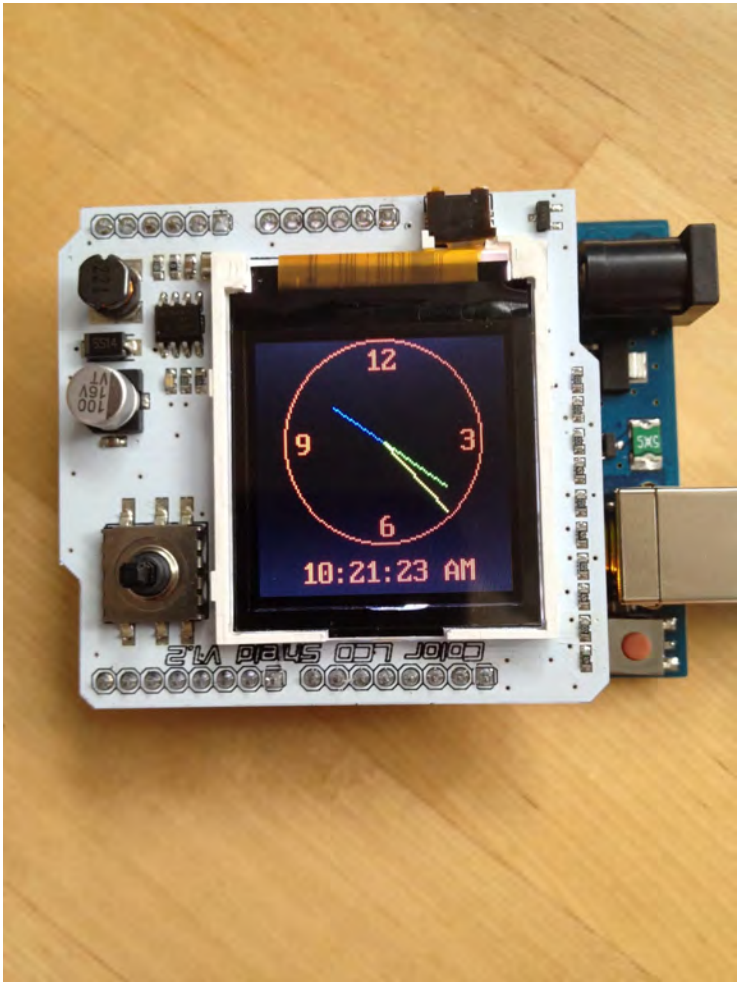
Learn more

Jump to another article

2×16 LCD – 4-bit parallel wiringLCD screen – I2C wire wiring (soon)The Seven Segment Display (soon)128×64 OLED (soon)

TFT LCD Screen

This tutorial will show you how to use the TFT LCD screen shield with an Arduino. I'll go over how to install the necessary library, how to connect the shield to the Arduino, and how to upload example sketches. In addition, we will investigate the functionality of the shield's buttons, and I will suggest exercises to help you better understand the shield's capabilities.

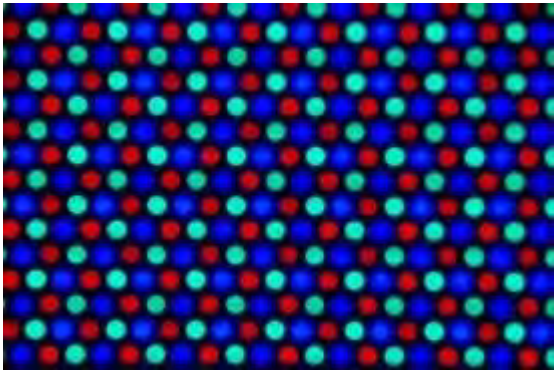


Introduction

The **Thin-Film Transistor (TFT) liquid crystal display** screen belongs to the **LCD** family of displays. **TFT** screens are composed of numerous pixels, which are tiny screen elements that can be turned on and off to create visible markings. These pixels are arranged in rows and columns.

A TFT computer screen can contain many millions of pixels. In small consumer electronics, such as feature phones, there could be several tens or hundreds of thousands of pixels.

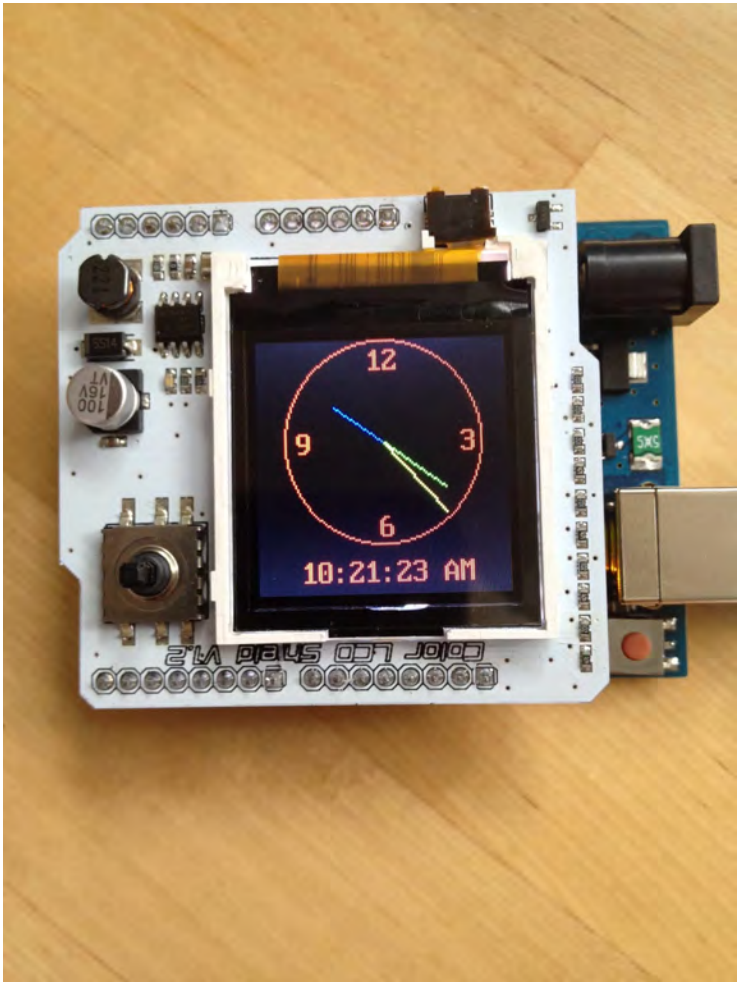
If you use a magnifying glass on your computer screen, you will see these tiny pixels. In the color example below, notice that each pixel is actually made up of three: one for green, one for blue, and one for red. The circuitry that controls the screen decides which ones to turn on, how bright they should be, and how long they should stay on. The end result is rich, colorful graphics.



TFT LCD screens are more versatile than LCD screens because there are no restrictions on the things you can display on

them. While a humble LCD screen can only display text, a TFT screen allows you to draw anything at all.

In the example below, I am displaying a clock on my Arduino's TFT display shield. Later in this article, I will provide several examples and show you how to display a clock on the TFT display.



A clock image on a TFT display

TFT screens consist of many pixel elements that require precise control. To achieve this, they come equipped with a screen controller IC as part of the package. Your Arduino communicates with this IC via a serial connection and makes requests for shapes to be drawn on the screen.

However, communicating with the screen controller can be complicated for anyone other than its designers. To simplify this process, libraries have been created. These libraries allow programmers to use simple instructions such as setCircle to draw a circle or setStr to draw a string of text on the screen.

In this article, I am using the first shield we played with in the course **Arduino Step by Step Getting Started**. A shield is a printed circuit board (PCB) on which several components are mounted, with pin connectors that match exactly those of the Arduino's headers. You simply plug the shield on top of the Arduino, and you are good to go. No wiring or soldering is required, which greatly minimizes the chance of making a mistake or breaking something. This is a good way to experiment with more complicated devices, such as Ethernet adapters, SD card readers, motor controllers, etc.

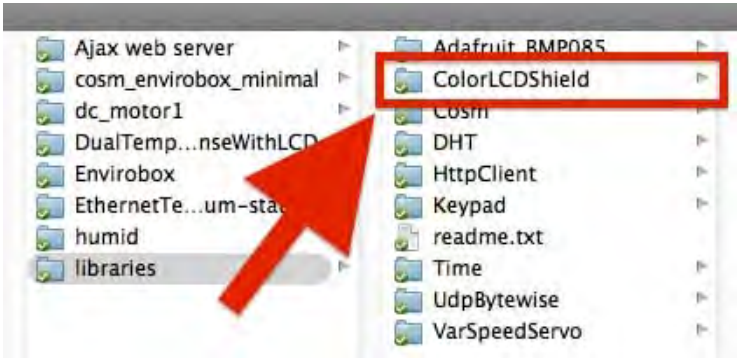
Demo 1: quick start guide

To get started, you should download and install the library that matches your screen. In my case, I purchased a screen from eBay. After searching on Google, I found that the manufacturer's website had the best matching library.

2023 Note: the LCD shield shown may no longer be available in the seller's website

To make the library available to your Arduino project, and to get access to the example sketches, follow the exact same procedure as you did back in the servo lecture:

1. Download and extract the archive file.2. After extraction, make sure that the folder name matches exactly the name of the .h file inside it (the folder name should not include the ".h" extension). For this library, the folder name should be "ColorLCDShield". Capitalization matters.

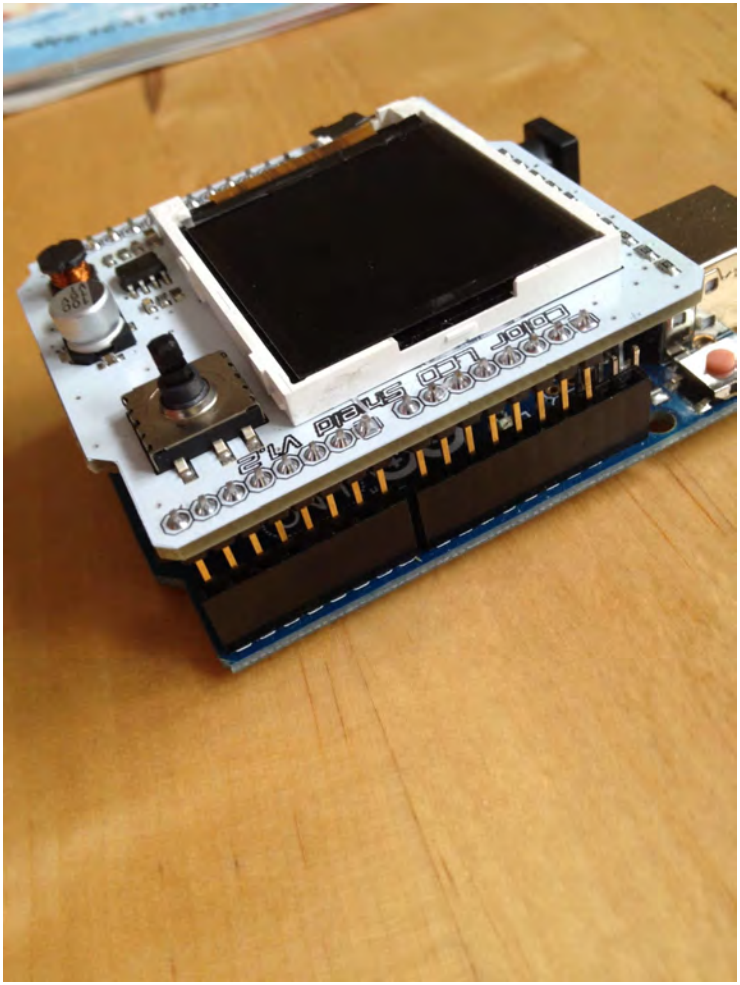


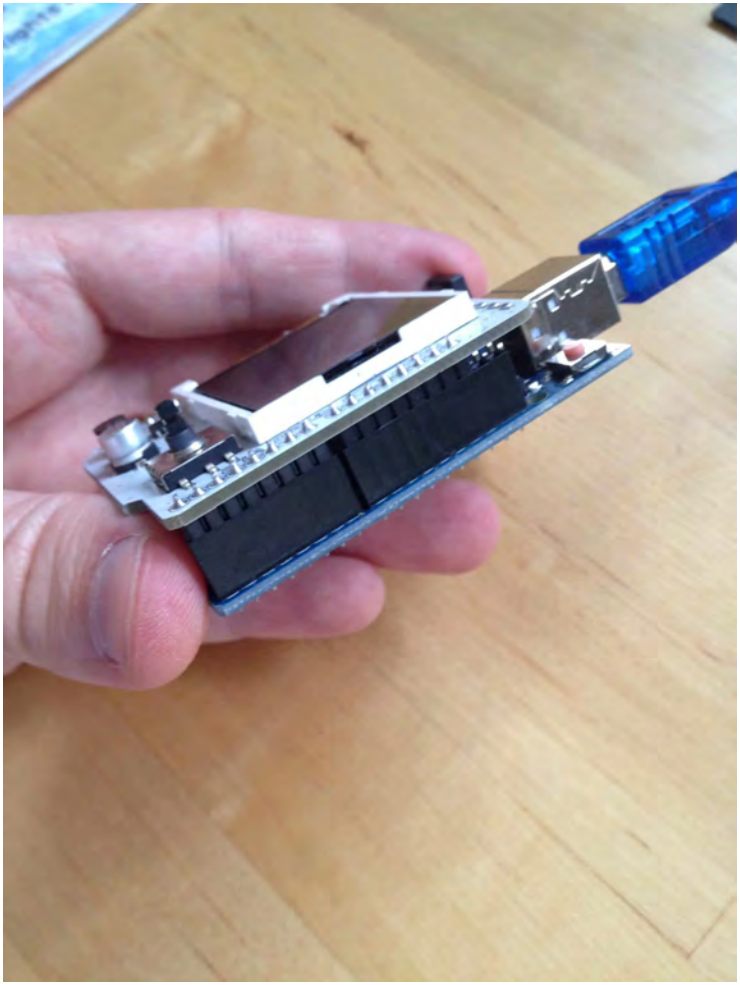
The folder of the ColorLCDShield library

3. Copy the folder to your Arduino's library folder. If you're not sure where that is, check your Arduino IDE preferences.4. Restart the Arduino IDE.

To view the ColorLCDScreen examples, click on **File > Examples > ColorLCDShield > Examples**.

Next, plug the shield onto the Arduino. The headers on the Arduino are designed in a way that makes it difficult to connect the shield incorrectly without causing damage to its pins. Play around with the shield and Arduino to ensure perfect pin alignment, and then apply gentle force to securely connect the two components.


Color LCD Shield
V1.2



Now that we have the shield installed, let's upload one of the example sketches. Plug the Arduino into your computer, open the Arduino IDE, and navigate to **File > Examples > ColorLCDShield > Examples > ChronoLCD_Color**.

Below is the sketch. I am showing you only the relevant part at the moment.

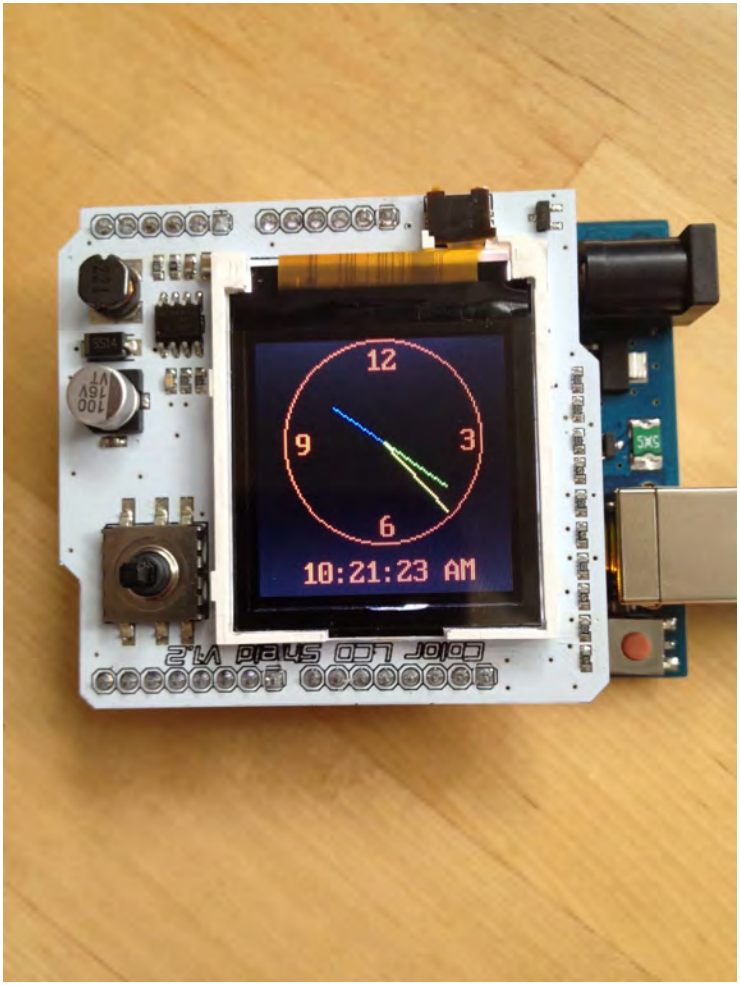
[tcb-script

src=”https://emgithub.com/embed-v2.js?target=https%3A%2F%2Fgithub.com%2Flina-txplore%2Ftechexplorations%2Fblob%2Fmain%2FColorLCDSciEld.ino&style=night-owl&type=code&showFullPath=on&fetchFromJsDelivr=on”][/tcb-script]

Currently, the most important consideration is to determine the type of screen controller used by your screen. Unfortunately, there are no external markings to assist you, and often the manufacturer does not provide any documentation. Therefore, determining the controller type is usually a process of trial and error. The controller will be either **PHILLIPS** or **EPSON**. In my case, it happens to be **PHILLIPS**.

Upload the sketch and check if it works. If it doesn’t, edit the line to “EPSON” and see if it fixes the issue.

Wait a few seconds, and a clock should appear on the screen.



A clock image on a TFT display

There’s a lot happening in this sketch, so it’s better to leave it alone for now. Instead, let’s look at something simpler.

Keep everything connected and load this sketch into the IDE:

[tcb-script
src=”https://emgithub.com/embed-v2.js?target=https%3A%2F

%2Fgithub.com%2Flina-txplore%2Ftechexplorations%2Fblob%2Fmain%2FColorLCDShieldSketch.ino&style=night-owl&type=code&showFullPath=on&fetchFromJsDelivr=on”][/tcb-script]

When you upload it, you should see the following on the screen:

The sketch starts by including the LCD library. Next, we define three constants. A constant is simply a name given to a value, so that when we need to use it later, we can refer to it by name. Constants also make code more readable, especially when the values assigned to them are “esoteric” data such as hexadecimal or binary values.

In this sketch, we define the background color as follows:

#define BACKGROUND BLACK

Note that regular lines in the C programming language (which the Arduino language is based on) do not require an ending semicolon (;). In this code, **BACKGROUND** is a constant with a value of **BLACK**.

Later in the sketch, there is a reference to this code:

lcd.clear(BACKGROUND);

Here, we are calling the clear function of the **lcd** object and passing the **BACKGROUND** constant reference to it. Alternatively, we could have written

lcd.clear(BLACK);

and the result would have been exactly the same. However, consider this: the same code appears in several places within this sketch. If we ever decide to change the background color to pink, we will need to find all references to the clear function and modify its attribute. With a constant reference as an argument, we can simply go to the definition of this reference at the top of the sketch and change **BLACK** to **PINK** there. Problem solved!

The next line of interest is:

LCDShield lcd;

We declare an object reference of type **LCDShield**. We can now use this reference, **lcd**, to call all of the functions that the **LCDShield** object class offers.

In the setup() function, we configure digital pin 10 to provide power to the screen's backlight.

pinMode(10, OUTPUT);analogWrite(10, 1023);

We set the output to maximum brightness by writing a value of 1023 to it. If you want your screen to be dimmer, simply reduce the value written to pin 10.

Next, we inform the lcd object about the type of controller chip it is communicating with:

lcd.init(PHILLIPS);

In my case, my screen uses a **PHILLIPS** controller. Keep in

mind that the alternative is an **EPSON** controller.

Next, we set the contrast as follows:

lcd.contrast(40);

Adjust this parameter to observe its effect on contrast.

Lastly, these two commands modify the displayed content:

lcd.clear(BACKGROUND);lcd.setStr(“STARTING”, 50, 0, C_COLOR, BACKGROUND);

The first line makes everything black because at the beginning of the sketch, we defined the constant **BACKGROUND** to be black. Next, we write our first string using the setStr function, which takes several parameters.

lcd.setStr([text to show], [start vertical pixel – y], [start horizontal pixel – x], [text color], [background color]);

To write something at the top-left corner of the screen in red with a green background, you would use the following code:

lcd.setStr(“hello”, 0, 0, RED, GREEN);

Simple, isn’t it?

Within the loop() function, we begin by defining an array that contains three pieces of text:

char* Str4[ ] = {“arduino 1,”arduino 2,”arduino 3”};

Note that the definition is char* Str4[];. The char* indicates to the Arduino that the variable that follows is a pointer (indicated by the * symbol) to a location in its memory where a sequence of characters (indicated by the char data type) are stored. Since a string of text is made up of many characters, we use char. The pointer is then given a name, Str4, and we indicate that it is actually an array by using square brackets

after the name: [].

Finally, we initialize a pointer variable to an array of strings using curly brackets, with the 3 strings inside: {“arduino 1”, “arduino 2”, “arduino 3”}.

We iterate through this array using a for loop. For each item in the array, we call the printString function, which is defined at the end of this sketch.

The printString function requires three parameters: the string to write on the screen, the horizontal (x) start position, and the vertical (y) start position.

void printString(char* toPrint, int x, int y)

You may also remember that void means that a function does not return a value. Instead, it performs an action (like writing text on the screen) and then finishes execution.

In printString, there is a single call to the setStr function, just like we saw in the setup() function earlier.

With this knowledge, you now know how to write text on the screen.

Demo 2: Use the buttons

In this second demo, I will show you how to use the five buttons integrated into the joystick that comes with the shield.

Here is the sketch:

[tcb-script src=”https://emgithub.com/embed-v2.js?target=https%3A%2F%2Fgithub.com%2Flina-txplore%2Ftechexplorations%2Fblob%2Fmain%2Fdemo2sketch.ino&style=night-owl&type=code&showFullPath=on&fetchFromJsDelivr=on”][/tc

b-script]

Here's the first one:

int buttonPins[5] = {A0, A1, A2, A3, A4};

In this code snippet, we declare an array that will contain references to the five Arduino analog pins that convey the status of the five buttons provided by the LCD shield. These buttons are integrated into a mini joystick soldered onto the PCB, next to the screen. Rather than using individually declared variables for the buttons, which we've done in previous sketches, we bundle them in a single array. We then cycle through this array to set them up or to read their values, using and reusing the same code.

That is what we do in the very beginning of the setup() function:

for (int i=0; i<5; i++)

{ pinMode(buttonPins[i], INPUT);

digitalWrite(buttonPins[i], HIGH); }

In this code, we cycle (loop) through the contents of an array that contains the references to the button pins. First, we set them as inputs, and then we turn on their pull-up resistor. This pull-up resistor keeps the voltage of the switch high. When we close the switch by pressing it, the voltage goes down, and the Arduino, guided by the code in the takeInput() function, detects the event.

In the loop() function, we constantly call the takeInput() function and then call the printString function.

Since we have already covered the printString function in Demo 1, let's focus on takeInput() instead.

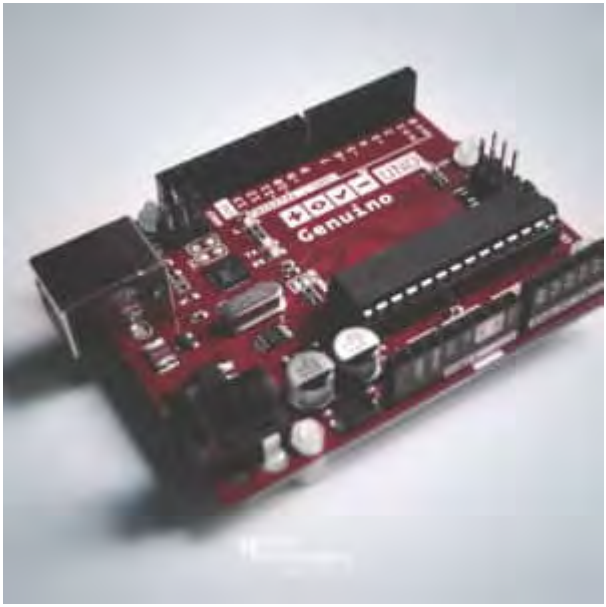
Here, we individually examine the state of each button. When

you push the joystick to the side, one of the buttons creates a connection to **Ground**, so the digitalRead function returns **FALSE**. Using the ! operator, we can invert this **FALSE** value and get **TRUE** instead. With the if function, we examine this value, and if it turns out to be **TRUE**, the Arduino will continue within the block and execute any instructions in it, such as clearing the screen and writing text on it.

So, now you know how to use the TFT LCD screen shield with the Arduino. However, there is more to learn, and the best way to do so is through exercises.

Exercise

Take a look at the examples included with the **ColorLCDShield** library. Can you figure out how to draw a line, a rectangle, and a circle?



Done with the basics? Looking for more advanced topics?

Arduino Step by Step Getting Serious is our comprehensive Arduino course for people ready to go to the next level.

Learn about Wi-Fi, BLE and radio, motors (servo, DC and stepper motors with various controllers), LCD, OLED and TFT screens with buttons and touch interfaces, control large loads like relays and lights, and much much MUCH more.>

Learn more

Jump to another article

2×16 LCD – 4-bit parallel wiringLCD screen – I2C wire wiring (soon)TFT LCD screenThe Seven Segment Display (soon)128×64 OLED (soon)