

Chapter 4 – Blinking a LED

May, 2020

David Ackmann – Gateway Division NMRA

For most model railroaders, their first attempt at programming an Arduino is to make a LED blink. These have lots of applications for this, like a beacon on top of a water tower, lights on top of traffic cones, traffic lights, emergency vehicles and many more (see “Project 2 – Lighting a Building” for blinking LEDs “on steroids”). This document tells you how this is done.

For starters, we are going to try to keep this real simple. We are going to use a red LED and turn it on and off in one second intervals. But I can tell you right off the bat that if we wire a LED directly into an Arduino, we will burn it up because the voltage coming from an Arduino (5VDC) is too high for most LEDs. We have to wire a resistor into the LED to “calm down” the power. But how big of a resistor do we need? Too big and the LED won’t light, but too small and the LED will “let out its magic smoke”.

And speaking of “smoke”, have you ever had a “short circuit” in your house? A short circuit happens when too much electricity runs unimpeded from a positive voltage directly to ground. At home, with 120Volts, it makes a loud “BANG”, there is smoke, and hopefully a circuit breaker trips to keep the wires from overheating and starting a fire. It isn’t pretty, and it happens because more current flows through a circuit than it is designed to handle. The same thing can happen with an Arduino if you ask an output pin to carry more current than it is designed to handle.

The specifications for an Arduino output pin says they supply 5 Volts, and can carry up to 40 milliamps (abbreviated ma). Now I have to lay a bit of electronic theory on you called “Ohm’s Law”. The law states that to avoid a short circuit, there has to be some electrical resistance, measured in “Ohms” between the high voltage and the ground. And there is a formula for that:

$$\text{Volts} / \text{Amps} = \text{Resistance}$$

If we apply the 5 volts that an Arduino pin supplies, and divide it by the 40mA current which is the maximum an Arduino can tolerate ($5 / 0.040$) we learn that the minimum resistance that must exist between a pin and ground is $5 / 0.040$ or 250 ohms (ohms are usually abbreviated with the Greek letter “Omega”)



But why is this important when blinking a LED? It's important because a single LED does not provide enough resistance keep the current under 40ma. We need to add some additional resistance in the form of a "resistor". But resistors come in many sizes, so how large of a resistor do we need? To find out, we first need to know the specifications of the LED we wish to use.

My box of LEDs has different specifications for each color. For the

red LED, we are told that it runs at 2.2V and can tolerate up to 20mA without self-destructing. So how much resistance does it supply? Plugging in the numbers from the specification into Ohm's Law, we calculate the resistance of this LED by dividing 2.2 volts by 0.020 amps, yielding a resistance of 110Ω. But we just learned that an Arduino circuit must be at least 250Ω to avoid short circuiting. But since the LED already provides 110Ω of resistance, we can subtract 110Ω from the 250Ω, which leaves 140Ω required to keep the LED protected from the 5V Arduino. That's the theoretical minimum resistance we need to provide in the circuit to avoid burning up the LED, so we need to solder in a 140Ω resistor or larger (probably up to about 330Ω), to protect ourselves from shorting out the Arduino (but too large of a resistor will restrict the current so much current that the LED won't light at all). The point of all this science and math is that we need to solder in a resistor to one of the wire "leads" of a LED to keep it from burning up, and that the size of the resistor for a red LED needs to be between about 110 and 330 Ohms. The smaller the value of the resistor, the brighter the LED will shine; a larger resistor will make it dimmer.

But look sharp! See that the Green, Blue and White LEDs run at 3.2 Volts, not 2.2V as the red one needed, so when we do the math for these LEDs, we see that 3.2 volts divided by 0.020 amps gives us a resistance of 160 ohms for the LED alone. If we want to keep the current for the resistor/LED combination still at 20 milliamps, we take the 250 total ohms for the circuit, minus 160Ω for the LED, which means that these three LED colors need a resistor of 90 ohms or higher. The values for your LED may vary!

I really hope that you "do the math" based on the specifications of your LEDs, but if doing so makes you feel ill, just use a 220 ohm resistor and "forgetaboutit". If your LED is dimmer than you want, use a smaller resistor until you start burning them up or frying your Arduino. If they are too bright, use a larger valued resistor. Also, the Arduino can supply only 200 milliamps for all pins combined, so if you are using a lot of LEDs you might need to use resistors higher than the minimum to tone things down and avoid harming the Arduino.



1280
Pcs

AMAZON sells resistors, as do other electronics supply stores. Visit AMAZON and search for “Resistors 1/4W” (the “W” stands for “wattage”, which is a combination of volts and amps, and 1/4 watt is strong enough for just about anything we will encounter in model railroading). Resistors in bulk are about a penny a piece, and you can get a combination pack for about \$13, with 20 resistors at many different values (AMAZON ASIN: B07L851T3V). I don’t need all those different values, so I might restrict my search to just the values I need for LEDs, and the common values between 90 and 330 ohms

are 100, 120, 150, 180, 220, 270, and 330 ohms, so add something like “220 ohms” to your search criteria. Limiting the search to a small number of specific values does drive up the unit price, but I still can find a pack of 100 resistors for under \$5. If I was starting out, I would probably buy something at the lower end of the range and maybe one in the middle. Your choice, but remember to start by reading your LED’s specifications.



Now take a look at your LED leads, and you will see that there is a short wire and a long wire, respectively called the “cathode” and the “anode” leads. It doesn’t matter which lead you solder to the resistor, but make a choice, and keep doing it that way; make it a habit. Me, I solder resistors on the short side, but that’s just me.



After I solder the resistor to the cathode, I take a black jumper wire with a male pin on one end, cut off the connector cut from the other end, and strip about 1/4" of insulation from the end, and solder it to the other end of the resistor. Alternatively, you can take a piece of some plain black 24 gauge wire (actually anything between about 20 gauge and 28 gauge wire will work, and the color doesn’t really matter, but start a habit and be consistent) and use plain wire instead of a jumper. Slip some shrink wrap tubing over the entire joint, and use a heat gun to shrink the insulation. After working on the cathode lead, I then take some different colored jumper or plain wire (I usually use red) and solder it to the longer lead, the Anode. Use heat shrink. You are now ready to connect the protected LED to the Arduino.

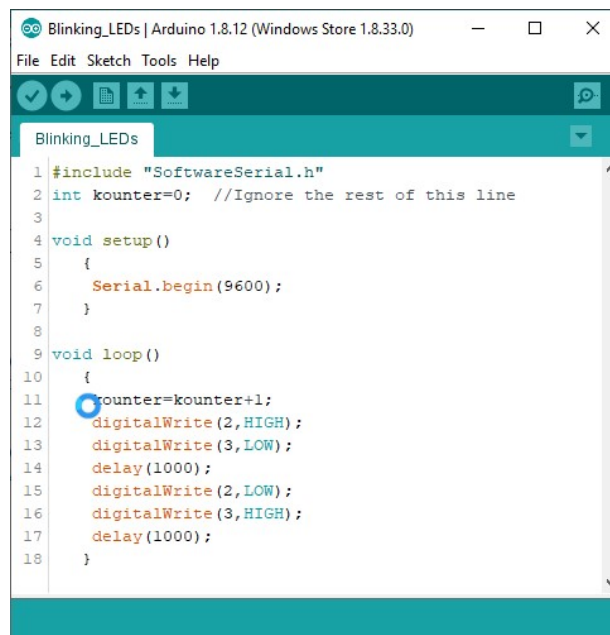
Any digital pin will work to connect the LED Anode to the Arduino; I am going to use pin D2. Connect the other lead, the Cathode, to a pin labeled GND for “Ground”; there is a ground pin on either side of the Arduino, and you may use either one. Plug your Arduino into its USB cable and the other end of the cable into a USB port on your PC. The hardware is now connected and we are ready to start the Interactive Development Environment application, the IDE.

Now we need to load a sketch into the Arduino to make the LEDs blink, and I have a small sketch that does just that in my GitHub account, ready for you to download a copy and paste it into the IDE. This is accomplished by visiting

<https://daackm.github.io>

and scrolling down to the Chapter 4 subheading. There you will find an entry to view a small program to test blinking LEDs; click on it.

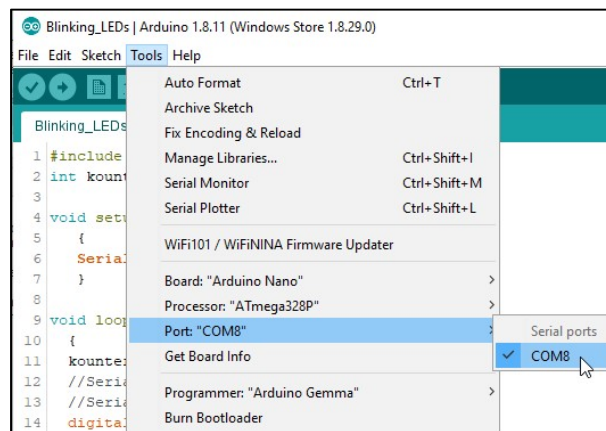
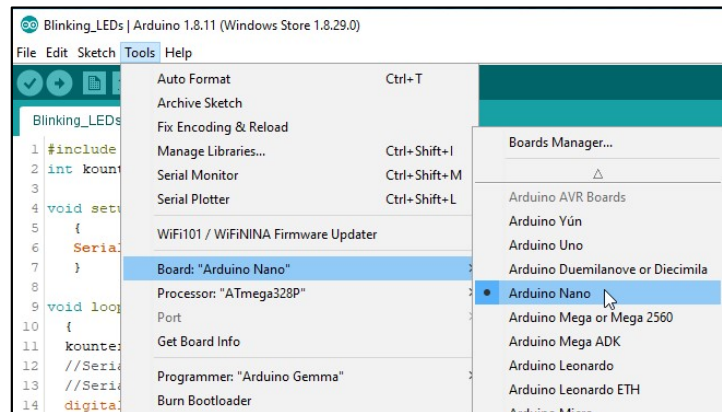
When you do so, the code will appear. Highlight the code and copy it by hitting “Ctrl/C” (hold the Ctrl key while simultaneously hitting the “C” key) from your keyboard. Then launch the Arduino IDE, click on “File”, then “New”. Highlight all the code lines in the new file and then delete them with your keyboard’s “Delete” key, and then hit “Ctrl/V” to paste my “BlinkingLEDs.ino” code into the IDE. Do a “File”, “Save” and give the sketch a name on your PC.

A screenshot of the Arduino IDE window titled "Blinking_LEDs | Arduino 1.8.12 (Windows Store 1.8.33.0)". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu bar is a toolbar with icons for opening, saving, and running. The main text area shows the code for "Blinking_LEDs.ino". The code is as follows:

```
1 #include "SoftwareSerial.h"
2 int kounter=0; //Ignore the rest of this line
3
4 void setup()
5 {
6     Serial.begin(9600);
7 }
8
9 void loop()
10 {
11     kounter=kounter+1;
12     digitalWrite(2,HIGH);
13     digitalWrite(3,LOW);
14     delay(1000);
15     digitalWrite(2,LOW);
16     digitalWrite(3,HIGH);
17     delay(1000);
18 }
```

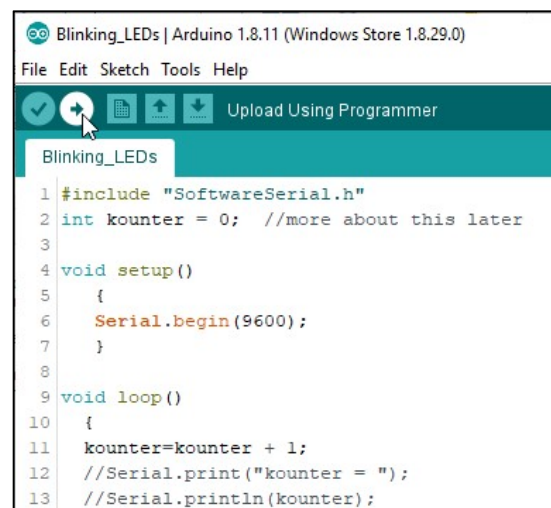
Note that the IDE will ignore any blank line, or any part of a line that follows two consecutive slashes. Thus, lines 3 and 8 will be total ignored, as well as everything after the semi-colon in line 2. Blank lines just make parts of the code easier to separate visually, and anything after the double slashes is treated as comments to the code. I also like to indent the various pieces of the code. All the code you download from me will be structured this way.

But before we try to execute the sketch, we need to make sure two parameters in the IDE are set properly. First, click on the “Tools” tab, then on the “Board” entry, and make sure that the Arduino form factor you are using is highlighted; if not, then click on the entry you need (for me, I am using an Arduino Nano). Then again click on the “Tools” tab and then click on the “Port” entry.



Each Arduino uses a “port” to communicate with your PC, and these may differ from board to board. If multiple entries appear, click on the one with a check next to it. You are now set to compile your sketch.

Now let’s download the code from the IDE into the Arduino. Do this by clicking your mouse on the circle with the green arrow, just below the “Edit” tab. Below the code, in the green bar, the IDE will display “Compiling Sketch”, then “Uploading”, then “Done Uploading”. If instead the IDE displays an orange bar with white text, then you have some coding error, which will be explained in the box below the orange bar. Unfortunately, there is no way I can predict what might have gone wrong, so review the steps, compare the code in your IDE with the example above, and see if you can find the error, then try again. Perhaps you have a colleague nearby who has some Arduino experience who can help.



But if you did everything correctly, the red LED should be blinking on for 1000 milliseconds, and then off for 1000 milliseconds. It will be brightest if viewed from the top, not the sides. But how could this be a model railroad animation? You could drill a hole in the top of a water tank and have a blinking beacon. Instant animation! The possibilities are endless.

Now let's see how we can build on what we have done to make some changes. For starters, let's change the number values in lines 16 and 19. In line 16, change the 1000 to 100, and in line 19, change the 1000 to 1900, and then download the code. We have changed the light from a beacon to more of a strobe light.

Next, let's take another LED/resistor combination, a blue one and wire it to pin D3 and the ground on the other side of the Arduino, (You did use a different resistor value for the Blue LED, didn't you)? Remove the double slashes in front of lines 15 and 18, and then download this modified code. The LEDs should now be blinking alternately, with the blue one on much longer than the red.

```
Blinking_LEDs | Arduino 1.8.11 (Windows Store 1.8.29.0)
File Edit Sketch Tools Help

Blinking_LEDs$
1 #include "SoftwareSerial.h"
2 int kounter = 0; //more about this later
3
4 void setup()
5 {
6   Serial.begin(9600);
7 }
8
9 void loop()
10 {
11   kounter=kounter + 1;
12   //Serial.print("kounter = ");
13   //Serial.println(kounter);
14   digitalWrite(2,HIGH);
15   digitalWrite(3,LOW);
16   delay(100);
17   digitalWrite(2,LOW);
18   digitalWrite(3,HIGH);
19   delay(1900);
20 } // end of main loop
```

If these sketches don't work the way you expected, I have two suggestions for you. When I first started in computing, a wise tutor told me the first three things to do if a computer wasn't working properly: 1) Check the cables; 2) Check the cables again; 3) ask someone else to check the cables. That, and "is your system plugged in" and "have you turned your system off and on again" corrected at least 50% of the issues. Arduino connection points are fragile, so check your connections first when you are trying to figure things out.

But what if we wanted even more LEDs to blink? We could use any of the D2 through D13 pins to drive some more LEDs, but we only have 2 Ground (GND) pins, and it is difficult to get more than a single wire connected to each connection point (and frankly, I like to reserve the ground connection point next to the VIN point for use by an external power supply). This is where barrier strips come into play. Earlier in Part 2 of this series I provided a source on AMAZON where I get these things. Instead of wiring the ground wires from the LEDs into the GND connections, what if we took the cathode wires from the LEDs and wrapped them around a screw of a barrier strip? Then, what if we took a wire from the barrier strip to the GND connection point? Presto! We got all our LEDs connected and used only a

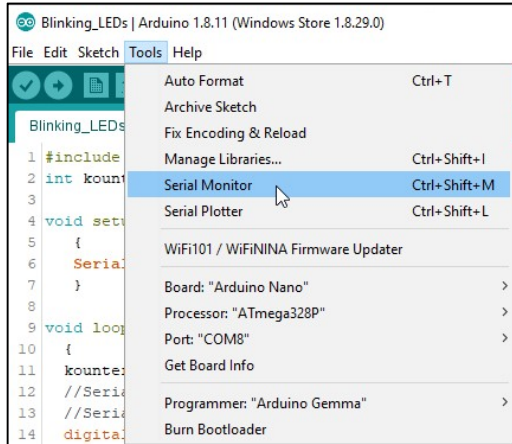


single GND connection on the Arduino. Easily we can get up to 3 wires around each screw, and even more if we solder spade connectors to the ends of the wires before attaching them to the barrier strip. If that's not enough, we can run a wire from one pair of screws to another set and expand to even more LED. I use barrier strips often when connecting one component to another, and particularly when I need to expand the "ground bus".



We have covered a lot of territory already, and if you never intend to create any sketches yourself, then you can stop here. But if you think you might ever want to write some sketches from scratch yourself, please read on a bit more.

I want to cover a little piece about debugging your code. Let's start by deleting the two slashes in front of lines 12 and 13 which will make them active and no longer commentary. Now download the code one more time, but after downloading, click on the "Tools" tab and then on "Serial Monitor". A new window will open, the sketch will



restart, and values for the "kounter" variable will appear, along with the time of day. The ability to check the value of variables while the sketch is executing is immensely valuable in debugging.

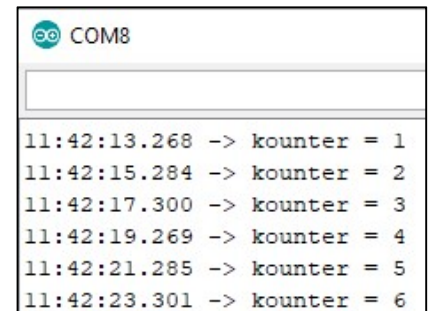
The "Serial.print" statement can print any literal value, like "kounter =", or the value of any

variables, like "kounter". The

only difference between the

"Serial.print" and

"Serial.println" statements is that the latter starts a new output line after its printing. Debugging a sketch would be almost impossible without such diagnostic features.



That about does it for this tutorial on blinking LEDs. I will try to make future parts shorter and build on what we have covered here.