

SUBSYSTEMS

Introduction to Arduino

Table of Contents

Arduino	1
Objectives	2
Tools	2
Arduino Software Load	5
The Integrated Development Environment.....	10
Menu.....	11
File Menu	11
Edit Menu.....	12
Sketch Menu.....	12
Tools Menu	13
Help Menu	14
Quick Access Icons	14
Open Sketch Tabs	15
Code Area	15
Notification Area	15
System Log.....	15
Board Selected	15
Your First (Sample) Program	16
Let's Program	22
Traffic Light Code	25

Serial Output.....	27
Input.....	30
Adding a Switch	33
Using Libraries.....	37
Conclusion	42
Appendix	43

Arduino

Arduino (spoken Ar-dwee-no) is a computer and software company, user community, and open-source project that produce microcontroller kits and assembled units that connect the physical world to a small programmable computer. Arduino was established as a completely open-source project. That



means the hardware and software designs are free to develop, produce, and distribute. This has made Arduino extremely popular in the hobby (frequently referred to as *Maker*) community. The company has created a standard computer and interface, and the community has expanded it by adding software functionality using add on libraries. This means, if you want to add a display to your project, you will find many different kinds, all with a supporting library. So rather

than having to write all code from scratch, you can import the library, hook up your display, and write simple commands like `print("Hello")` to display text. This means that you can get started on complex projects quickly.

In addition to the base computer hardware, the Arduino community has produced many add on boards referred to as “**shields**.” These shields snap in to the pin headers on the computer board itself and route the computer input/output ports to various hardware on the shield.

Arduino is at the heart of many technologies used by hobbyists like mobile robots, 3-D printing, Internet of Things, and many others. As you start to play with this device, you will quickly realize that you are tinkering with an amazingly versatile and powerful device with an immense support community behind it and an internet full of easy weekend projects. So let’s go tinker.

Objectives

When completed with this Subsystem Module, the student will:

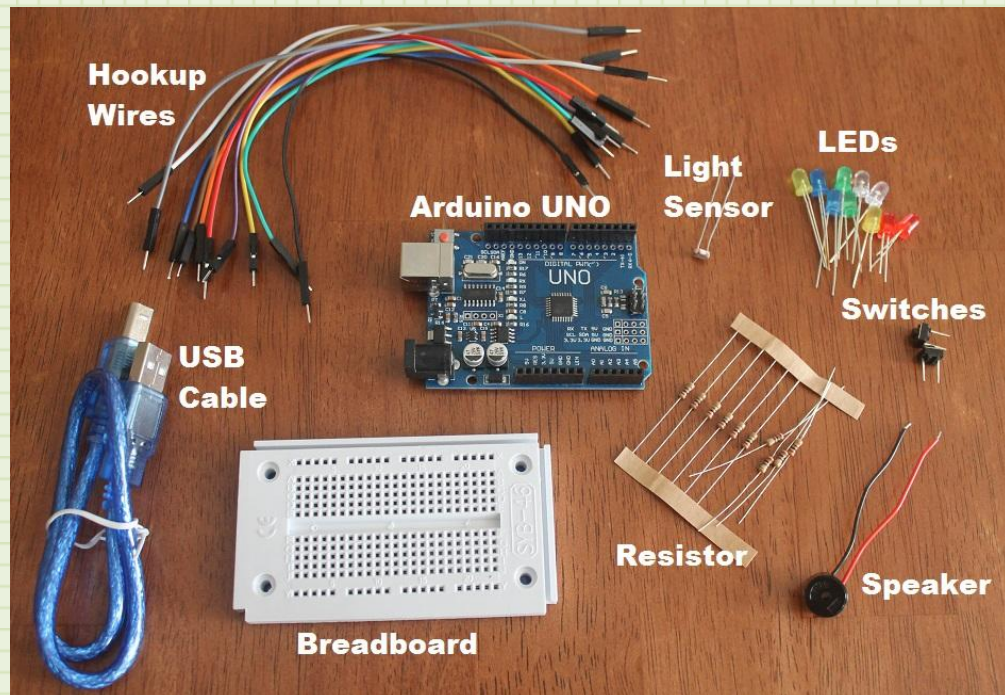
- 1) be able to explain what Arduino is and how it is used.
- 2) describe what is meant by a “shield” with respect to Arduino boards.
- 3) connect an Arduino UNO to a computer and establish a link.
- 4) be able to load a sample program and run it.
- 5) understand the different capabilities of the Arduino Integrated Development Environment (IDE).
- 6) write programs that use input/output on Arduino.
- 7) interface Arduino with external hardware.
- 8) understand the basics of programming in C and C++.
- 9) understand what libraries are and how to use them.

Tools

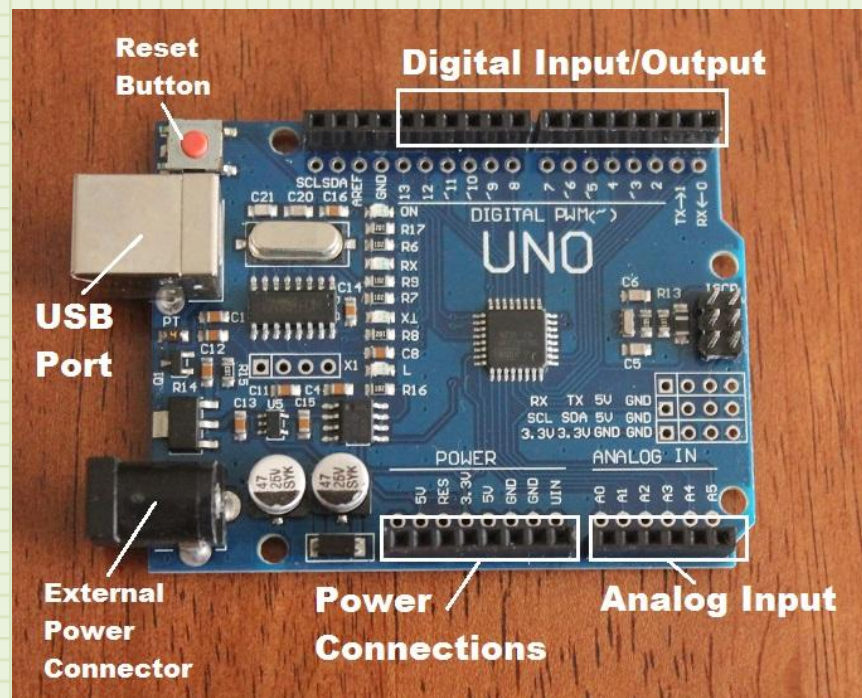
To accomplish these objectives, you have a few tools at your disposal.

First, you have this curriculum guide. It will walk you step by step through the entire curriculum that includes the text, system setup, and programming examples.

You also have **an Arduino UNO** board, USB cable, electronic breadboard, and assorted electronic parts to interface to Arduino.



A close up of the Arduino UNO board shows the connecting headers for the digital and analog input and output.



The UNO can be powered from the USB cable or from an external supply. We will be using the USB connection for both power and for uploading programs to the board.

Let's start our dive into Arduino by loading the Arduino software.

Section 1

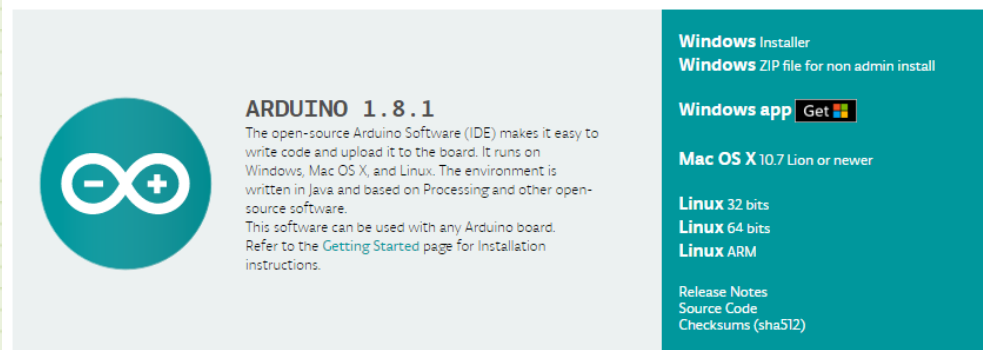
Arduino Software Load

We will start our setup with loading the Arduino software and making sure it is configured properly. If you already have a current copy of Arduino on your computer, you can skip this section.

Visit www.arduino.cc and download the latest version of the software for your particular system. Go to the main site and click on the menu item “Software” on the top navigation banner.

Scroll down until you find the link to download the IDE (integrated development environment).

Download the Arduino IDE

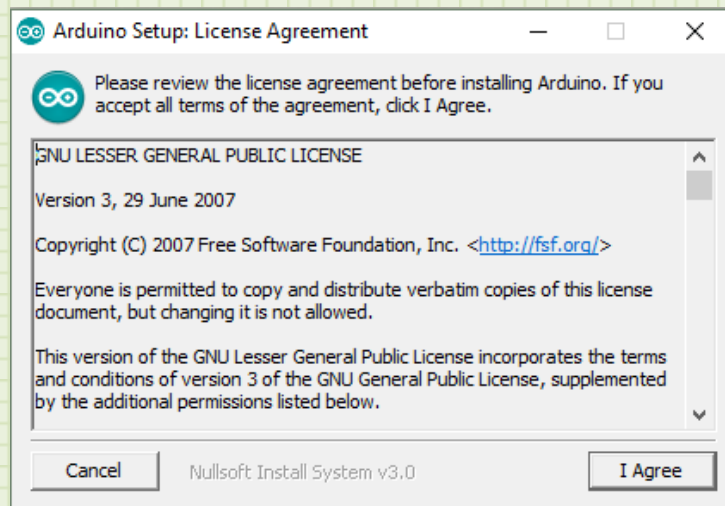


The screenshot shows the Arduino IDE download page. On the left, there is a large teal circle containing the Arduino logo (an infinity symbol with a minus and plus sign). To the right of the logo, the text reads: **ARDUINO 1.8.1**, followed by a paragraph describing the IDE as open-source software that runs on Windows, Mac OS X, and Linux. On the right side of the page, there is a teal sidebar with links for different operating systems: **Windows Installer**, **Windows ZIP file for non admin install**, **Windows app** (with a 'Get' button), **Mac OS X 10.7 Lion or newer**, **Linux 32 bits**, **Linux 64 bits**, **Linux ARM**, and links for **Release Notes**, **Source Code**, and **Checksums (sha512)**.

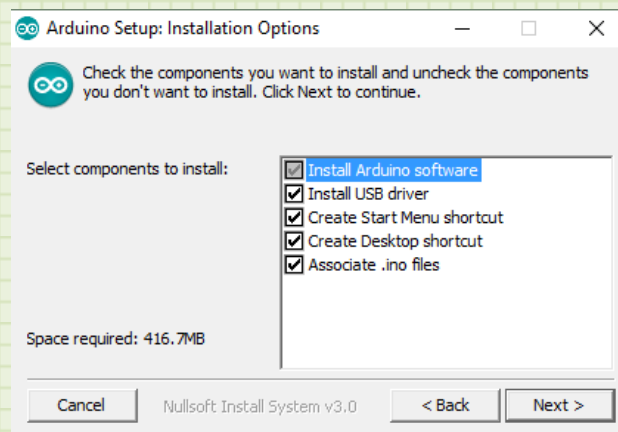
On the right, is a list of the different operating systems. Click on the one that corresponds to the one you use. Opt for the full version and not the app if it is offered.

Download this file to a place on your computer where you can find it. Windows will normally put downloads in the “Downloads” folder.

Now, run that installation program. You will be greeted with a user agreement.

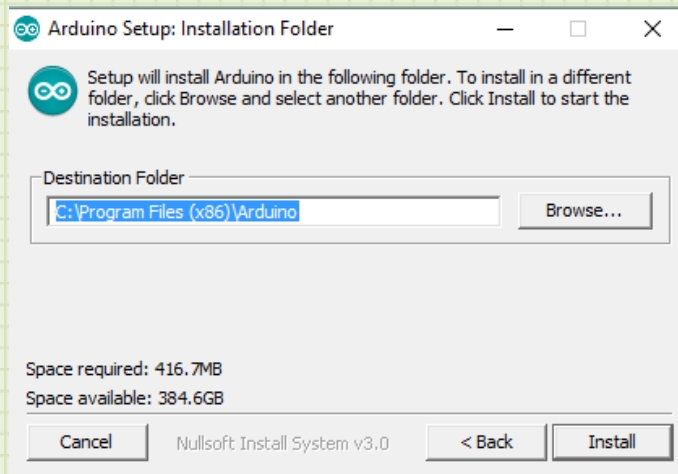


If you agree with the license, click "I agree" to move on to an "Installation Options" dialog.



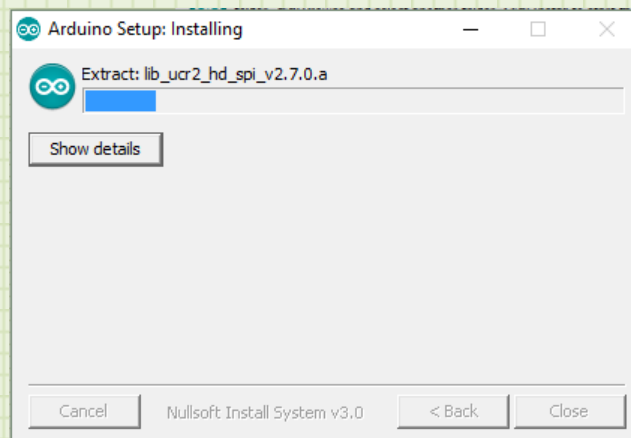
The default values are good for our purposes so click "Next."

You will then be presented with a dialog that has you choose the installation location.

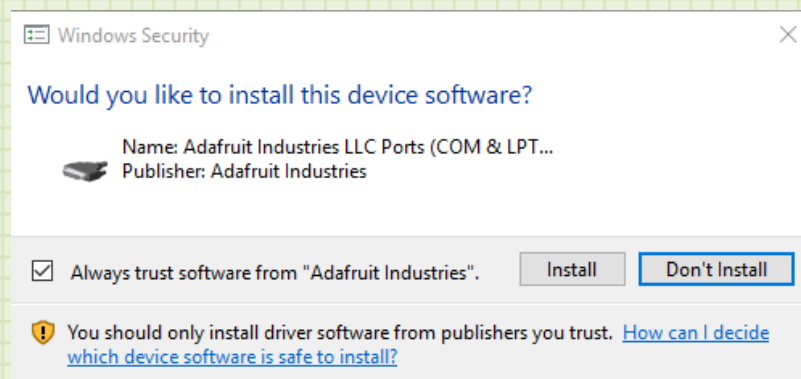


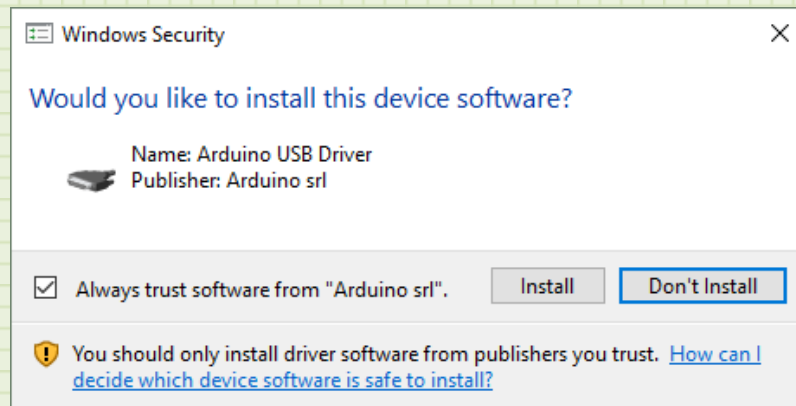
And again, the default value works great for us so just click “Install.”

The installation software will start loading the program and all the support files.



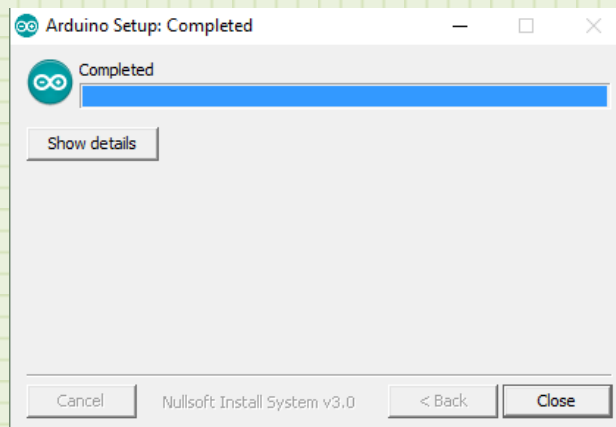
This installation will take a little bit to complete. At some time, you may be presented with driver installation warnings like the following:





These are trusted, well know sites that develop software for the hobby community so click “Install” to install these drivers. These will allow us to communicate over the USB port with the UNO.

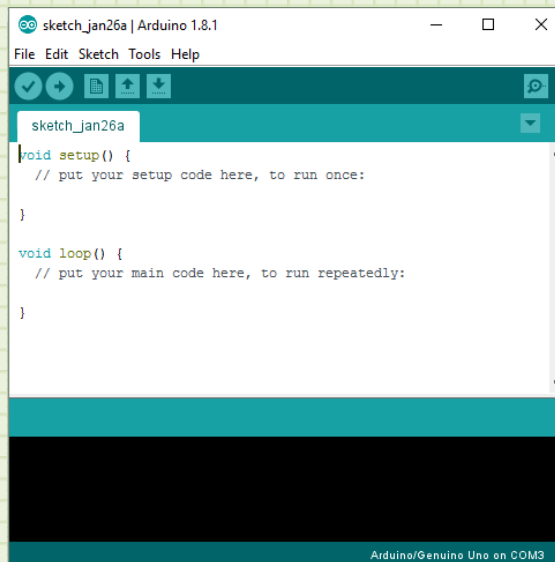
When the installation is complete, you will be presented with a final dialog.



Click the “Close” button.

If all, went well, the Arduino software and drivers are loaded on your computer. Let’s open up that software and finish setting it up.

Find the Arduino program on your computer (from the Start icon in Windows or the Launchpad on a Mac) and run it. You should see the following:



If you are presented with any other dialogs that show options for downloading other components, you can either accept them or cancel them. Your base software has everything we need.

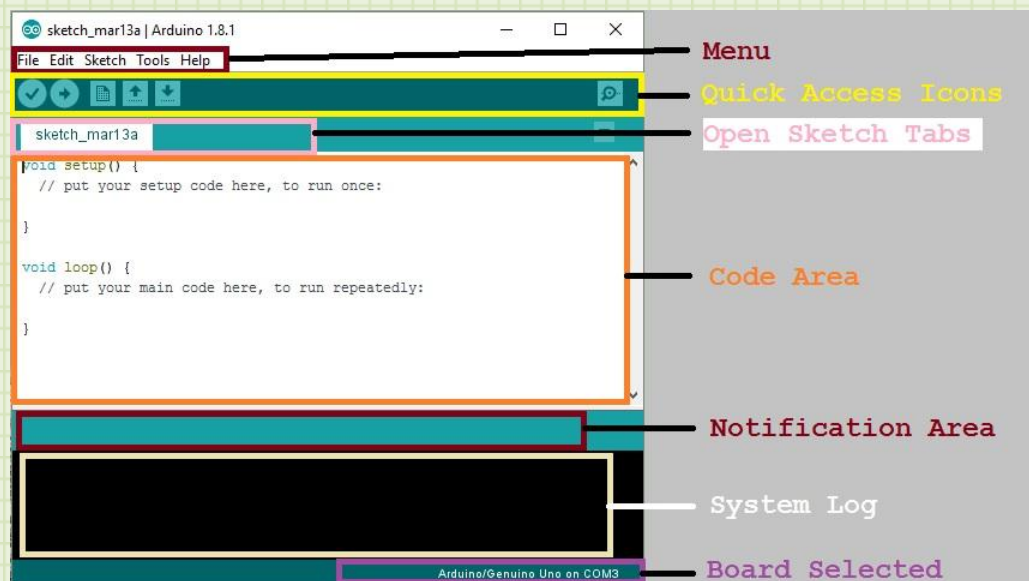
Section 2

The Integrated Development Environment

Software development has changed since the dawn of computers. Initially, computers were programmed memory location by memory location using a series of switches and buttons. Then, more powerful computers and better hardware interfaces allowed us to program the computer using the computer. As computer complexity grew and more and more systems had computers embedded in them, computers were then used to write programs that were for other computer systems. This is called **cross-compiling**. Let's say you wanted to write an app that runs on an Android phone. But you want to write it on your Windows based computer. You could write the code in a text editor, and then send it to a small computer program (called a compiler) to convert it to the Android system language. The resulting file couldn't be run on your Windows based computer, but could be transferred to the Android phone and then run. This also opens us up to the ability to reuse code. There are some computer programming languages (like Java) that can be written once, and then compiled for many different platforms. The compiler does the work of taking the human readable program and converting it to the machine specific code to run on that platform.

But one of the biggest needs in programming is the ability to find problems with your completed software. This is called troubleshooting (or de-bugging). Wouldn't it be great if there were a way to test the program out on the machine you were writing the code on? Well, you can. A small program called an **emulator** can act like the device you are developing for and run your program locally. That means you can write your program, compile it, and run it on the Windows laptop, even though it is going to be loaded on an Android phone. Coordinating all this interaction, code de-bugging, and program monitoring is a

piece of software called and **Integrated Development Environment (IDE)**. An IDE dramatically simplifies the task of writing computer software. Arduino is programmed in an IDE. You can use your Windows, Apple, or Linux computer to write programs and then compile them for Arduino. You can use the same software to download the compiled programs to the Arduino hardware and can even use it to monitor the hardware while it is running. Let's take a tour of the Arduino IDE.



Above shows a breakdown of the different areas of the IDE. Let's look at each area.

Menu

The menu is the standard place for you to access all the functionality of the program. Note: Arduino refers to a program written in the IDE as a **sketch**.

File Menu

New – creates a new Arduino sketch.

Open... - allows you to open a saved sketch.

Open Recent – this gives a convenient listing of the most recent sketches for you to select.

Sketchbook – This is a listing of all the sketches you have already saved.

Examples – this gives you access to the examples that come with the Arduino software as well as others that come with additional libraries you load.

Close – this closes the current sketch. It will prompt you to save if you have not.

Save – this will save the current sketch under the previously saved name. If it is a new sketch, you will be prompted to give it a name.

Save As – this allows you to save the current sketch as a new name.

Page Setup – this allows you to change the page characteristics. This is important if you will be printing your sketch.

Print – allows you to print the current sketch.

Preferences – this opens a dialog that allows you to modify preferences and other setup related information.

Quit – this exits the program. You will be prompted to save your work if you have not.

Edit Menu

The Edit menu selection has all the normal editing functions like cut, copy, paste, undo, etc. Of note are the two copy options (**Copy for Forum**, **Copy as HTML**). Since Arduino has such a large following, you will find many times you may want to post your code to a forum or on a webpage. These two options automatically format the code so it will have a standard appearance when pasted into these destinations.

Sketch Menu

Verify/Compile – this selection compiles your sketch into a form ready for download to your Arduino hardware. You will see progress in the Notification Area and the System Log area. This is also where you will see if the compile was successful or if there were errors.

Upload – this will take a compiled sketch and upload it to connected Arduino hardware. If the software needs to compile the program first, it will do that.

Upload Using Programmer – this will upload a sketch to the Arduino hardware and overwrite the onboard software that communicates with the IDE. This is an advanced option for programmers who need to use the entire memory of the Arduino and will not need to communicate with the IDE anymore. DO NOT use this option to download programs.

Export compiled Binary – this will save the compiled program so you can have a local copy. This is useful if you want to back up the code or want to use another programmer to load it onto an Arduino.

Show Sketch Folder – this opens the file explorer on the folder where the current sketch is located.

Include Library – this allows you to select a library to add to your code. It will automatically add the correct “include” statement.

Add File... – this adds an existing file to the current sketch and opens it in another tab.

Tools Menu

Auto Format – this puts your code in a standard, easy to read, format.

Archive Sketch - saves a copy of the sketch in .zip format.

Fix Encoding & Reload - fixes problems between the editor and other operating systems character maps when you copy or load sketches from other sources.

Serial Monitor - opens the serial monitor window for the currently selected Port. This allows the Arduino board to receive and send data in this window. This may reset the board (it will for your UNO board).

Board – this allows you to select the current board. The compiler needs to know the type of board to assign correct values to things like pins, ports, memories, and clocks.

Port - this contains all the serial devices available on your machine. It should automatically refresh every time you open the top-level tools menu.

Programmer – use this if you are using some other programmer than the built in one that comes with Arduino (we will be using the built in one).

Burn Bootloader – a bootloader is a small program that lets the Arduino communicate with your computer. This is what allows us to program it directly from the IDE. If you are using a brand-new chip or if your chip gets corrupt, you may need to download the bootloader to the chip again. This selection allows you to do that.







Help Menu

Here you will find a lot of useful resources. Many of these link back to the Arduino page for more information. The Find in Reference selection is context sensitive and will use your cursor to give you help based on the command present at the cursor.

Quick Access Icons



These are readily available icons that represent frequently used menu items. They are as follows:

	Verify/Compile
	Upload
	New Sketch
	Open
	Save
	Serial Monitor

Open Sketch Tabs

These are tabs that select each of the sketches you have loaded. For most applications, you will just have one tab for the current sketch you are working on. As you use Arduino more and more, you will find that there may be standard routines that all your applications are using. You could save them in a separate sketch and open them in your current sketch as a separate tab. This will give you access to all the enclosed software routines.

Code Area

This is where you will compose your code. This area has some great editor features. It allows the normal cut and paste functions. But it also will color code the lines depending on recognized functions and commands. This can be helpful and makes the code more readable.

Notification Area

This space is used to convey the status of certain operations like save, compile, and upload. It will also turn red when there is an error that requires attention.

System Log

The IDE calls different compilers and other helper applications during compiling and uploading. This area is used to display the information returned from those applications. Sometimes there will be extra information on errors that occur. After compiling, this area will show the amount of memory used and the percent of available memory for the particular device.

Board Selected

This area shows the currently selected board (set by Tools->Board:). This should match the board you are working with. If not, select it from the menu.

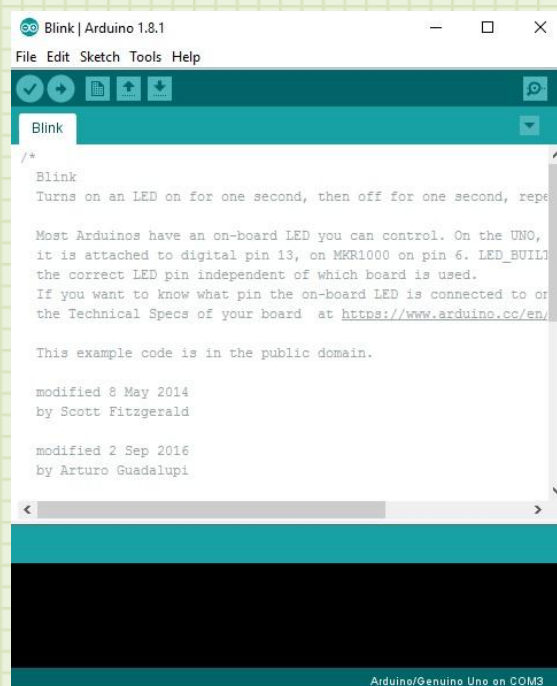
Section 3

Your First (Sample) Program

Now that you understand the basic layout of the IDE, let's start using some of the rich set of features it provides.

The easiest way to get started is with one of the included sample programs. From the menu bar select:

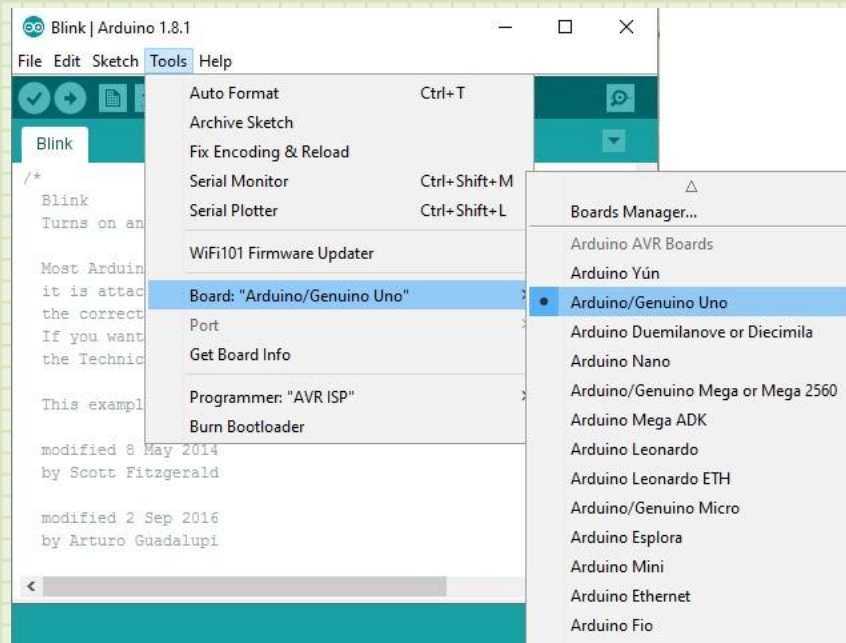
File->Examples->01.Basics->Blink



When you open the sketch, the code will appear in the code window. The Blink example will just load the code to blink the LED on the Arduino board. Let's hook up the board and get it ready to receive code. First, let's make sure that the IDE

is set to the correct board. The board included in your kit is an Arduino UNO. From the menu, select:

Tools->Board:->Arduino/Genuino UNO

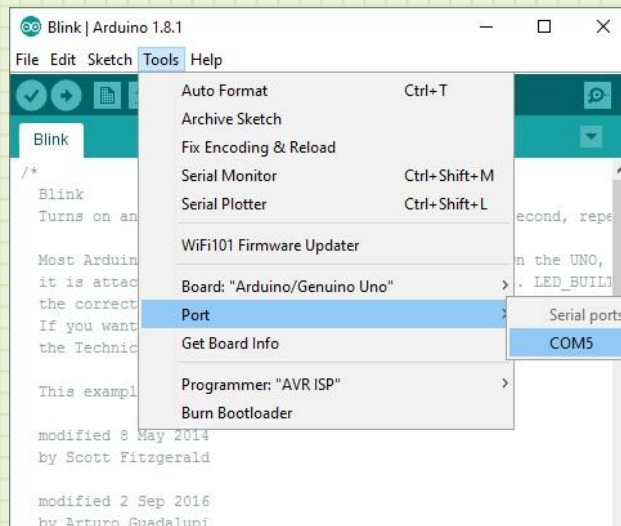


You can see from the board menu that Arduino has many variants. As you experiment with this computer system, you may find that want more inputs and outputs or you want to have a very small computer board that you can wear. Arduino and many other suppliers make many different boards that fill a variety of needs. But they can all be programmed with the Arduino IDE.

Plug the Arduino UNO into a USB port on your computer using the enclosed USB cable. The power light on the UNO will come on and your computer will register the new device. Now, let's tell the IDE what port our UNO is plugged into.


Select **Tools->Port->COM...**

When you view the selections, there should be a port labeled COM followed by a number. This is the reference to a serial port. Serial communication is when you send data one bit at a time over the transmission line.



In the above example, the UNO shows up on COM5. Whatever COM port is present, select it. If there are multiple COM ports, you could either select the different ports and attempt to download the program (described below) until you find the right port. Or you could go into your computer's hardware section and see which COM port the UNO is hooked up to. For most systems, there will most likely only be one COM port.

With the IDE selected to the correct board and the communication ready, we can now upload the Blink program to our UNO.

Select **Sketch->Upload** or click the Upload icon. 

You will see a message in the notification area showing that the upload has begun as well as a progress bar. After a short time, you will see that the upload is complete and if you look at your UNO, you will see the red LED flashing slowly. Congratulations. You have compiled and downloaded your first Arduino program.

Let's look at the program to start learning how to write code. The first thing you notice is a lot of comment text. Programmers often want to place descriptions and instruction and even copyright notices within the program. This is good practice. You insert comments by using a `/**` on the line right before the comment.

```

// the setup function runs once when you press reset or power the board

void setup() {

  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);

}

// the loop function runs over and over again forever

void loop() {

  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)

  delay(1000);           // wait for a second

  digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage
LOW
  delay(1000);           // wait for a second
}

```

This first thing to observe is that all Arduino programs have two main functions.

void setup() – this is where you place code that you want to execute once when the program first starts. This is where you will place your initialization code and where you will setup your hardware and start certain processes.

void loop() – this is where you place code that will continually repeat. This is normally where the bulk of your coding takes place. Code is sequentially executed and when it gets to the end, it recycles back to start executing code from the top of this function again.

The term **void** in front of the function names is a return type definition. When functions are executed, they run a specific set of code and then return a result. This result may be the value of a light sensor, the result of a math equation or the name of a person in your phone directory. It is some data that is being

returned to the original program. When we declare functions, we must tell the software what type of data is being returned. Sometimes we return an integer number, sometimes a character, sometimes a Boolean value (True/False). If the function will not return anything, we declare it as void. These two functions do not return anything so they are declared with the void keyword.

The first line in the setup() function is:

```
pinMode(LED_BUILTIN, OUTPUT);
```

pinMode is a function that allows us to tell the Arduino how we want to use a specific pin on the computer. The UNO has 13 general purpose input/output pins. Before we use them, we must define how we will use them. The choices are:

- **INPUT** – the pin will be used to send data TO the computer.
- **OUTPUT** – the pin will be used to send data FROM the computer.
- **INPUT_PULLUP** – the pin will be used as an INPUT, and the internal resistor connected to the pin will be enabled. This is used sometimes to ensure a digital input is either high or low. Without the input resistor, the input port voltage might just wonder around if no input is present and give a non-reliable reading of the input. By using an internal resistor tied to the positive supply, the port will go high with no input present. This puts the port in a known condition when there is no input present.

The **pinMode** function takes 2 arguments. The first is the desired pin. In the sample program, this is **LED_BUILTIN** which the software already knows is assigned to pin 13 on the UNO. The second is the actual mode (**OUTPUT** for the sample program). Since we use the pin to light an LED, we are sending the signal from the computer to the LED so we need to establish it as an OUTPUT.

That is the only code required for the setup of our program. Now let's look at the loop function.

```
digitalWrite(LED_BUILTIN, HIGH);
```

The first command that is executed in the loop is a digitalWrite command. This command sends a High or Low signal out on the desired pin. The function takes two arguments. The first is the desired pin (LED_BUILTIN for our example). The

second is the state we want the pin in (HIGH in this example). When you write a HIGH to the LED_BUILTIN pin (pin 13 on UNO), the LED on the board turns on.

```
delay(1000);
```

The next command is a delay function. This function takes one argument which is the number of milliseconds to pause program execution. In our example, it is 1000 milliseconds or 1 sec. So far in the loop we have turned on the LED and waited 1 second.

```
digitalWrite(LED_BUILTIN, LOW);
```

The next command is another digitalWrite. This time, it sets the LED port to LOW. This turns off the LED.

```
delay(1000);
```

The last command in the loop is another 1 second delay.

So in all, this bit of code turns on the LED, waits 1 second, turns off the LED, and waits one second. This has caused our LED to flash. Now that we reach the end of the loop, the loop is executed again and the LED flashes again. This loop continued indefinitely so the LED keeps flashing.

This was a simple example, but it incorporated a lot of the elements essential for writing your own programs. You got the IDE loaded and configured. You opened a sketch and downloaded it to the UNO. And you know the basics of outputting a HIGH or LOW on a port. You also understand the structure of a sketch and how the setup function is executed once and the loop function just continually repeats.

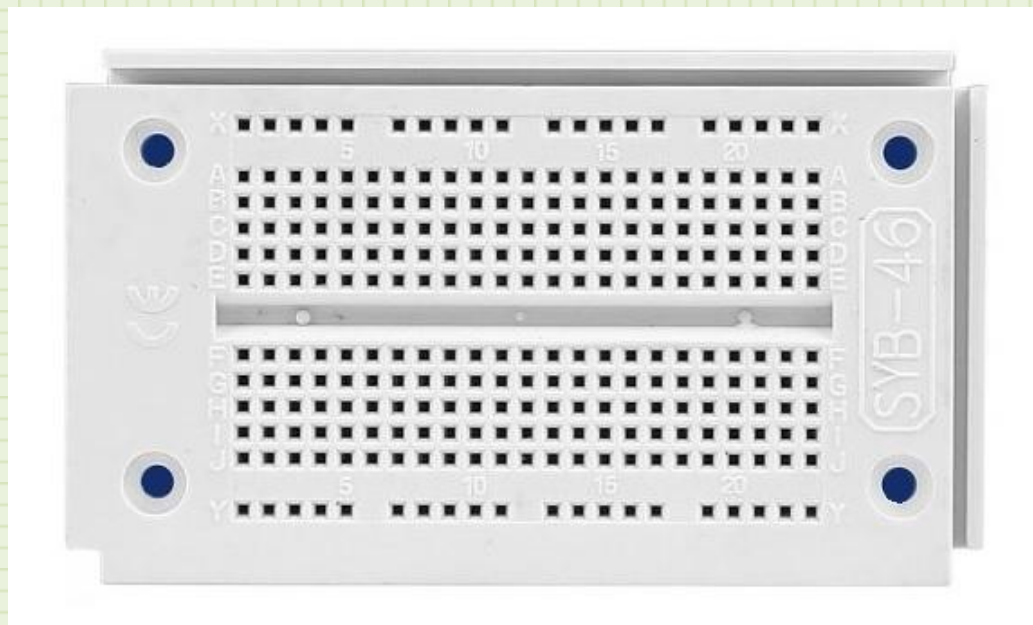
Now you're ready to author your own sketches. Let's play with some more LEDs and make ourselves a traffic light.

Section 4

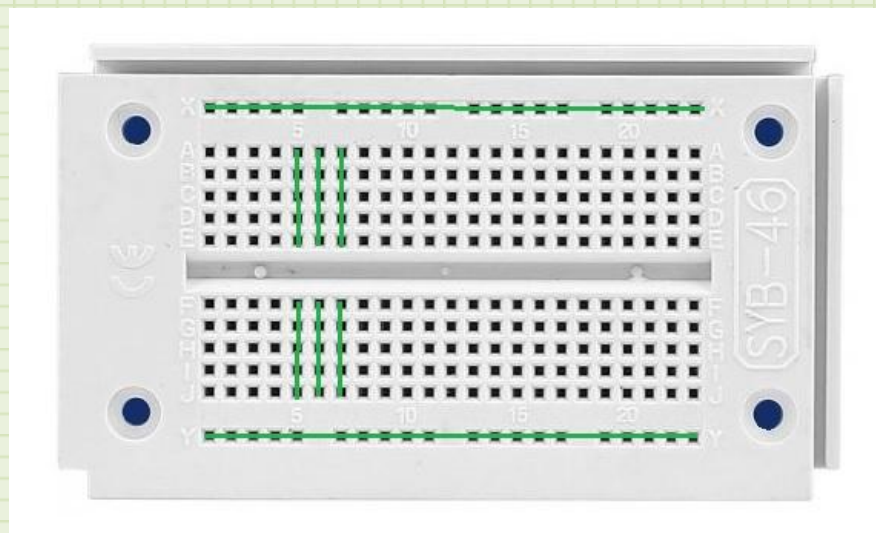
Let's Program

Constructing a traffic light will be a variation on the basic Blink program. We will set up three outputs to drive LEDs, turn on and off the LEDs, and add delays to mimic a traffic light operating. First, let's learn how to use a powerful prototyping tool; the **breadboard**.

A breadboard is an electronic prototyping device that allows you to quickly assemble circuits. After you are done, you can remove all the components and reuse the breadboard for another circuit. It gets its name from the early days of electronics where people would hammer nails into a literal wood bread board and wrap components and wires around the nails to create circuits. Below is a picture of the breadboard included with this module.



The breadboard is laid out with a row of holes on the top and bottom. Each row of holes is electrically connected. That means, whatever we plug into those holes will be connected to anything else in the other holes in the row. The upper row is marked with an 'X' and the lower is marked with a 'Y'. This is a convenient place to hook up the positive and negative supply voltages to a circuit so that we can easily access it anywhere on the board. We will normally hook up the breadboard in the orientation you see in the picture. The 'X' supply line on the top and the 'Y' supply line on the bottom. The columns labeled with numbers on every 5th column in the middle of the board are also electrically connected. That connection is broken in the very center of the board by that small valley in between the rows marked by the letters E and F to allow mounting of electronic integrated circuits. The green lines on the diagram below show you a sample of the places that are connected internally inside the board. It shows the top and bottom and a few of the columns (col 5, 6, and 7). Remember that the columns are not connected across the gap in the middle.



The holes themselves have small metal receptacles that grab on to leads of components and wires that we stick in them. By placing the components in specific places and using the conductive paths of the breadboard, we can create complete circuits quickly. As a note, new boards tend to be a little finicky. Sometimes it is hard to get wires to go in and sometimes when they are in, they can make poor contact. With use, most boards loosen up and become more easy to use. If you are having problems with certain circuits, check the connections and consider moving them to a different rows on the board.

Let's hook up the breadboard for our traffic light project.

From the kit of parts, take out 3 resistors, a Red LED, Green LED, and Yellow LED, and 4 hookup wires (Red, Yellow, Green, and Black)

Bend the leads of the resistor down so they can be inserted into the breadboard. Using the letter and numbers on the board, hook up the resistors between the following points:

Resistor	Lead 1	Lead2
1	7C	7H
2	9C	9H
3	11C	11H

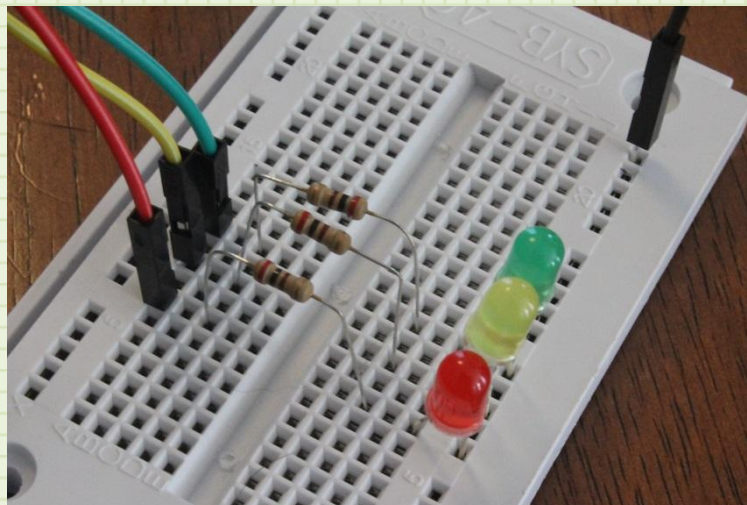
Hook up the LEDs to the breadboard into the following holes:

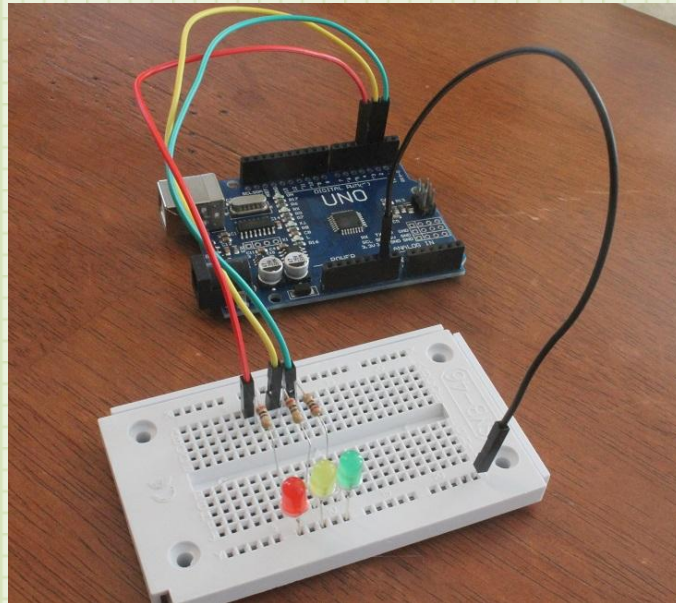
LED	+Lead (longer)	-Lead (shorter)
Red	7J	7Y
Yellow	9J	9Y
Green	11J	11Y

Keep in mind that LEDs are polarity sensitive. The longer lead is the positive lead. Finally, connect the hookup wires as follows:

Wire	Breadboard	Arduino
Red	7A	Pin 4
Yellow	9A	Pin 3
Green	11A	Pin 2
Black	23Y	GND

Your setup should look like the picture below:





With our LEDs hooked up to the Arduino, it is time to start writing some code.

Traffic Light Code

Open the Arduino IDE. The IDE opens up to the last Sketch you were working on. Select:

File->New

This will open a brand new sketch.

For our code, we want to program the Arduino to use 3 outputs to control the three LEDs. We are not adding any new commands then you have already seen in the sample program.

First, we need to tell the Arduino that we want to use pins 2, 3, and 4 as the LED outputs. In the setup() function, add the following lines of code:

```
pinMode(2, OUTPUT);  
pinMode(3, OUTPUT);  
pinMode(4, OUTPUT);
```

As before, this tells the UNO that pins 2, 3, and 4 will be used to output a digital signal. When a computer first starts, it's outputs and inputs may be in an unknown condition. To ensure you get the response you are looking for, it is good to place them in a known condition. We will do this by telling the UNO to

send the LED ports to the LOW condition. LOW is zero volts, so our LEDs will not light. Add the following code to the setup() function:

```
digitalWrite(2, LOW);  
digitalWrite(3, LOW);  
digitalWrite(4, LOW);
```

This should be all of the setup code we need. Now let's go into the sequencing of our traffic light. We want it to simulate a real traffic light so we would expect it is green for a while, turns yellow briefly, and then turns red for a while. After that, the cycle repeats. Since our loop() function repeats, we can just setup the code to cycle the lights once, and let the loop() function continually repeat the sequence.

Let's add the Green light code to the loop() function:

```
digitalWrite(2, HIGH);  
delay(6000);  
digitalWrite(2, LOW);
```

Looking at the code, the first line turns on the Green LED (it is attached to pin 2). The next line waits 6000 ms (6 seconds). The last line turns off the LED. We will do the same sequence for the yellow and the red, but we will delay the yellow only one second. Add the following code after this code in the loop() function:

```
digitalWrite(3, HIGH);  
delay(1000);  
digitalWrite(3, LOW);  
  
digitalWrite(4, HIGH);  
delay(6000);  
digitalWrite(4, LOW);
```

When you are all finished, the completed code should look like this:

```
void setup() {  
  // put your setup code here, to run once:  
  pinMode(2, OUTPUT);  
  pinMode(3, OUTPUT);  
  pinMode(4, OUTPUT);
```

```

digitalWrite(2, LOW);
digitalWrite(3, LOW);
digitalWrite(4, LOW);
}

void loop() {
  // put your main code here, to run repeatedly:
  digitalWrite(2, HIGH);
  delay(6000);
  digitalWrite(2, LOW);

  digitalWrite(3, HIGH);
  delay(1000);
  digitalWrite(3, LOW);

  digitalWrite(4, HIGH);
  delay(6000);
  digitalWrite(4, LOW);
}

```

Plug your UNO into your computer's USB port. Take this opportunity to save your sketch. Select **File->Save** and then give the sketch a name (like *trafficLight*). Make sure your UNO is set to the correct serial port as before. Now upload the sketch (Sketch->Upload or click the upload button).

After the sketch uploads to the UNO, you will see your traffic light operating. If there is a problem, check your breadboard hookup. Sometimes it is easy to put LEDs in the wrong direction and they will not light.

Congratulations! You have started down the road of Arduino programming. So far, you have loaded and configured the IDE, hooked up an external circuit, started a new sketch, configured ports on the UNO, wrote code to change the condition of the ports, added delays, and compiled and uploaded the sketch. That is quite a lot. But there is so much more.

Serial Output

One of the most convenient aspects of the UNO is that the same port that we use to upload sketches can also be used to download data. This is a great way to

pass status, text results, and sensor data back to your computer. Let's modify our traffic light program to include sending the state of the traffic light to the serial output.

Serial communication is a complicated transmission between computers. Luckily, all of the code to accomplish this has already been figured out and is immediately available to you in the Serial library included in the IDE. A library is a set of code files that provide some kind of functionality to a program. By containing setup, configuration, and communication functions in a library, we can easily reuse code, and we can keep our sketches easier to read and troubleshoot. There are many libraries included with the IDE. But, you can also import external libraries and even write your own. Right now, we will use the built-in Serial library.

In our traffic light sketch, we need to initialize our Serial library. Place the following code in the setup() function:

```
Serial.begin(9600);
```

This line of code tells the UNO to set up the Serial port at a speed of 9600 baud (this is a measure of how many bits of data are transmitted per second).

With the serial port configured, we can now just print to it as if it were a display. Let's put a serial print statement after each time an LED is lit. Below is the whole sketch with the new lines highlighted in yellow. We actually use the println() command because it adds a new line character to the output so it will print the next output on a new line.

```
void setup() {  
  // put your setup code here, to run once:  
  pinMode(2, OUTPUT);  
  pinMode(3, OUTPUT);  
  pinMode(4, OUTPUT);  
  
  digitalWrite(2, LOW);  
  digitalWrite(3, LOW);  
  digitalWrite(4, LOW);  
  
  Serial.begin(9600);  
}
```

```

void loop() {
  // put your main code here, to run repeatedly:
  digitalWrite(2, HIGH);
  Serial.println("Green LED");
  delay(6000);
  digitalWrite(2, LOW);

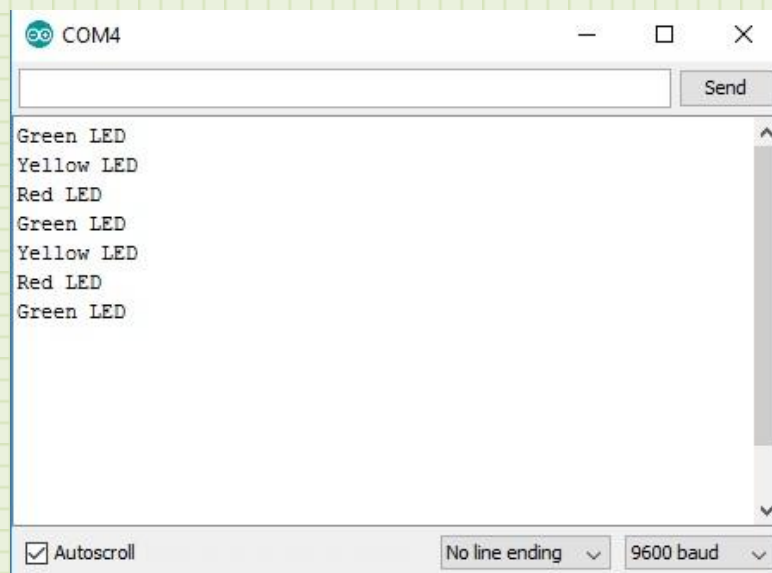
  digitalWrite(3, HIGH);
  Serial.println("Yellow LED");
  delay(1000);
  digitalWrite(3, LOW);

  digitalWrite(4, HIGH);
  Serial.println("Red LED");
  delay(6000);
  digitalWrite(4, LOW);
}

```

The `println()` function will send whatever text you place in the quotes out on the serial port. Save this sketch (clicking save now will just save it to the same file name) and upload this sketch to the UNO as before. You should see the traffic light sketch control your LEDs as usual. Now, select **Tools->Serial Monitor**.

In the bottom right, make sure the data rate is selected to 9600 baud. You will now see the text for each LED that lights.



Input

We have seen how to use the UNO to output high and low voltages. Let's see how we get input into our UNO.

Let's use the UNO to read a light sensor value and report it to the Serial port. Then we will use it to set up a warning circuit if the light level gets too high.

Set up your breadboard with the following connections:

Component	Lead 1	Lead 2
Red LED	20J (positive lead)	20Y (negative lead)
Resistor	20H	20C
Light Sensor	5J	5Y

And connect wires to the UNO as follows:

Wire	Breadboard	Arduino
Black	23Y	GND
Yellow	20A	Pin 2
Green	5H	Pin A0

The pins A0 thru A5 on the UNO can be used as basic input and output pins. But they have an alternate function that will take a voltage between 0 and 5V on the pin and convert it to a digital number that can be used in the sketch. We will display this number on the Serial output.

With the hardware all set up, let's turn our attention to code. Start a new sketch in the IDE.

Let's add the following code to the setup() function:

```
Serial.begin(9600);

pinMode(A0, INPUT_PULLUP);
pinMode(2, OUTPUT);

digitalWrite(2, LOW);
```

Here, we initialize the Serial port as before. In the next line, we set up pin A0 to be an input. To access this, instead of using the `digitalWrite()` command we will use the `analogRead()` command. We use the `INPUT_PULLUP` setting to switch in an internal resistor tied to the 5V supply. The rest sets up the LED pin and makes sure it is outputting a LOW to ensure the LED is off.

With the setup complete, let's instruct the UNO to read the value of the analog input on A0 and output it to the Serial port.

Add the following to the `loop()` function:

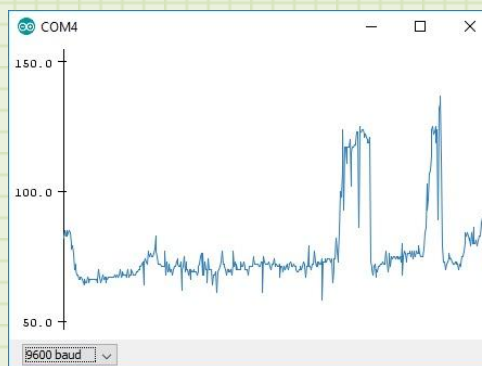
```
Serial.println(analogRead(A0));  
delay(250);
```

The first line reads the value of the analog port (`analogRead(A0)`) and prints it out on the Serial port. We sent text information before which needed to be enclosed in quotation marks. Here, we are just sending raw number so no quotes are needed. We add a small delay so the output is easier to see.

Save this sketch (name it anything you like). Plug the UNO into your USB port and upload this sketch. When complete, open the Serial monitor. You will see the values of this voltage display on the screen. Move your hand to cover and uncover the light sensor. You will see a corresponding change in the values. An even better way to view this is to see if graphed. Close the Serial monitor and select:

Tools->Serial Plotter

Initially, the graph will hunt a bit to adjust to the range of data. After just a few seconds, you will have a stable scale and will more easily see how covering and uncovering the light sensor causes a change in the output.



You can see from the graph that the output is low until you cover it up and then the output goes higher. Let's use that to come up with a sensor that gives a warning when the light gets too dim.

We have the red LED hooked up for just that reason. Let's add a conditional statement to test if we reach a threshold and then light the LED if it is met. From the graph above for our system, it looks like 100 might be a good level since it is toward the middle of the high and low values. Look at your output and pick a good value (the ambient light will change this value greatly so yours may be much different).

The conditional statement we will use is an **if..then...else** statement.

Add the following code after the `Serial.print()` in the `loop()` function:

```
if (analogRead(A0) > 100) {  
    digitalWrite(2, HIGH);  
}  
else {  
    digitalWrite(2, LOW);  
}
```

The curly brackets always enclose groups of commands that will be run as a set. You will use them in `if...then...else` statements if you have multiple commands to run. What the above code does is compare the analog A0 channel with 100. If it is greater than 100 it executes the code in curly brackets that follow. In our case, light the Red LED. The else statement says, if the above is not true then execute the commands in the curly brackets that follow the else keyword. In our case, it will turn off the Red LED.

It is very important to note the difference in using the equal sign.

If I want to set a variable in my program to a certain number I use a single equal sign (like `A = 5`). If I want to compare two things to see if they are equal, I use a double equal sign. If I write an *if* statement to check if A is equal to 5 it would look like the following:

```
If (A==5) {do something}
```

This is important because if we use a single equal sign in the above expression, the computer would first assign the value 5 to our variable A. Since this would be

successful, the if statement would evaluate the action as a true statement and would execute the code that followed. This would not give you the desired result so keep that in mind when you logically test values.

Upload your code to the UNO. You should see the LED off initially. But when you cover the light sensor, the LED will turn on. If it is not happening, take a look at the Serial Plotter again and see if your numbers have changed and the compare level needs to be adjusted.

Since the ambient light might change through the course of the day, it would be good to program in a way to set this threshold value for our LED to light. Let's do that with a switch.

Adding a Switch

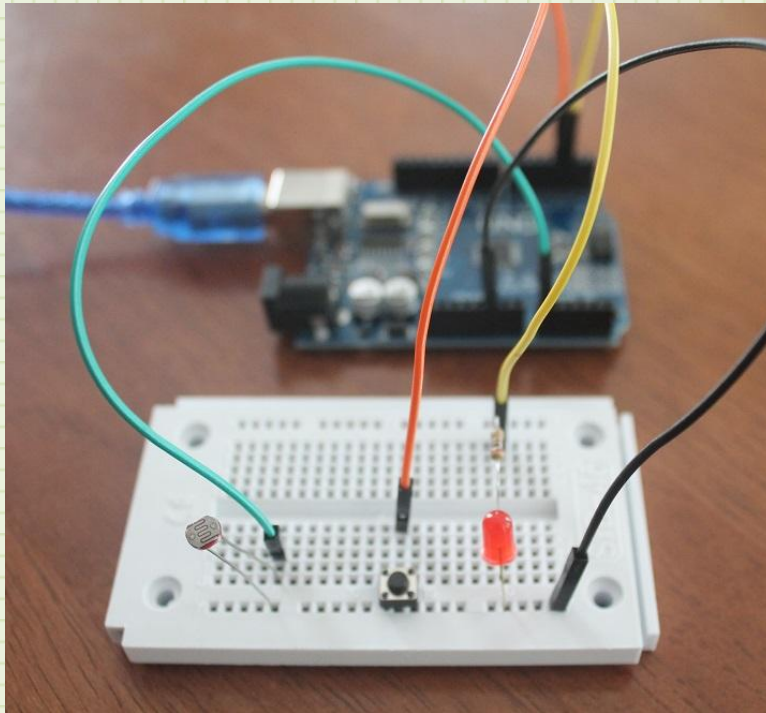
A switch is a way to turn on and off something. In this case, we will use it to create an input signal to an UNO port that will read HIGH when the switch is not pressed, and LOW when it is pressed. To the previous circuit, add the following switch connections on the breadboard:

Component	Lead 1	Lead 2
Switch	13J	13Y

And hook to the UNO as follows:

Component	Breadboard	Arduino
1	12A	Pin 3

Your setup should look similar to the following:



Now we just need to add some code to let the UNO know how to handle this input. First, let's add a line in the `setup()` function to let the UNO know that this will be an input:

```
pinMode(3, INPUT_PULLUP);
```

We will use the `INPUT_PULLUP` option this time so when the switch is not pressed, the pin will be internally pulled up to the 5 volt supply and the input will register as HIGH. When we press the button, the switch will close and apply a ground, or LOW to the pin. This is how we will be able to check the condition of the switch.

We need to decide how to implement the variable light trigger setting. One way to do this is to create a **variable**. A variable is a memory location that can hold data and can be addressed using a simple name. We will name our variable ***trigger_level***. The only thing left is to decide what type of value this will be. There are many different types of data in programming. We can use variables to hold integers, real numbers, letters, logic values, and more. For the `analogRead()` function, it returns an integer number between 0 and 1023. So we will define our variable to hold an integer value. Before the `setup()` function, add the following line:

```
int trigger_level;
```

This line tells the software that we want it to reserve a memory location that will store an integer called `trigger_level`. We can now use this variable by that name anywhere in our program. The reason we declared it right at the top is so all of the functions have access to it. If we declared it in `setup()`, it would only be able to be used in the setup function. This is called the **scope** of a variable. We are going to set the default value once (in `setup()`) and then change its value in `loop()`. That means we need it to have a global (whole program) visibility. That is why we declared it first thing in our program.

Now let's give it a default value. Add the following code to the `setup()` function:

```
trigger_level = 200;
```

This tells the software that we want to assign the value of 200 to the memory location we have set aside.

Now, we will change the code to use this new variable. In the `loop()` function, on the line we put the constant to compare the light signal with (*if* `(analogRead(A0) > 100)` change the constant to our variable. The line will look like the following:

```
if (analogRead(A0) > trigger_level) {
```

This if statement will now evaluate the current light level with our `trigger_level` variable. This is great because it means we can change it during program run time. And that is what we will do now.

Add the following conditional if statement before the other one in the `loop()` function:

```
if (!digitalRead(3)){  
    trigger_level = analogRead(A0);  
}
```

This code looks at the condition of the UNO pin 3 input. Remember, we set the input to normally look HIGH and it will go LOW when the button is pushed. The exclamation point is used in code to mean NOT. So the condition in the if statement really says if NOT `digitalRead(3)`. The HIGH condition is considered TRUE in computer terms. LOW is considered FALSE. So to get a TRUE output of the condition, `digitalRead(3)` would have to be LOW (button pressed) so when we NOT it, we get HIGH or TRUE and the condition is satisfied and the following

code in the brackets is run. Getting this logic right so the code does what you want is half the battle of programming.

So when the button is pressed, the code in brackets is run and the variable `trigger_level` is set with the current `analogRead` value. Now, that value will be used to determine if the LED should turn on or not. The full code listing is as follows:

```
int trigger_level;

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);

  pinMode(A0, INPUT_PULLUP);
  pinMode(2, OUTPUT);
  pinMode(3, INPUT_PULLUP);
  digitalWrite(2, LOW);

  trigger_level = 200;
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.println(analogRead(A0));

  if (!digitalRead(3)) {
    trigger_level = analogRead(A0);
  }

  if (analogRead(A0) > trigger_level) {
    digitalWrite(2, HIGH);
  }
  else {
    digitalWrite(2, LOW);
  }
  delay(250);
}
```

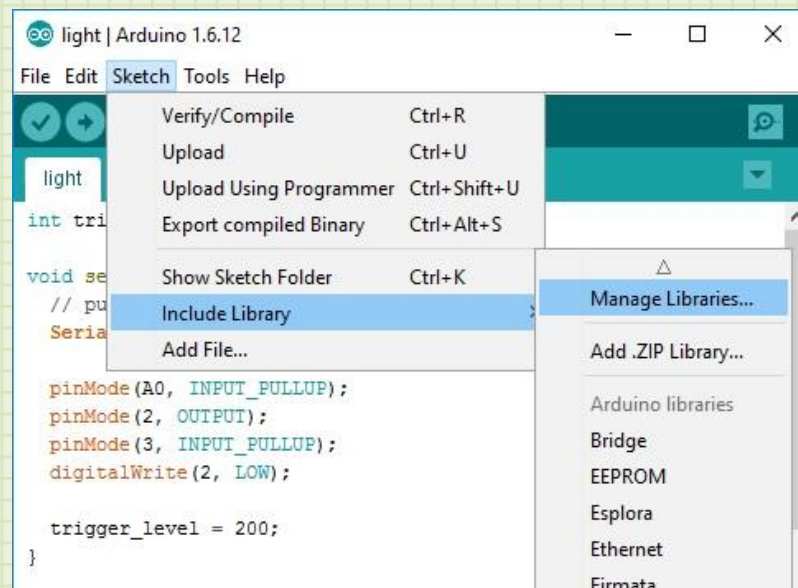

Upload this code to your UNO. Cover the light sensor partway and press the button. This will set the new value of the `trigger_level`. Release the button and move your hand over the light sensor to get the LED to turn on and off. This is such an advantage over the basic program where we hard coded the trigger level of the LED. If we took this device outside, we might have a really hard time triggering it. This modification allows us to update the trigger without having to upload new code. This was a great use of a variable.

Using Libraries

One of the more convenient aspects of Arduino is the ability to reuse code written by others. Let's build one final feature into our light alert program. Let's say the LED is not enough warning because it requires that you are looking at the device to know when the LED turns on. It would be very useful to have an alarm sound a short tone so you could have an aural signal as well as visual. Let's do that. Programming sound can be a complicated process on a microcontroller like the UNO uses. It requires that you generate a waveform of a specific frequency and often requires you to do this in the background using programming techniques called interrupts. Luckily, someone has already done this. So we will import the library and then use its functionality without having to write all of the code.

To do this, open our light program in the Arduino IDE.

Select **Sketch->Include Library->Manage Libraries...**



This will bring up the Library Manager. Here is a list of standard libraries that are directly supported in the IDE. There are a lot of libraries to choose from, but there are many libraries written by hobbyists that are not on the list. You can still import them into the IDE by downloading them as a .zip file, selecting **Sketch->Include Library->Add .ZIP Library...** and this will add it for your use.

In the Library Manager, scroll down the list until you get to the available **tone** libraries. Alternately, you can type tone in the filter box at the top to limit the choices and get you to the tone libraries quicker.

You may see multiple Tone libraries. Click on the one labeled **ToneLibrary**. You will now see a button on the lower right of the box that says “**Install.**” Click this button to install the library. After the library loads, click Close.

Now, let’s set up the speaker for our project on our breadboard. Make the following connections:

Component	Lead 1	Lead 2
Speaker	Black - 22Y	Red - 22H
Resistor	22G	22C

And connect to the UNO board:

Component	Breadboard	Arduino
Wire	22A	Pin 4

Turning attention back to our code, the first thing we will want to do is add the library to our sketch. Loading the library to the IDE just made it available. We need to actually add it to our sketch.

Select **Sketch->Include Library->ToneLibrary**

You may need to scroll down the list to find it. If you look at your code, it now has the following line:

```
#include <Tone.h>
```

This has been added automatically by the IDE. This tells the sketch to include all of the functions of this library. We can now make calls to the routines in the library and the IDE will recognize them.

To get the full functionality of the library, we will create an object that will have access to all of the code in the library. This is called creating an instance of the object. We could actually add a bunch of different speakers to the UNO board and create an instance for each.

Add the following code outside of all of the functions (right after the integer variable declaration is a good spot):

```
Tone spkr;
```

This creates a variable of the type `Tone`. This variable is how we will access the routines in the `Tone` library.

We have saved memory for our instance. But we still need to instruct our sketch to use it. We will do this with the `begin` function. Many libraries have a `begin` function (remember our `Serial` function?) to initially assign and setup a library device. Add the following to the `setup()` function right after the `Serial.begin` line:

```
spkr.begin(4);
```

This initializes the `spkr` variable and lets the library know what pin we are using.

We added a speaker to our project so we will need to configure the UNO pin as an output. Add the following line to the `setup()` function:

```
pinMode(4, OUTPUT);
```

Let's implement our tone warning as its own routine. Just like the functions `setup()` and `loop()`, we can create small packages of code under a function name and then call that function to execute the code. It is a great way to code because it prevents you from having to repeat code and it keeps your code more readable. Let's create a function called **triggerSound**. In your code, after the very last bracket, insert the following:

```
void triggerSound(){
  spkr.play(NOTE_A3, 25);
  delay(25);
  spkr.play(NOTE_A4, 50);
  delay(50);
}
```

Since this function will not return any values, it is declared as void. The next command is the play command. It will play a note for a given duration. You can get more information on any of the libraries from the sources that supply them. The libraries available in the Library Manager have links to the developer's site. Another advantage of libraries is that they sometimes define difficult values into easy to understand and use formats. The ToneLibrary comes with a included file that defines the frequency of the musical notes that it can play. You access them with NOTE_ followed by the note and octave (A3) to play. You can then enter a duration that the tone will play. This function is referred to as non-blocking. That means that if you ask it to play something, it will start that process and then return to the main program. Because of this, it can then be triggered again before the tone has completed playing. This can cause problems in some programs if the function is called too many times. To prevent issues, we will just have our program wait the same amount of time the note is playing. We use two notes in our function, but you can add anything you want. This could be the sound of a French police siren, a song, or whatever you like.

Now, when the sketch determines that the light limit has been reached and turns on the LED, let's also call our triggerSound() function. In the loop() function right after the digitalWrite(2, HIGH) command, add the following:

```
triggerSound();
```

This will play our alert in the spkr. The complete code listing is below:

```
#include <Tone.h>

int trigger_level;
Tone spkr;

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  spkr.begin(4);

  pinMode(A0, INPUT_PULLUP);
  pinMode(2, OUTPUT);
  pinMode(3, INPUT_PULLUP);
```

```

pinMode(4, OUTPUT);

digitalWrite(2, LOW);

trigger_level = 200;
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.println(analogRead(A0));

  if (!digitalRead(3)) {
    trigger_level = analogRead(A0);
  }

  if (analogRead(A0) > trigger_level) {
    digitalWrite(2, HIGH);
    triggerSound();
  }
  else {
    digitalWrite(2, LOW);
  }

  delay(250);
}

void triggerSound() {
  spkr.play(NOTE_A3, 25);
  delay(25);
  spkr.play(NOTE_A4, 50);
  delay(50);
}

```

Upload the code to your UNO. Set the trigger level as before. Now, when you trigger the LED, you will hear a quick tone to let you know the trigger level has been reached.

Conclusion

You have come a long way. You have loaded the Arduino IDE, ran a sample project, learned how to construct temporary circuits with a breadboard, wrote code to control the input and output of the UNO, communicated over the Serial port, and added and used a library. These are the basic skills needed to do so many projects of your own. It is our hope that you will continue to program and learn the full potential of this amazing platform.

Appendix

For more information on Arduino, visit: www.arduino.cc

For more Arduino information, add-on kits, and different board variants visit:
www.sparkfun.com

www.adafruit.com