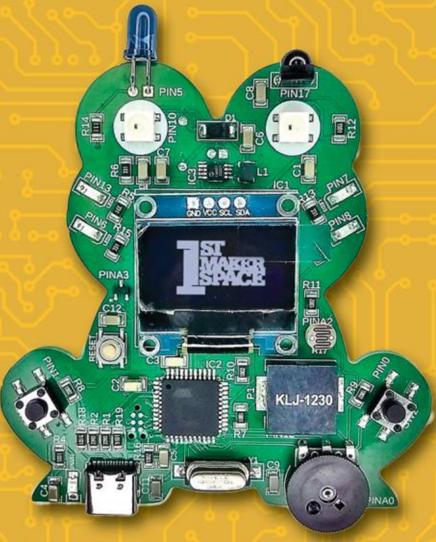
LEARN ARDUINO PROGRAMMING *with the* Microcontroller *Trainer*



1ST MAKER SPACE
Design.
Build.
Sustain.

We bring hands-on learning to life.

Nikolas Mahaffey & Kim Brand

www.1stMakerSpace.com

Edition 2.0: January 2023

Contents

Table of Contents

Getting Started

This book aims to get beginners started with the Arduino Programming system – people who may not know anything about programming or electronics. The best way to learn is by doing, so we want to give you just enough information so that you can understand what's going on and then start doing things. This is how we started with Arduino.

You'll use our special Arduino Leonardo compatible board, the Micro Controller (MC) Trainer, to help you get started immediately.

What is an Arduino?

It can be confusing at first. You might hear people say, "I have an Arduino," "Do you know Arduino?" or "I like to work with Arduino." Arduino is three things:

1.) **The hardware**. Arduino is based on a family of microcontrollers built by Atmel. These microcontrollers are miniature computers. They incorporate means of communicating with the outside world (called Input/Output or I/O in microcontroller language), memory (a combination of volatile and non-volatile types*), and a computer processor. All on one chip. *Volatile memory is erased when the power is shut off from the chip. Non-Volatile memory persists even when the power is off.

2.) **The language.** The founders of Arduino came up with a unique programming language to instruct the chip on what to do. The Arduino language is a subset of a popular programming language used by many computer programmers called C++. We will use Arduino (the language) to program our MC. An Arduino program is called a 'sketch'.

3.) **The community.** What makes Arduino unique is the community of users around the world. And Arduino users like to share. Having so many people using Arduino inspires us to do projects we never imagined. It's also easy to find examples of hardware and software designed by other Arduino users to aid in our understanding of how microcontrollers work and what is possible.

How does it Work?

The basic idea behind an Arduino project is to write a computer program, called a sketch, to control the operation of an electronic circuit – usually to send electronic commands to devices like LEDs or motors or receive input from sensors like thermistors or pushbuttons. The MC Trainer has several interesting electronic circuits built in. You’ll soon be ready to write your sketches and build your own circuits!

The basic process begins with ‘sketches’ (programming source code) that are written and compiled, and the resulting ‘machine code’ is uploaded to the MC Trainer using the Arduino IDE (integrated development environment). Compiling a program converts your program's English language version into the MC's binary commands.

Our first step is to download the IDE software to a PC or Mac and configure it for use with the MC Trainer.

Setting up Your Computer

The setup is easy. You can find complete instructions for your operating system here:

http://arduino.cc/en/Guide/HomePage

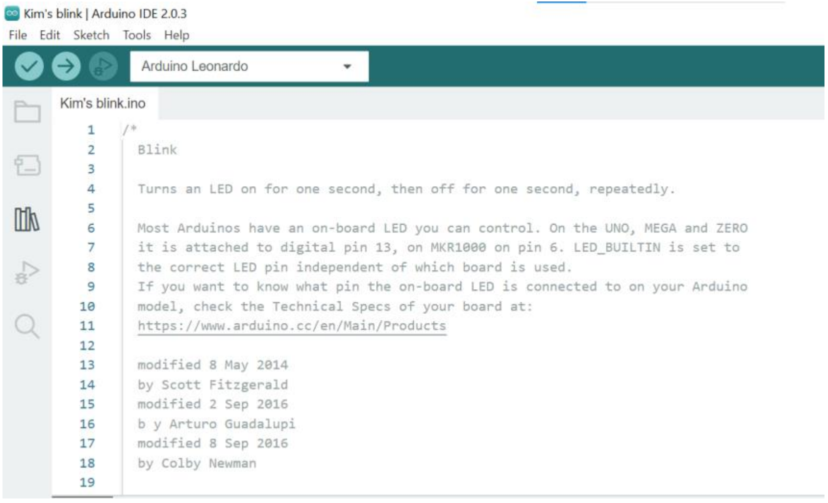
We'll go through the setup process for Windows machines here. Some parts of the process may differ for Linux and Mac OSX. Follow the link above for instructions for these operating systems.

You first need to download the Arduino IDE. You can find it here:

https://www.arduino.cc/en/software

The most recent version of Arduino as of writing this book is 2.1.0. Click to download the installer and follow the directions to install Arduino on your PC.

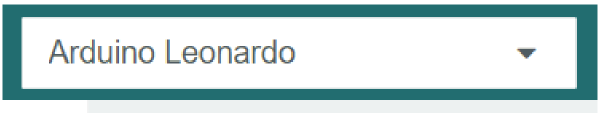
Now let's open the IDE and take a look at what each button does:



```
/*
  Blink

  Turns an LED on for one second, then off for one second, repeatedly.

  Most Arduinos have an on-board LED you can control. On the UNO, MEGA and ZERO
  it is attached to digital pin 13, on MKR1000 on pin 6. LED_BUILTIN is set to
  the correct LED pin independent of which board is used.
  If you want to know what pin the on-board LED is connected to on your Arduino
  model, check the Technical Specs of your board at:
  https://www.arduino.cc/en/Main/Products

  modified 8 May 2014
  by Scott Fitzgerald
  modified 2 Sep 2016
  b y Arturo Guadalupi
  modified 8 Sep 2016
  by Colby Newman
```

Verify: This button *compiles* the sketch to get it ready to upload. When a sketch is compiled, it is translated into a unique format readable by the microcontroller. The compiler will tell you if there are any errors and give you hints as to what's wrong. You don't need to verify the sketch before you try to upload, but it won't upload if there are any errors.

Upload: Compiles the sketch and uploads it to your board in one step. Like the verify button, it will return an error message if the sketch contains errors.


Arduino Leonardo

The 1st Maker Space Microcontroller Trainer appears like an Arduino Leonardo. Make sure the IDE is configured for that board type.

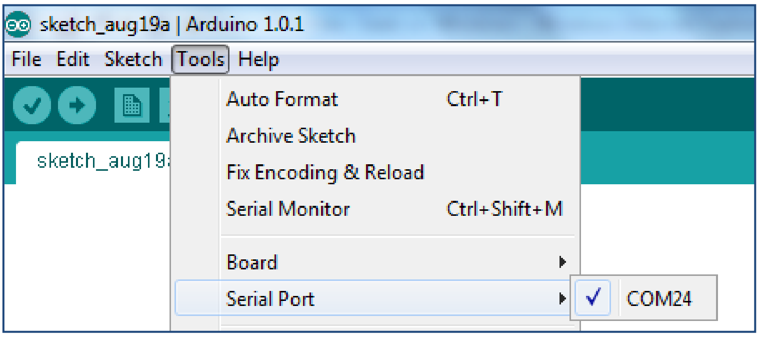
When you open the IDE, a new tab with today's date will appear. You can write a sketch in the tab or copy and paste it from another source.

You can download the sketches for our projects from our website:

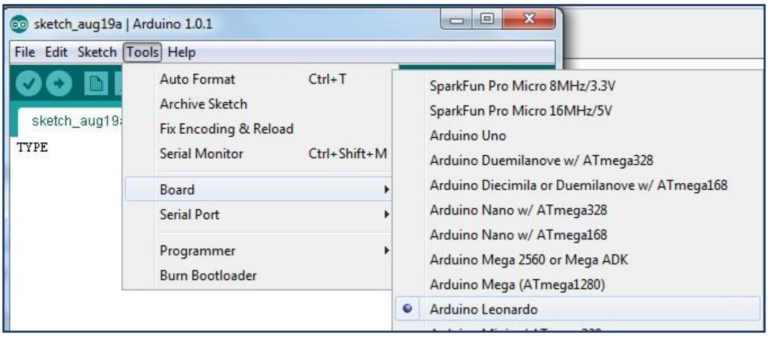
http://www.1stmakerspace.com/MC-Trainer-sketches.html

Although downloading the sketches is an option, we suggest you type them into the IDE, especially if you're new to programming. It will help you learn the particular syntax of the Arduino programing language, and you'll be able to start creating sketches of your own faster than if you only downloaded or copied the sketches.

The first time you plug in your MCU Trainer, the computer will try to find a driver. It will prompt you to download these drivers during installation; click "install." You may need to disconnect the MC Trainer after the driver is installed and then reconnect it. You will know that the MC Trainer is connected properly when you plug it into the USB port, and you can select the serial port in the IDE (Tools > Serial Port).   You need to select the serial port before you can continue.



The last thing you need to do before you can write and upload a sketch is tell the IDE which type of board you are using. There are many different types of Arduino boards. The MC Trainer is based on the Leonardo, so select “Arduino Leonardo” under Tools > Boards.


sketch_aug19a | Arduino 1.0.1
File Edit Sketch Tools Help
sketch_aug19a
TYPE
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Board
Serial Port
Programmer
Burn Bootloader
SparkFun Pro Micro 8MHz/3.3V
SparkFun Pro Micro 16MHz/5V
Arduino Uno
Arduino Duemilanove w/ ATmega328
Arduino Diecimila or Duemilanove w/ ATmega168
Arduino Nano w/ ATmega328
Arduino Nano w/ ATmega168
Arduino Mega 2560 or Mega ADK
Arduino Mega (ATmega1280)
● Arduino Leonardo

The Arduino IDE has everything you need to write a sketch (a computer program) and upload it to your MC Trainer. Once the sketch is on the MC Trainer, it will run continuously if it has adequate power. The sketch stays in the microcontroller’s memory when the MC Trainer is unplugged from your computer. It will begin again when the MC Trainer is powered back on. Your computer’s USB port usually powers the MC Trainer, but you could also power it with a USB “wall wart” charger, other power source with a USB connector or the battery connection on the back. The battery connection on the back should not be powered by more than 3V (3V = Two AA or Two AAA).

The Arduino IDE has a few features to help you with programming. Programming keywords (words that do tasks) are color-coded orange in the programming window. Pre-defined variables (labels for values stored in memory) are color-coded blue. The main thing to remember is that words that appear in a color other than black are special. If you want to know what a keyword does, select the entire word and press Crtl-Shift-F. This will automatically open the Arduino reference webpage for the word. The IDE also does parentheses () and bracket {} matching. Both parentheses and brackets have special meanings in Arduino and are used in pairs. It is often helpful to find the matching pair. The IDE will highlight the match to a parenthesis or bracket when the cursor is on the right-hand side of one member of a pair.

The MC Trainer Board

The MC Trainer board was built so that you can learn the basics of Arduino without doing any soldering or wiring.
The brain of any Arduino is the microcontroller. A microcontroller is a miniature computer with memory, a processor, and input and output channels all in one package. Our microcontroller is the ATmega32U4, manufactured by the Atmel Corporation. This is the same microcontroller used on the Arduino Leonardo.

The microcontroller chip looks like a bug with 44 short metal legs. These are the microcontroller pins. The legs are soldered to the circuit board and provide electrical connections between the inside of the chip and the rest of the board. Some pins bring power into the chip (VCC) and provide paths to ground. Other pins are for communication through USB and with another integrated circuit on the board. The rest of the pins are input/output (I/O) pins. These pins are used to communicate with or control other components on the board. The **Pin Key** at the end of this book shows which pins are connected to each of the circuits on the MC Trainer.

The USB hub of your computer powers the microcontroller and the rest of the MC Trainer board. The USB connector on the MC Trainer is located below the microcontroller. Your computer supplies 5V and up to 500 mA of current. That’s just right for powering the microcontroller and other components. Using the USB connection, we also load sketches and communicate between the board and the computer.

Programming

A program, or sketch, is a set of instructions the MC Trainer will execute. Computers take the world very literally. They need to know precisely what to do in a language they understand. Luckily, programming in Arduino is relatively simple. But you need to know the rules and follow them exactly. Here are a few rules to remember:

- All sketches need at least two parts: the **setup**() and **loop**() functions. We'll explain them below.
- Most lines end with a semicolon**;** We usually press the Enter key after the semicolon, but Arduino knows that the line ends when it sees a semicolon.
- Arduino is case-sensitive. So the word 'setup' is different from 'Setup.' This is true of both Arduino commands and variables that you create.
- When Arduino sees the two characters '//,' it ignores everything until the end of the line (when the Enter key is pressed). This allows us to add human-readable comments to our sketches. Arduino doesn't need to know what our comments mean. They're used to describe to other people what the program does and remind us what we thought when we wrote it. You can also include a block of several lines of comments in a sketch by starting the block with **/*** and ending it with ***/**.

In the remainder of the book, we <u>underline</u> the names of variables, bold programming keywords, and other symbols when discussing Arduino code.

The setup() and loop() functions

There are many different ways of writing sketches, but every sketch must have at least two parts: the setup() and loop() functions. The setup() function conventionally appears in the sketch before the loop() function. A function is a section of code that runs together. To distinguish a function from the rest of the sketch, it will always start with a left-hand curly bracket { and end with a right-hand curly bracket }. The basic form of a sketch looks something like this:

```
void setup(){
        do a task;
        do another task;
}

void loop(){
        do the main tasks; and
        more tasks; and more
        tasks;
        .
        .
        .
}
```

The keyword **void** must appear before these function names. We discuss what it means later in this chapter, where we cover **functions.**

The **setup()** function runs only once when the sketch begins. This means it will run after a sketch is uploaded to the MC Trainer. If a sketch is already on the MC Trainer, the **setup()** function will run once when the MC Trainer is powered up. We typically do "housekeeping" tasks in the **setup()** function to prepare things for the sketch's main part. For example, we might set the modes of the input/output pins that we will need in the sketch or get some initial input from the user or a sensor.
After the **setup()** function runs once, the sketch enters the **loop()** function. The lines of code in the **loop()** function are run one after another. When we hit the **}** at the bottom of the **loop()** function, the sketch returns to the top and runs the lines over again. This continues as long as the MC Trainer is powered up.

Variables

A variable is a label that we give to a piece of information. This gives us a simple way to save, change, and access the information. We need to tell the Arduino that we want to create a spot in its memory to store the information by *declaring* a variable. We can declare a variable in different sketch parts depending on where we want to use it. We also have to tell the Arduino what type of variable we want so it can reserve enough space and interpret it.

Computers see the world as a bunch of 0's and 1's. These are called bits. The more bits we use for a variable, the greater the range of values it can take on. The range of values we can store is 2 to the power of the number of bits used ($2^{\text{\# of bits}}$). For example, with 8 bits, we can store up to 256 different values (e.g., 0, 1, 2, 3, …, 255). With 16 bits, we can store 65536 distinct values.
In many cases, we want to be able to store both positive and negative values, so we may use 16 bits to store values between -32,768 and 32,767. Different variables in Arduino use either 8, 16, or 32 bits. We only have limited memory for these bits, so we want to use the smallest number to complete the job. Some of the most common variable types are:

- **byte**: an 8-bit variable representing a number between 0 and 255.
- **char**: also 8-bits, but Arduino interprets as a character like 'a' or '!'
- **boolean**: an 8-bit variable that can only hold true or false values.
- **int**: a 16-bit integer. Integers are numbers without decimal places. An **int** can hold positive or negative values, ranging from -32768 to 32767.
- **long**: a 32-bit integer. The extra bits allow us to store values between -2,147,483,648 to 2,147,483,647.
- **float** or **double**: these are 32-bit variables with decimal places with values like 3.14159. Some of the bits are used to tell where the decimal place goes. This leaves 6 to 7 digits of precision. Many programmers avoid using **float** variables since they require more complicated math.

To declare a variable, you tell Arduino what type of variable it is and its name. You can also give it an initial value (which can be changed later if you want). Some examples:

```
int start;   //we'll assign a value to this variable later in the code
int count = 10;
long pastime = 2350000;
char firstLetter = 'a';
```

A few more notes on variables. In some cases, you may need to store values outside the normal range of the **long** type. We don't use them in our projects, but you can use an **unsigned long** to store very large values. An **unsigned long** cannot hold negative numbers (unsigned means that the +/- sign isn't used), but the range is from 0 to 4,294,967,295.

Also, variables are stored in volatile memory, meaning their values are lost when the MC Trainer loses power.

Arrays

Each variable type can also be declared as arrays, which are groups of values of the same type. For example, we can declare an **int** variable with one value:

```
int myValue = 1;
```

Or an array with multiple values:

```
int myValue[] = {5,3,2,7,8,10,155};
```

Here we declared an array with 7 values. Arduino creates 7 places in memory for these values. We can also tell Arduino to create the spots in memory and put the values in later:

```
int myValue[7];
```

To assign a value to the first spot, we use a command like this:

```
myValue[0] = 155;
```

The number in the [] brackets is called the index. We want to change or look at this spot in the array. The first spot always has an index value of 0, and the last spot has an index value of 1 less than the array's length. For example, an array of 7 values has index values between 0 and 6.

Arrays of **char** variables work a little differently:

```
char myMessage[] = “Hello World”;
```

This creates an array with 12 places. This might look wrong since there are only 11 characters in “Hello World” (Including the space). Arduino creates an extra spot for a special character (called the null termination) that keeps track of where the array ends. This is helpful when we do things like sending the array to a computer screen.

Another special type of “variable” is a **String.** A **String** is not actually a variable; it’s an object. Although it’s an object, a **String** can be used like any other variable type. Like a **char** array, a **String** holds multiple characters (computer programmers often call groups of characters strings). Here’s how we declare a **String:**

```
String myMessage = “Hello World”;
```

Notice that we didn’t use the [] brackets. One special property is that we can change the length of a **String** after we declare it:

```
String myMessage = “Part 1”;
myMessage = myMessage + “ and  “ + “Part 2”;
```

Now myMessage has the value “Part 1 and Part 2”. We can check the length of the **String** object with:

```
int howLong = myMessage.length();
```

This command looks weird because we give the variable's name, followed by a period, and then ask for the length. This is an example of a *method* for an object of the **String** class. You don’t need to worry about this now, but you might see this type of syntax in other parts of Arduino sketches. There are a lot of additional useful methods for **String** objects. You can look at the Arduino reference online to see all the others.

Some Thoughts on Programming

Programming is a lot of fun once you get the hang of it. It's like solving a puzzle. You start with an idea, goal, or problem you're trying to solve. You start by breaking the problem into manageable pieces: how do I collect the information I need? Which decisions am I going to make based on the information? What are my outputs? Then you translate these ideas into your computer language (Arduino in this case), following all its rules. Then you test your program and make improvements. It's a great feeling to start with an idea and then bring it to life through programming.

It's important to do things step by step and make sure your code works along the way. For example, you might write a program that takes a sensor reading and controls a motor. If you write the entire program, test it, and it doesn't work, you're left wondering, "Is the problem with the sensor, the motor, or my program?" It makes more sense to write the code for the sensor first and then output the results to the serial monitor. You can move on to the motor once this works and the results make sense. With the motor, you might start with "hard coded" values to control it. For example, if the motor is supposed to turn forward when the value of the variable <u>sensorOutput</u> is > 10 and backwards when it is < 0, you can just include the line <u>sensorOutput</u> = 12 and see what the motor does. Then you can see what the motor does when you change the line to <u>sensorOutput</u> = -2. Once you're sure each part of the program works on its own, you can focus on combining them.

Programming languages can seem rigid with their need for exact syntax and structure. But as long as you follow the rules, you can do *anything*. There's tons of room for creativity. It will probably work if you assemble a logical sequence of steps to solve a problem and put it into the proper syntax. You need to check your syntax and rethink your logic if it doesn't. Invariably, you're going to learn something from the experience. There's almost always more than one way to do the same task. As you become more skilled, you'll be able to accomplish more tasks with fewer lines of code. You'll find shortcuts and learn to write flexible, reusable code that will work in different situations.

You're now ready to jump into the projects. You can download these projects from our website or type them in from the book. I suggest that you type them in. It will help you become familiar with the language and learn the syntax. Once you get a project to work, you can tweak it. Test some hypotheses: “If I change this line, then that will happen.” Try it. It’s a great way to learn.

Projects

We covered downloading and setting up the Arduino IDE in the **Getting Started** section. If you haven't set up the IDE yet, do that now. Let's review the steps to loading a sketch onto the MC Trainer:

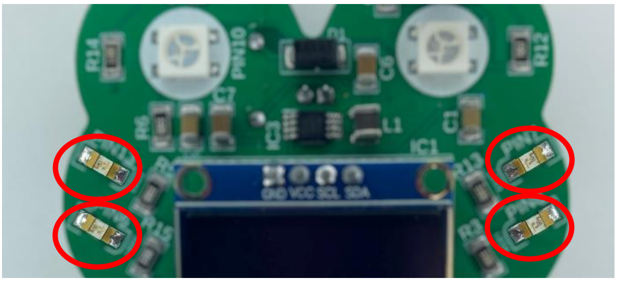
1.) Type the sketch into the Arduino IDE's programming window or download it from 1stmakerspace.com and open it in the IDE.
2.) Connect the MC Trainer to your computer with the USB cable.
  a. Select the "Leonardo" board from the Tools>Board menu in the IDE.
3.) Select the available COM port from the Tools>Serial Port menu in the IDE.
4.) Upload the sketch by clicking on the upload button in the IDE toolbar.

Most Arduino platforms interact with circuits connected to the microcontroller's input/output pins. You can go to the **MC Trainer Board** chapter to see how each of the circuits work. The pin numbers for each circuit are listed at the back of the book in the **MC Trainer Pin Key.** We use the same variable name in each sketch when interacting with a pin. For example, the momentary switch on the left-hand side of the board is always referred to as SW1. These variable names are also listed in the **Pin Key.**

You can find sketches that use particular parts of the Arduino language by looking at the **Project Index** at the back of the book. The projects are divided into several sections. Each section focuses on a different set of circuits and programming concepts (although there's a lot of overlap between the sections). After we present each sketch, we go into detail in describing how the sketch works. We underline variable names and **bold** Arduino statements and functions when describing the sketch's parts.

Now let's get started

Lesson 1: Starting with LEDs



An LED, or **L**ight **E**mitting **D**iode, is an electronic component that emits light when an electrical current flows through it.

Project 1.00 Blink

Nearly everyone starts by learning how to blink an LED. Let's take a second to think about how a light blinks. First, a light turns on, then waits for some amount of time, then turns off, waits for some amount of time, and repeats. That process is what we need to create in Arduino code. There are three main components to this sketch:

Keyword	Description	Parameters
pinMode	**pinMode** is used to set a pin as either an input or an output	1. Pin number 2. INPUT / OUTPUT Example: **pinMode**(13, OUTPUT);
digitalWrite	**digitalWrite** is used to write a pin either as a digital HIGH (5v) or a digital LOW (0v)	1. Pin number 2. HIGH / LOW Example: **digitalWrite**(13, HIGH);
delay	**delay** is used to wait for a specific amount of time in milliseconds	1. Time in milliseconds to delay the program Example: **delay**(1000);

Project Code:

```
////////////////////////////////////////////////////
//Project 1.00 Blink
byte LED1 = 13;
void setup(){
   pinMode(LED1,OUTPUT);
}
void loop(){
   digitalWrite(LED1,HIGH);
   delay(1000);
   digitalWrite(LED1,LOW);
   delay(1000);
}
////////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

Let's take a closer look at how this sketch works. We declare one **byte** variable at the top of the sketch. It is a global variable since it is declared outside the **setup()** function, **loop()** function, or any other function. This means we can use it anywhere else in the sketch and it will be recognized. LED1 gets assigned the value 13 because that's the pin number (on the microcontroller) that LED1 is connected to.

```
byte LED1 = 13;
```

Every sketch needs one **setup()** and one **loop()** function. The **setup()** function runs only once. That's all we need to set the **pinMode** of the LED to output so that we can switch it on and off:

```
void setup(){
   pinMode(LED1,OUTPUT);
}
```

Now comes the **loop()** function. This function will run repeatedly. At the top of the block comes the **digitalWrite** statement. This powers the pin attached to LED1 with 5 V, causing the LED to light up.

```
void loop(){
   digitalWrite(LED1,HIGH);
```

LED1 will remain in a **HIGH** state until we tell it otherwise or we disconnect the MC Trainer from its power source. We want it to stay on for only a second, so we wait 1000 milliseconds (1 second):

```
      delay(1000);
```

And then switch the pin to **LOW.** Now the LED switches off:

```
      digitalWrite(LED1,LOW);
```

We keep it off for another second and then finish the **loop()** function:

```
      delay(1000);
   }
```

The closing bracket tells the MC Trainer to go back to the top of the **loop()** function and repeat it.

Try seeing how fast the LED can blink by changing the number in the **delay** function. Just a hint, it can blink faster than we can see!

Project 1.01 Blink x2

In this project, we bring a second LED, LED3, into the mix. We are essentially doing the same thing as the last project, but this time we are using two LEDs.

Project Code:

```
//////////////////////////////////////////////////
//Project 1.01 Blink x 2
byte LED1 = 13;
byte LED3 = 7;

void setup(){
   pinMode(LED1,OUTPUT);
   pinMode(LED3,OUTPUT);
}

void loop(){
   digitalWrite(LED1,HIGH);
   digitalWrite(LED3,HIGH);
   delay(1000);
   digitalWrite(LED1,LOW);
   digitalWrite(LED3,LOW);
   delay(1000);
   digitalWrite(LED1,HIGH);
   digitalWrite(LED3,LOW);
   delay(1000);
   digitalWrite(LED1,LOW);
   digitalWrite(LED3,HIGH);
   delay(1000);
   digitalWrite(LED1,LOW);
   digitalWrite(LED3,LOW);
   delay(1000);
}
//////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

Like all sketches, this simple sketch includes a **setup()** and a **loop()** function. Before the **setup()** function, we declare two variables that refer to the pin numbers for LED1 and LED3:

```
byte LED1 = 13;
byte LED3 = 7;
```

In the **setup()** function, we set both pins to **OUTPUT** using two **pinMode** statements:

```
   pinMode(LED1,OUTPUT);
   pinMode(LED3,OUTPUT);
```

In the **loop()** function, we first switch both LEDs on by setting the pins to **HIGH** using two **digitalWrite** statements:

```
void loop(){
   digitalWrite(LED1,HIGH);
   digitalWrite(LED3,HIGH);
   delay(1000);
```

After a 1-second **delay**, we turn both LEDs off:

```
   digitalWrite(LED1,LOW);
   digitalWrite(LED3,LOW);
   delay(1000);
```

Next, we turn only <u>LED1</u> on:

```
   digitalWrite(LED1,HIGH);
   digitalWrite(LED3,LOW);
   delay(1000);
```

And then switch so that only <u>LED3</u> is on:

```
   digitalWrite(LED1,LOW);
   digitalWrite(LED3,HIGH);
   delay(1000);
```

Finally, we turn both LEDs off for 1 second before the **loop()** function reaches its closing bracket **}** and begins again at the top:

```
   digitalWrite(LED1,LOW);
   digitalWrite(LED3,LOW);
   delay(1000);
}
```

Project 1.02 Simple LED Chase

In this project we will use all four LEDs to blink a chase pattern. This example shows a very simple way to create a pattern using **digitalWrite**s and **delay**s.

Project Code:

```
//////////////////////////////////////////////////
// Project 1.02 LED Chase

byte LED1 = 13;
byte LED2 = 6;
byte LED3 = 7;
byte LED4 = 8;

void setup() {
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);
  pinMode(LED4, OUTPUT);
}

void loop() {
  digitalWrite(LED4, LOW);
  digitalWrite(LED1, HIGH);
  delay(250);
  digitalWrite(LED1, LOW);
  digitalWrite(LED2, HIGH);
  delay(250);
  digitalWrite(LED2, LOW);
  digitalWrite(LED3, HIGH);
  delay(250);
  digitalWrite(LED3, LOW);
  digitalWrite(LED4, HIGH);
  delay(250);
}
//////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

Here we set the four variables needed to represent the pins that the LEDs are attached to:

```
byte LED1 = 13;
byte LED2 = 6;
byte LED3 = 7;
byte LED4 = 8;
```

Next, we need to tell the microcontroller that these pins are outputs:

```
void setup() {
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);
  pinMode(LED4, OUTPUT);
}
```

After that, we can sequentially turn on and off the LEDs to represent a chase pattern. To do this, we have to turn the last LED off, the next LED on, and then wait.

```
void loop() {
  digitalWrite(LED4, LOW);
  digitalWrite(LED1, HIGH);
  delay(250);
  digitalWrite(LED1, LOW);
  digitalWrite(LED2, HIGH);
  delay(250);
  digitalWrite(LED2, LOW);
  digitalWrite(LED3, HIGH);
  delay(250);
  digitalWrite(LED3, LOW);
  digitalWrite(LED4, HIGH);
  delay(250);
}
```

What you may have noticed about the code above is that something happened over and over again. Usually when things happen over and over again it is a sign that it should be in some kind of loop and/or function.

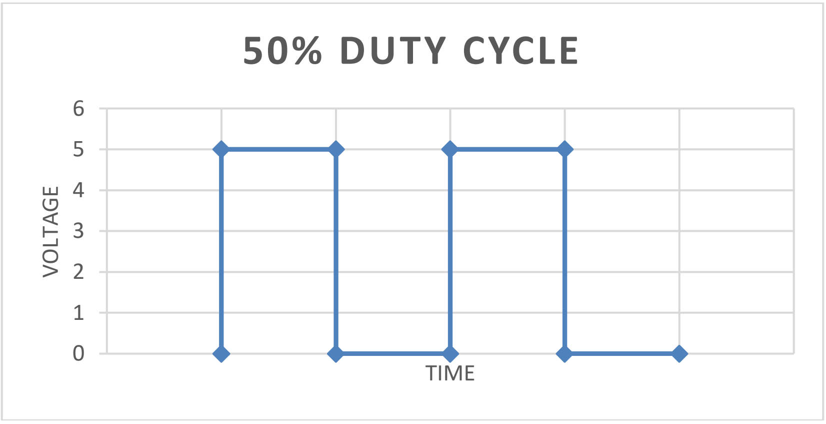
Project 1.03 Analog Write

In this project you will learn about a new function called **analogWrite**. This function is useful for setting the brightness of LEDs. It can also be used to set the speed of motors, servo motor position, generating audio tones, and more.

The MC Trainer can only produce values of 0v and 5v but what happens when you need 2.5v? Simply put, the MC trainer cannot produce 2.5v. Instead, what it can do is switch a pin between 0v and 5v very quickly and produce an average voltage of 2.5v. This process is known as PWM.

It is important to note that this can only be done with certain pins. In our case, LED1 and LED2 can be used to do PWM.

This also introduces the Idea of duty cycle. Duty cycle is the ratio of how long a signal is on VS how long a signal is off. In this case the duty cycle would be 50%.



Project Code:

```
////////////////////////////////////////////////
// 1.03 - Analog Write

byte LED1 = 13;

void setup() {
  pinMode(LED1, OUTPUT);
  analogWrite(LED1, 127);
}

void loop() {

}
////////////////////////////////////////////////
```

*If you’re copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

The **analogWrite** function takes an 8-bit value between 0 – 255. 0 represents 0v and 255 represents 5v. The average voltage produced by that pin can be calculated with the equation: (analogValue * 5) / 255, with analogValue being the 8-bit number passed to the **analogWrite** function. For example, (127 * 5) / 255 = 2.49V. So, 127 will produce 2.49 volts.

```
analogWrite(LED1, 127);
```

When you run the code, the LED will be at half brightness.

Project 1.04 Pulse LED

This project demonstrates the full range of the **analogWrite** function, and how that can be used to create a pulsing effect.

Project Code:

```
/////////////////////////////////////////////////
// 1.04 - Pulse LED

byte LED1 = 13;

void setup() {
  pinMode(LED1, OUTPUT);
}

void loop() {
  byte wait = 10;

  for (int i = 0; i < 255; i++) {
    analogWrite(LED1, i);
    delay(wait);
  }

  for (int i = 255; i > 0; i--) {
    analogWrite(LED1, i);
    delay(wait);
  }
}
/////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

Before we jump into the code let's take a look at what a **for-**loop is. A **for-loop** is a way to repeat a chunk of code a specified number of times. See the table below for a breakdown of a **for-loop** structure:

Part	Description
for	Loop type
Int i = 0;	This creates a variable for the loop to keep track of how many times it's looped
i < 255;	This is a conditional. The **for-loop** will continue to run while this is true. In this case, i is less than 255
i++	This increments the i variable

	by 1 every time the loop executes

{
analogWrite(<u>LED1</u>, i);
delay(<u>wait</u>);
}
	This is the code that runs every time the loop executes

for-loops can be a foreign concept. For a video explanation please visit:

https://www.youtube.com/watch?v=b4DPj0XAfSg

Here in the program a **for-loop** is used to count 0 – 255, slowly increasing the brightness of the LED. The index of the **for-loop** (i)is passed to the **analogWrite** function as the **for-loop** loops. A small **delay** is added after each **analogWrite** so the changes can be perceived by the human eye.

```
  for (int i = 0; i < 255; i++) {
    analogWrite(LED1, i);
    delay(wait);
  }
```

Here the program counts down 255 – 0, slowly decreasing the brightness of the LED.

```
  for (int i = 255; i > 0; i--) {
    analogWrite(LED1, i);
    delay(wait);
  }
}
```

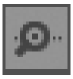
Lesson 2: Using Serial

Serial refers to the process of sending or receiving data one bit at a time, sequentially, over a communication channel or computer bus. It's a built-in object that represents the serial port on the Arduino to send and receive data.

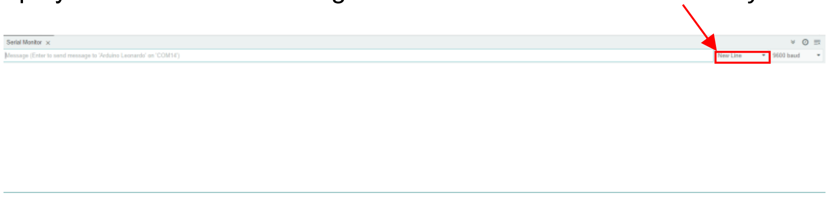
Project 2.00 Serial Printing

In this project you will be introduced to **Serial**. **Serial** is a great way to see what is happening while your projects are running. **Serial** allows you to print strings of text, numbers and more out into a “Serial Port”. To open the **Serial** Port, click the magnifying glass in the top right of the Arduino IDE, or press ctrl + shift + m.

Here is what the magnifying glass will look like:



When you open the **Serial** port, it will appear in a window at the bottom of the screen. This is where the things you print will be displayed. Make sure to change this to “Newline” if it’s not already.


Serial Monitor
Message (Enter to send message to 'Arduino Leonardo' on 'COM14')
New Line
9600 baud

Project Code:

```
///////////////////////////////////////////////
// 2.00 - Serial Printing

void setup() {
  Serial.begin(9600);
}

void loop() {
  Serial.print("My name is: ");
  Serial.println("YourNameHere");
  delay(1000);
}
///////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

This line initializes the **Serial** port for communication. The 9600 value passed is the "Baud rate". Baud rate is the rate of bits transmitted per second. Because our microcontroller has a built-in USB interface, the number we pass here does not matter.

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

There are a few common ways to print information to this screen. We can use the **Serial.print** and/or **Serial.println** methods.

The difference is that the **Serial.println** method will print a new line after it prints what was passed to it. It is important to note that if the data being printed is a string of characters, they need to be enclosed in quotations. If it is a single character, it can be enclosed in quotations or apostrophes. Otherwise, numbers and variables do not need to be enclosed in anything special.

```
  Serial.print("My name is: ");
  Serial.println("YourNameHere");
```

Change out the "YourNameHere" string with your name (enclosed in quotation marks).

Try making them both just **Serial.print** and see what happens!

A **delay** is needed to keep our MC Trainer from printing thousands of lines of text in a few seconds!

```
  delay(1000);
```

Make sure and open the **Serial** port to see what is being printed!

Project 2.01 Talking to the Board

In the last project the MC Trainer sent messages to the computer using **Serial**. In this project we will do the opposite! We can also use the **Serial** Port to send messages to the MC Trainer! These messages will pop up in the **Serial** port when you type them in and send them.

Project Code:

```
//////////////////////////////////////////
// 2.01 - Talking to the board

void setup() {
  Serial.begin(9600);
}

void loop() {
  while (Serial.available() == 0) {
    ;
  }

  byte messageSize = Serial.available();
  char message[messageSize];

  for (byte i = 0; i < messageSize; i++) {
    message[i] = Serial.read();
  }

  Serial.print("Message sent: "); Serial.println(message);
}
//////////////////////////////////////////
```
*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

There are a few simple steps when we want to use the computer to communicate to the MC Trainer. Firstly, we must wait for **Serial** data to be sent from the **Serial** Port. The **Serial.available()** method will return the amount of data available from the **Serial** Port.

Let’s take a look at another kind of loop. In this program we use what’s called a **while-loop**. This kind of continuously runs while the conditional is true. See the table below for a breakdown of the **while-loop** used in the program:

Type	**while-loop**
Conditional	**Serial.available()** == 0
Code executed	;

The goal for the following lines of code is to keep us waiting in one spot in the program until we receive **Serial** data from the computer.

Serial.available() returns the amount of **Serial** data available to read from the computer. When it returns 0, that means there is no data to read. So, **while** there is no data to read, execute the code inside of the brackets “**{}**”.

The code inside of the brackets Is just a semi-colon. This essentially does nothing but take up some time. It’s an easy way to wait until something happens to move on. We are stuck in this **while-loop** (doing nothing) until our condition is met.

```
while (Serial.available() == 0) {
  ;
}
```

The next thing we need to do (After we’re out of the **while-loop**, meaning a message has been sent from the **Serial** port) is create a variable that can hold the incoming message. This will be an array of variable type **char** because **char** type variables hold a character. Using an array will allow us to store multiple characters in a convenient way.

How will we know how to size our array? In other words, how will we know how many spots our array needs for the incoming message? For that, we can use **Serial.available()** again. Remember that **Serial.available()** returns the amount of **Serial** data that is available to read. We can assign this returned value to a **byte** type for later use.

```
byte messageSize = Serial.available();
```

Once we know the size of the message, we create a character array to hold that message. First, we make the array type **“char**”, then name it “message”, then we size it with the variable we made earlier “[messageSize]”.

```
  char message[messageSize];
```

Let’s do an example. Let’s say our message is “Hello”. What you have to know is that although it looks like there are only 5 characters that make up the word “Hello”, there are really six when it is sent over from the computer. There is ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, and ‘\0’. The ‘\0’ is called a **NULL** terminator and signifies the end of a string of text.

Serial.available() will return the number 6 (Indicating that there are 6 characters to be read) and we can now create a variable to hold all of the characters.

Now we have a character array with 6 empty spots. Next, we need to use the **Serial.read()** method to read the Serial data. This method will read one **byte** (character) at a time sequentially. That means that calling it once will return the first character of what is being sent over, calling it again will read the second, and so on. As the incoming message is being read it needs to be stored in the array we created earlier.

Since we stored the length of the message in the messageSize variable, we can tell a **for-loop** how many times it needs to loop to read the entire message. The **for-loop** loops as many times as the message is long, and puts the characters read into the message **array**. We use the variable i to keep track of our spot in the **array**.

```
for (byte i = 0; i < messageSize; i++) {
   message[i] = Serial.read();
}
```

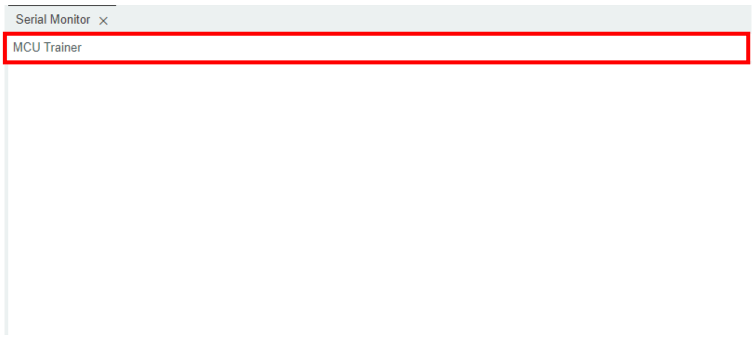
Visualized is the character array below when the **for-loop** finishes with the example “Hello” (Arrays start at Index 0):

Index ->	0	1	2	3	4	5
Character ->	H	e	l	l	o	/0

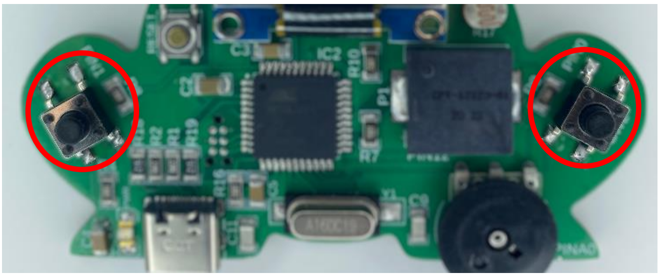
Once the message is stored into the **array**, it is printed.

```
  Serial.print("Message sent: "); Serial.println(message);
}
////////////////////////////////////////
```

To send data to the MC Trainer you first have to open the **Serial** monitor. Once the monitor is open, the text box at the top of it can be used to send a message. All you need to do is type in a message and hit the enter button. The computer will then attempt to send the message over **Serial**.



Lesson 3: Using Buttons



A button, also known as a push-button or momentary switch, is a simple device that allows a user to interact with a piece of machinery or electronic device. Buttons complete or break an electrical circuit when pressed, allowing for control of an electrical system.

Project 3.00 Read Input

In this project you will be introduced to buttons. By the end of this project, you will know how to read the state of a button (on or off) and how to store that in a **bool** type variable.

There are two buttons on the MC Trainer we can use, SW1 and SW2. SW1 is on the left side of the board and SW2 is on the right side. The abbreviation “SW” stands for switch.

Project Code:

```
//////////////////////////////////////////////////
// 3.00 - Read Input
byte SW1 = 1;

void setup() {
  Serial.begin(9600);
  pinMode(SW1, INPUT);
}

void loop() {
  bool buttonState = digitalRead(SW1);
  Serial.print("The state of the button is: "); Serial.println(buttonState);
  delay(250);
}
//////////////////////////////////////////////////
```

*If you’re copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

SW1 is connected to pin 1 so we first create a variable to represent our button in code:

```
byte SW1 = 1;
```

Next, in **setup()**, we initialize **Serial** so we can get feedback from our microcontroller:

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

Then because we are trying to read something from the outside world (a button press), SW1 is an input.

```
  pinMode(SW1, INPUT);
```

If you think about a button, it is either pressed or not pressed, there is no in-between. In similar terms, it is HIGH or LOW, ON or OFF, TRUE or FALSE, 1 or 0. A **bool** type variable is strictly meant to hold values like this, 1 or 0. So to store our buttons current state we will use a **bool** type variable.

```
  bool buttonState
```

The **digitalRead** function can be used to read the digital state of a pin. This function will return a 1 if the voltage is HIGH (the button is not pressed) and a 0 if the voltage is LOW (the button is pressed).

```
  bool buttonState = digitalRead(SW1);
```

Once read, the state of the button is printed out to the **Serial** port. Make sure to open it so you can see what is being printed!

```
  Serial.print("The state of the button is: "); Serial.println(buttonState);
```

Lastly, there is a **delay** so thousands of lines don't print out at once and the **loop**() is ended.

```
  delay(250);
}
```

Press SW1 and watch the button state change in the **Serial** monitor!

Now that we can read the button state, we can combine that with programmatic logic to make decisions on what to do when the button is pressed!

Project 3.01 Blink an LED with a Button

In the last project we learned how to read the state of a button. In this project we will learn how to make a decision based on the buttons state that we read. The goal of this project is to turn on an LED when the button is pressed!

Project Code:

```
////////////////////////////////////////////////////////
// 3.01 - Blink an LED with button press

byte SW1 = 1;
byte LED1 = 13;
bool pressed = 0;

void setup() {
  pinMode(LED1, OUTPUT);
  pinMode(SW1, INPUT);
}

void loop() {
  bool buttonState = digitalRead(SW1);

  //digitalWrite(LED1, !buttonState);

  if (buttonState == pressed) {
    digitalWrite(LED1, HIGH);
  }
  else {
    digitalWrite(LED1, LOW);
  }
}
////////////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

The pressed variable makes the code easier to read. A good programmer makes their code easier for others and themselves to read. By adding the pressed variable, it is easy to see when is going on in the code.

```
bool pressed = 0;
```

Here we are using an **if-statement** to make a decision. If the button is pressed, turn the LED on. Else, turn the LED off. The “==” is used to make a comparison to see if something is equal. Using the “=” operator will assign a value to the variable. This is a very common mistake in programming!

```
  if (buttonState == pressed) {
    digitalWrite(LED1, HIGH);
  }
  else {
    digitalWrite(LED1, LOW);
  }
```

Like most things in programming, there is more than one way to do it! Below is a “one liner” that can be used to do the same thing. It **digitalWrite**s the opposite of what was read from the **digitalRead** function. “!” is the **NOT** operator. When applied, this operator will turn a 1 into a 0 and a 0 into a 1. This inversion is because the button reads a 0 when pressed and a 1 when not pressed. So, when the button is pressed LED1 is written a 1, and when the button is not pressed LED1 is written a 0 with the inversion.

```
  digitalWrite(LED1, !buttonState);
```

Try to make a different LED turn on with SW2!

Project 3.02 AND Logic

In this project you'll learn about using **AND** logic with buttons to turn on LED1 only when both buttons are pressed.

What if you wanted to only turn on an LED when SW1 and SW2 were pressed at the same time? This is where **AND** logic comes into play. Our **if-statements** need to get a little more complicated to make this happen!

Project Code:

```
//////////////////////////////////////////////////////
// 3.02 – AND Logic

byte SW1 = 1;
byte SW2 = 0;
bool pressed = 0;

byte LED1 = 13;

void setup() {
  pinMode(SW1, INPUT);
  pinMode(SW2, INPUT);

  pinMode(LED1, OUTPUT);
}

void loop() {
  bool SW1State = digitalRead(SW1);
  bool SW2State = digitalRead(SW2);

  if (SW1State == pressed && SW2State == pressed) {
    digitalWrite(LED1, HIGH);
  }
  else {
    digitalWrite(LED1, LOW);
  }

}
//////////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

We know that we want to have an LED turn on when both buttons are pressed but how do we do it? Breaking down a problem into smaller, more manageable chunks is a priceless skill in programming. That being said, let's break this down.

First, we need to create the variables to represent our button pins, a variable to represent the pressed state, and a variable to represent our LED:

```
byte SW1 = 1;
byte SW2 = 0;
bool pressed = 0;

byte LED1 = 13;
```

Next, we need to set the correct **pinMode**s for the pins we plan to use. Buttons are inputs and LEDs are outputs:

```
void setup() {
  pinMode(SW1, INPUT);
  pinMode(SW2, INPUT);

  pinMode(LED1, OUTPUT);
}
```

Now we need to create the **loop()** functions code. The first thing we need to know is if the buttons are being pressed. To do that we **digitalRead** the button pins and store them in **bool** type variables for later use:

```
void loop() {
  bool SW1State = digitalRead(SW1);
  bool SW2State = digitalRead(SW2);
```

Next, we need to make a decision. How do we make a decision? We need to use an **if-statement**. The decision we need to make is if <u>SW1</u> **AND** <u>SW2</u> are pressed.

```
  if (SW1State == pressed && SW2State == pressed) {
```

As promised, the conditional got a little more complicated. The “&&” symbols in the conditional are new. This is the programmatic way to say **AND**. The conditional will evaluate to be true if what is on the left side of the “&&” symbols and what is on the right side of the “&&” symbols are both true. The left **AND** right sides need to be true for the code to execute.

If true, it turns <u>LED1</u> on:

```
    digitalWrite(LED1, HIGH);
```

```
}
```

Else, if it is not true, it turns LED1 off:

```
else {
  digitalWrite(LED1, LOW);
}
```

AND actually has a very strict definition in programming. It can be represented by what is known as a “Truth Table”. It shows all the possible outputs based on the inputs and logic being applied. Here is the **AND** Logic table:

Input A	**Input B**	**Output**
0	0	0
0	1	0
1	0	0
1	1	1

As you can see, the output is only 1 if both inputs are 1. In other words, if input A **AND** input B are 1, the output is a 1.

For a more familiar example let’s make the table based on our program’s inputs:

SW1State == pressed	SW2State == pressed	Output
0	0	0
0	1	0
1	0	0
1	1	1

As you can see, the output is only a 1 if SW1State and SW2State equal 1; I.e., both buttons are pressed.

Project 3.03 OR Logic

In this project you'll learn about using **OR** logic with buttons to turn on LED1 when SW1 **OR** SW2 are pressed.

What if you wanted to turn on an LED when SW1 was pressed **OR** SW2 was pressed? **AND** logic won't work for this. We need to learn how to use **OR** logic to do this.

Project Code:

```
////////////////////////////////////////////////////////
// 3.03 - OR Logic
// "&&" is the logical "and" comparator. See truth table below.

byte SW1 = 1;
byte SW2 = 0;
bool pressed = 0;

byte LED1 = 13;

void setup() {
  pinMode(SW1, INPUT);
  pinMode(SW2, INPUT);

  pinMode(LED1, OUTPUT);
}

void loop() {
  bool SW1State = digitalRead(SW1);
  bool SW2State = digitalRead(SW2);

  if (SW1State == pressed || SW2State == pressed) {
    digitalWrite(LED1, HIGH);
  }
  else {
    digitalWrite(LED1, LOW);
  }

}
////////////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

All of the code up to the **if-statement** is the same as the last project, so we'll skip to the difference.
The difference between this project and the last project is instead of having the "&&" symbols in the conditional, we now have the "||" symbols. This is the programmatic way to say **OR**.

```
 if (SW1State == pressed || SW2State == pressed) {
```

The conditional will evaluate to be true if what is on the left side of the "||" symbols or what is on the right side of the "||" symbols is true. The left **OR** right side need to be true for the code to execute.

If true, it turns LED1 on:

```
   digitalWrite(LED1, HIGH);
 }
```

Else, if it is not true, it turns LED1 off:

```
 else {
   digitalWrite(LED1, LOW);
 }
```

OR has a similarly strict definition in programming. Here is the **OR** logic truth table:

Input A	**Input B**	**Output**
0	0	0
0	1	1
1	0	1
1	1	1

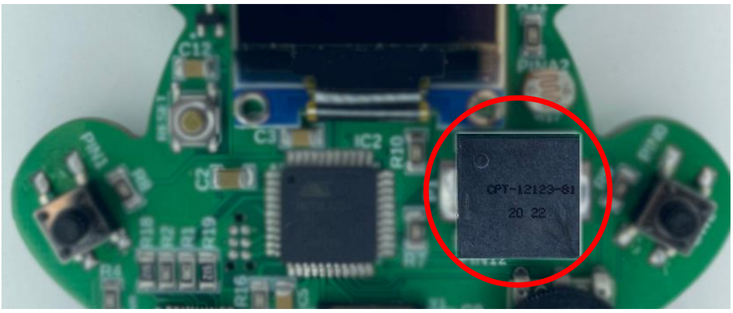
As you can see, the output is 1 if either input is 1. In other words, if input A **OR** input B is 1, the output is a 1.

Again, for a more familiar example let's make the table based on our program's inputs:

SW1State == pressed	SW2State == pressed	Output
0	0	0
0	1	1
1	0	1
1	1	1

As you can see, the output is a 1 if SW1State or SW2State equal 1; I.e., either button is pressed.

Lesson 4: Using the Piezo



A piezoelectric buzzer, or piezo buzzer, is a small, simple electronic device that produces sound based on the piezoelectric effect, a phenomenon where certain materials generate an electric charge when mechanical stress is applied. The core component of a piezo buzzer is a piezoelectric element—often a ceramic disc—placed between two conductive plates. When an oscillating electric signal is applied, the disc expands and contracts rapidly, creating sound waves that produce a buzzing noise. The frequency of the electric signal determines the pitch of the sound. These buzzers are found in a variety of devices like alarm clocks, microwave ovens, smoke detectors, and toys due to their compact size, reliability, and low power consumption.

Project 4.00 Using the Piezo

This project shows how to use the MC Trainer to produce a noise from the piezo onboard when SW1 is pressed.

Project Code:

```
/////////////////////////////////////////
// 4.00 - Using the Piezo

byte piezoPin = 12;

byte SW1 = 1;
bool pressed = LOW;

void setup() {
  pinMode(piezoPin, OUTPUT);
  pinMode(SW1, INPUT);
}

void loop() {
  if (digitalRead(SW1) == pressed) {
    digitalWrite(piezoPin, HIGH);
    delay(1);
    digitalWrite(piezoPin, LOW);
    delay(1);
  }
}
////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

The piezo is connected to pin 12 on the MC Trainer.

```
byte piezoPin = 12;
```

Since we are writing to this pin, it's an output.

```
  pinMode(piezoPin, OUTPUT);
```

The piezo buzzer creates sound through the rapid physical deformation of an internal crystal. When a voltage is applied, the crystal changes shape. By quickly fluctuating the applied voltage, a sound is generated with a frequency that matches the rate of these voltage changes. Since we know how to turn a pin on and off (**digitalWrite**) we can create noise! To prevent a continuous annoying buzzing, we will only write to the Piezo when SW1 is pressed.

```
  if (digitalRead(SW1) == pressed) {
    digitalWrite(piezoPin, HIGH);
    delay(1);
    digitalWrite(piezoPin, LOW);
    delay(1);
  }
```

Frequency, expressed in Hertz (Hz), indicates the number of cycles that occur per second. In the context of a piezo buzzer, understanding the frequency requires identifying the number of these cycles within a one-second timeframe. A "cycle", in this case, is defined as one complete sequence of the buzzer turning "on" and then "off". This sequence is also referred to as a "period", marking the duration of one full cycle of operation.

In our case the period is 2 milliseconds. It takes 2 milliseconds to make a full cycle. How many of those cycles happen in one second? Remember, 1 second is 1000ms.

1000ms / 2ms = 500Hz

The noise you hear is at a frequency of 500Hz!

Project 4.01 Using Functions

At this point in our programming journey, we need to be using functions in our programs. Functions make programming much easier to read and write. They make doing the same thing repeatedly much easier. This program is actually the exact same as the last but put into a function.

Project Code:

```
///////////////////////////////////////
// 4.01 - Using functions

byte piezoPin = 12;

byte SW1 = 1;
bool pressed = LOW;


void setup() {
  pinMode(piezoPin, OUTPUT);
  pinMode(SW1, INPUT);
}

void loop() {
  if (digitalRead(SW1) == pressed) {
    BuzzPiezo();
  }
}

/*
  It is good practice to put a function description here.
  Example:
  This function applies alternating voltage to the Piezo speaker
    at an period of 2ms.
*/
void BuzzPiezo() {
  digitalWrite(piezoPin, HIGH);
  delay(1);
  digitalWrite(piezoPin, LOW);
  delay(1);
}
//////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

This time in the loop instead of a bunch of code you'll only see a few lines. This line calls the function **BuzzPiezo**. Inside of this function is the code from the **loop()** function from the last program. The program just calls this function over and over again in the **loop()** if the SW1 is pressed.

```
void loop() {
  if (digitalRead(SW1) == pressed) {
    BuzzPiezo();
  }
}
```

When making a function it is good practice to put a function description right above it in a comment. This will remind you or tell someone else what the function does. Using the "/* */" format here will allow you to easily write a multi-line comment.

```
/*
  It is good practice to put a function description here.
  Example:
  This function applies alternating voltage to the Piezo speaker
    at an period of 2ms.
*/
```

Remember a function has a few parts. The return type, function name, parameters if it takes any, and the code inside of the brackets. See the table below for a breakdown of the **BuzzPiezo** function:

Part Name	Part
Return Type	Void (Void means it does not return anything)
Function Name	**BuzzPiezo**
Parameters	None in this case

Code inside brackets	digitalWrite(piezoPin, HIGH);
	delay(1);
	digitalWrite(piezoPin, LOW);
	delay(1);

```
void BuzzPiezo() {
  digitalWrite(piezoPin, HIGH);
  delay(1);
  digitalWrite(piezoPin, LOW);
  delay(1);
}
```

Project 4.02 Generating a Specific Tone

In this project you will learn how to write a function that will generate a specific tone.

Project Code:

```
////////////////////////////////////////
// 4.02 - Generating a Specific Tone

byte SW1 = 1;

bool pressed = LOW;

byte piezoPin = 12;

void setup() {
  pinMode(piezoPin, OUTPUT);
  pinMode(SW1, INPUT);
}

void loop() {
  if (digitalRead(SW1) == pressed) {
    BuzzPiezo(50);
  }
}

/*
  This function will generate a certain tone give a frequency in Hz.
*/
void BuzzPiezo(long frequency) {
  long period = 1000 / frequency;
  digitalWrite(piezoPin, HIGH);
  delay(period / 2);
  digitalWrite(piezoPin, LOW);
  delay(period / 2);
}
////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

The content of the **loop()** function is just like the last project except for this time the function takes a parameter (50). This parameter represents the frequency in Hz that we want the piezo to generate.

```
if (digitalRead(SW1) == pressed) {
  BuzzPiezo(50);
}
```

We are trying to create a certain tone, and to do that, we need to figure out something called a "period." But how do we find out what the period is from a frequency?

Well, a period is just the opposite (or reciprocal) of frequency. So, if we take 1 and divide it by the frequency, we get the period. However, since we're working with time in milliseconds, not seconds, we need to adjust our calculation. Instead of just 1 divided by the frequency, we use 1000 (because there are 1000 milliseconds in a second) divided by the frequency. That gives us our period in milliseconds!

```
long period = 1000 / frequency;
```

Remember that the period is the time for the piezo to turn on and off. This means that we have to **delay** by half the period after we turn it on and **delay** by half the period after we turn it off.

```
digitalWrite(piezoPin, HIGH);
delay(period / 2);
digitalWrite(piezoPin, LOW);
delay(period / 2);
```

We’ve now created a function to generate any frequency we want!

Project 4.03 Adding Duration

Building on the last project, in this project we will be adding duration to our function. This means that we can play different tones for different amounts of time.

Project Code:

```
////////////////////////////////////////
// 4.03 - Adding Duration

byte SW1 = 1;

bool pressed = LOW;

byte piezoPin = 12;

void setup() {
  pinMode(piezoPin, OUTPUT);
  pinMode(SW1, INPUT);
}

void loop() {
  if (digitalRead(SW1) == pressed) {
    //       Hz  ms
    BuzzPiezo(5, 1000);
    BuzzPiezo(50, 1000);
    BuzzPiezo(500, 1000);
  }
}

/*
  This function will generate a certain tone for a certain time
     given a frequency in Hz and a duration in ms.

*/
void BuzzPiezo(long frequency, long duration) {
  long period = 1000 / frequency;
  long cycles = duration / period;

  for (long i = 0; i < cycles; i++) {
    digitalWrite(piezoPin, HIGH);
    delay(period / 2);
    digitalWrite(piezoPin, LOW);
    delay(period / 2);
  }
}
////////////////////////////////////////
```
*If you’re copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

This time, the **BuzzPiezo** function is called with two parameters: the first one is the frequency and the second one is the duration of the tone in milliseconds.

```
void loop() {
  if (digitalRead(SW1) == pressed) {
    //        Hz  ms
    BuzzPiezo(5, 1000);
    BuzzPiezo(50, 1000);
    BuzzPiezo(500, 1000);
  }
}
```

Like before in the function, we have to know the period to generate the frequency. As a reminder, period is 1 / frequency but in our case is 1000 / frequency since we are working in milliseconds. Unlike last time we need the tone to happen for a certain amount of time. This means we have to know how many periods make up the desired time.

For example, If the frequency passed is 50Hz, and the time passed is 1000:

1000ms / 50Hz = 20ms (<- Period)

So how many 20ms cycles are there in 1000ms?

1000ms / 20ms = 50 cycles (<- How many times the period should happen)

So, we know that the period is 20ms and that period needs to happen 50 times. This sounds like the perfect opportunity for a **for-loop**!

```
  for (long i = 0; i < cycles; i++) {
    digitalWrite(piezoPin, HIGH);
    delay(period / 2);
    digitalWrite(piezoPin, LOW);
    delay(period / 2);
  }
```

We are basically saying play a 50Hz tone 50 times.

Project 4.04 There is a Library for that

In this project we will learn about the benefits of using a library to save development time and frustration.

Project Code:

```
////////////////////////////////////////////////
// 4.04 - Using The Piezo

byte piezoPin = 12;

byte SW1 = 1;
bool pressed = LOW;

void setup() {
  pinMode(piezoPin, OUTPUT);
  pinMode(SW1, INPUT);

}

void loop() {
  if (digitalRead(SW1) == pressed) {
    tone(piezoPin, 500, 100);
  }
}
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

What is a library? A library is a set of pre-made functions written by yourself or other people. Adding them into your program will provide additional functionality and cut down on development time significantly. They make programming much more manageable!

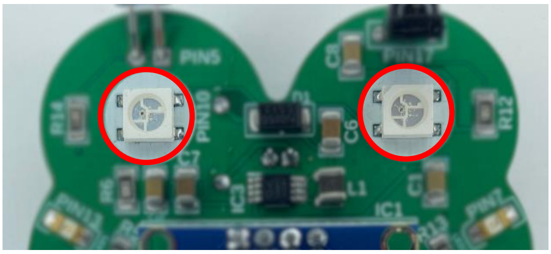
The library we use in the example is the "tone" library. This library is automatically included in Arduino, so you don't have to include any headers. We'll talk about what headers are later.

All we have to do to use it is call the **tone** function. See the table below for parameters:

Function Name	Parameter 1	Parameter 2	Parameter 3
tone	Pin number that the piezo is connected to	Desired frequency	Tone duration

This is almost exactly the function we made over the course of the last few sketches! By using this built-in library, we could have saved a lot of time and complications! This is the value of using libraries. Unless you need to, **try not to re-invent the wheel**. Search for a library for the part you are using. Many times, having a library or not is a huge factor in picking components.

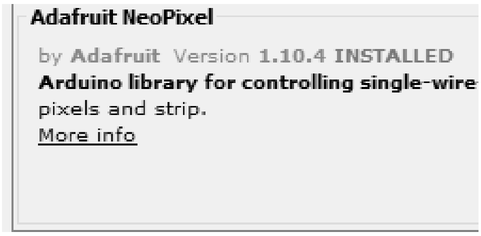
Lesson 5: Using Neopixels



Neopixels are digital RGB (Red, Green, Blue) LED pixels that are individually addressable. The Neopixels are the eyes of the MC Trainer.

Project 5.00 Using the Neopixels

In this project, we'll turn the Neopixels purple. We are going to use a library for this project. Most libraries are not built-in to Arduino. To add this library, go to tools -> Manage Libraries and type in “Adafruit Neopixel”. The library you're looking for looks like this:



Adafruit NeoPixel

by **Adafruit** Version **1.10.4 INSTALLED**
Arduino library for controlling single-wire
pixels and strip.
More info

Next, click “Install”.

Project Code:

```
////////////////////////////////////////////////////////////////
// 5.00 - Using The NeoPixel

#include <Adafruit_NeoPixel.h>

byte dataPin = 10;
byte numberOfPixels = 2;
byte brightness = 10;

byte redValue = 255;
byte greenValue = 0;
byte blueValue = 127;

Adafruit_NeoPixel pixels(numberOfPixels, dataPin, NEO_GRB + NEO_KHZ800);
void setup() {
  pixels.begin();
  pixels.setBrightness(brightness);

  pixels.setPixelColor(0, pixels.Color(redValue, greenValue, blueValue));
  pixels.setPixelColor(1, pixels.Color(redValue, greenValue, blueValue));

  pixels.show();
}

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

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

In order to use a library that is not built-in to Arduino, a library header needs to be added. This tells the program to compile that library along with the code you've written.

This does not need done for this sketch (unless you're typing from scratch), as it's already been included. For reference, this can be done by going to sketch -> include library -> Adafruit Neopixel. This will add the following line to your program:

```
#include <Adafruit_NeoPixel.h>
```

Six **byte** variables are used in this program: <u>dataPin</u> represents the pin that is connected to the first Neopixel, numberOfPixels represents the number of Neopixels connected to the board, <u>brightness</u> represents the max brightness of the Neopixels, <u>redValue</u> <u>greenValue</u> <u>blueValue</u> respectively represent their portion of the Neopixel light being emitted.

```
  byte dataPin = 10
  byte numberOfPixels = 2;
  byte brightness = 10;

  byte redValue = 255;
  byte greenValue = 0;
  byte blueValue = 127;
```

After the variable declarations, there is what's called a constructor. Constructors are used to set things up in many libraries, but not all. It's not important to understand constructors right now, but just know they create an object that we can use with the library. This constructor needs to know how many Neopixels are connected in a row, what pin they are connected to, and what type of Neopixels they are.

```
  Adafruit_NeoPixel pixels(numberOfPixels, dataPin, NEO_GRB + NEO_KHZ800);
```

For now you can think of "Adafruit_Neopixel" in the constructor as a sort of variable type like **int**, **float**, **char**, etc... "pixels" is just a common name to represent the object initialized by the constructor. It could be any name, like a normal variable. In contrast to a normal variable, objects are used with the "." (dot) operator. This dot operator allows you to access functions specific to that object.

In the **setup()** function, we first initialize the Neopixel strip and set its brightness.

```
    pixels.begin();
    pixels.setBrightness(brightness);
```

Next, we load the color of the neopixels using **setPixelColor** and **Color**. The **setPixelColor** function takes the pixel number, and a color. This second color parameter is passed with the **Color** function. This function allows us to pass an RGB value to the **setPixelColor** function. The color values passed to **Color** are 8-bit values (0-255).

```
    pixels.setPixelColor(0, pixels.Color(redValue, greenValue, blueValue));
    pixels.setPixelColor(1, pixels.Color(redValue, greenValue, blueValue));
```

Lastly, we need to update the Neopixels. In order to do so, all we need to do is call the **show()** function. This function needs to be called anytime that the Neopixel needs updated. When you set the pixel's color using **setPixelColor** that value is stored in program memory but not actually displayed. To display the new color, we need to call **pixels.show()**. If you only load the color but don't call this function, nothing will happen.

```
pixels.show(); //<- You must call this every time the Neopixel color needs updated.
```

You might be wondering how you know what functions you can access with your object. Usually there is a class reference that can be found online that details the library and the accompanying functions. This library class reference is located at:

https://adafruit.github.io/Adafruit_NeoPixel/html/class_adafruit___neo_pixel.html

You can also find examples of any library that you've installed by going to file -> examples and hovering over the library. This is a great way to learn how to set up and use new libraries and devices.

Project 5.01 Cycling Colors with a Button

The goal of this project is to change the color of the Neopixels every time SW1 is pressed. The colors will cycle between red, green and blue.

Project Code:

```
//////////////////////////////////////////////////
// 5.01 Cycling Colors With a Button

#include <Adafruit_NeoPixel.h>

byte dataPin = 10;
byte numberOfPixels = 2;
byte brightness = 10;
byte LEDSetting = 0;

byte SW1 = 1;
byte buttonState = 0;
bool pressed = 0;

Adafruit_NeoPixel pixels(numberOfPixels, dataPin, NEO_GRB + NEO_KHZ800);
void setup() {
  pinMode(SW1, INPUT);

  pixels.begin();
  pixels.setBrightness(brightness);
  pixels.show();
}

void loop() {
  buttonState = digitalRead(SW1);

  if (buttonState == pressed) {
    LEDSetting++;

    if (LEDSetting > 2) {
      LEDSetting = 0;
    }

    if (LEDSetting == 0) {
      pixels.setPixelColor(0, pixels.Color(255, 0, 0));
      pixels.setPixelColor(1, pixels.Color(255, 0, 0));
    }
    else if (LEDSetting == 1) {
      pixels.setPixelColor(0, pixels.Color(0, 255, 0));
      pixels.setPixelColor(1, pixels.Color(0, 255, 0));

    }
    else {
      pixels.setPixelColor(0, pixels.Color(0, 0, 255));
      pixels.setPixelColor(1, pixels.Color(0, 0, 255));

    }
    pixels.show();

    delay(250);
  }
}
////////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

In the **loop()** function, the first thing we do is check to see if a SW1 is being pressed.

```
  buttonState = digitalRead(SW1);

  if (buttonState == pressed) {
```

In this example a variable called LEDSetting is used to keep track of which color should be displayed. Every time the button is pressed, this variable is incremented by 1.

```
  LEDSetting++;
```

After incrementing that variable, it needs to be checked to see if it is out of the settings range (0 - 2). If it is, it is reset to 0.

```
  if (LEDSetting > 2) {
    LEDSetting = 0;
  }
```

Next, the appropriate colors are loaded into the Neopixels based on the LEDSetting variable according to the table below:

LEDSetting Value	Neopixel Color
0	Red
1	Green
2	Blue

```
  if (LEDSetting == 0) {
    pixels.setPixelColor(0, pixels.Color(255, 0, 0));
    pixels.setPixelColor(1, pixels.Color(255, 0, 0));
  }
  else if (LEDSetting == 1) {
    pixels.setPixelColor(0, pixels.Color(0, 255, 0));
    pixels.setPixelColor(1, pixels.Color(0, 255, 0));

  }
  else {
    pixels.setPixelColor(0, pixels.Color(0, 0, 255));
    pixels.setPixelColor(1, pixels.Color(0, 0, 255));

  }
```

To display the new color we have to update the Neopixels using the **show()** function.

```
pixels.show();
```

Lastly, a crude debounce is used to prevent more than one press being read at a time.

```
  delay(250);
 }
}
//////////////////////////////////////////////////
```

Project 5.02 Using ColorHSV

There are multiple ways to set the color of the Neopixels. Until now we have used the **Color** function to set the RGB color of the Neopixel. We can also use the function **ColorHSV** to set the color a different way. HSV stands for Hue, Saturation, and Value. Hue refers to the color of the Neopixel, Saturation refers to how white the Neopixel is, and Value refers to how bright the Neopixel is. Hue in this case is a 16-bit value (0 - 65535), while Saturation and Value are 8-bit values (0 - 255). Which color method you use is totally up to you and mostly comes down to preference and application.

Note: Using ColorHSV does make it easier to loop through the entire color spectrum.

Project Code:

```
///////////////////////////////////////////////////////
// 5.02 - Using ColorHSV

#include <Adafruit_NeoPixel.h>

byte saturation = 255;
byte value = 255;

byte dataPin = 10;
byte numberOfLEDs = 2;
byte brightness = 10;

Adafruit_NeoPixel pixels(numberOfLEDs, dataPin, NEO_GRB + NEO_KHZ800);
void setup() {
  Serial.begin(9600);
  pixels.begin();
  pixels.setBrightness(brightness);
}

void loop() {
  for (long hue = 0; hue < 65536; hue++) {
    pixels.setPixelColor(0, pixels.ColorHSV(hue, saturation, value));
    pixels.setPixelColor(1, pixels.ColorHSV(hue, saturation, value));
    pixels.show();
    delayMicroseconds(3);
  }

}
///////////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

Two variables are used to set the saturation and value parameters. These values are not changed throughout the course of the program. Feel free to change these values and see how it effects the program!

```
byte saturation = 255;
byte value = 255;
```

In the loop section of the program, all possible hue values are looped through and displayed. Remember, hue is represented by a 16-bit value (0 - 65535).

```
void loop() {
  for (long hue = 0; hue < 65536; hue++) {
    pixels.setPixelColor(0, pixels.ColorHSV(hue, saturation, value));
    pixels.setPixelColor(1, pixels.ColorHSV(hue, saturation, value));
```

Every time the hue value is changed, the Neopixels are updated to provide a smooth transition between colors.

```
    pixels.show();
```

A small **delay** is incorporated to slow down the color cycling. This **delay** function uses microseconds to **delay** instead of milliseconds. 1 second is 1000 milliseconds and 1 millisecond is 1000 microseconds. That means that we are **delay**ing 3 millionths of a second here! We use microseconds becuase we want the color to transition smoothly. There are 65,536 colors being displayed in the **for-loop**. Using the regular **delay** function between each one would make it look choppy and make the full color cycle take too long. Change the **delayMicroseconds** to **delay** and see what happens!

```
    delayMicroseconds(3);
  }

}
///////////////////////////////////////////////////////
```

Project 5.03 Individually addressing Neopixels using buttons

The goal of this project is to control the left Neopixel with SW1 and the right Neopixel with SW2. The left Neopixel will turn on when SW1 Is pressed, and the right Neopixel will turn on when SW2 is pressed. They will turn off when their respective buttons are not pressed.

Project Code:

```
///////////////////////////////////////////////////////
// 5.03 – Individually addressing Neopixels using buttons

#include <Adafruit_NeoPixel.h>

byte dataPin = 10;
byte numberOfPixels = 2;

byte SW1 = 1;
byte SW2 = 0;

bool pressed = LOW;

byte saturation = 255;
long neoPixelOneColor = 0;
long neoPixelTwoColor = 45000;

Adafruit_NeoPixel pixels(numberOfPixels, dataPin, NEO_GRB + NEO_KHZ800);
void setup() {
  pixels.begin();

  pinMode(SW1, INPUT);
  pinMode(SW2, INPUT);
}

void loop() {


  if (digitalRead(SW1) == pressed) {
    pixels.setPixelColor(0, pixels.ColorHSV(neoPixelOneColor, saturation, 255));
  }
  else {
    pixels.setPixelColor(0, pixels.ColorHSV(neoPixelOneColor, saturation, 0));
  }

  if (digitalRead(SW2) == pressed) {
    pixels.setPixelColor(1, pixels.ColorHSV(neoPixelTwoColor, saturation, 255));

  }
  else {
    pixels.setPixelColor(1, pixels.ColorHSV(neoPixelTwoColor, saturation, 0));
  }

  pixels.show(); // Show the Neopixel's color

}
////////////////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

Like previous sketches, we want to make a decision based on if the buttons are pressed or not. In order to do so, we need to use **if-statements**.

```
  if (digitalRead(SW1) == pressed) {
```

Next, we need to add the code that turns on the Left Neopixel if SW1 is pressed.

```
    pixels.setPixelColor(0, pixels.ColorHSV(neoPixelOneColor, saturation, 255));
```

Let's break this function down again. In the **setPixelColor** function there are two parameters. The first is the address of the Neopixel you'd like to target.

Neopixels are all chained together in one big line. Like most things in programming the line count starts at 0. The Neopixel on the left is first in line so it's Neopixel number 0. The Neopixel on the right is the second in line so it's Neopixel number 1.

So, in this case since we have passed a 0 as our first parameter in the **setPixelColor** function, we are addressing the Neopixel on the left.

```
    pixels.setPixelColor(0
```

The second parameter of **setPixelColor** is the pixel's color. In this case we use the **ColorHSV** function to set that.

The third parameter of the **ColorHSV** function is the most important here. Remember, It's the value of the color (brightness). If we set this to 255, the color is full brightness and if we set it to 0, the Neopixel is off. So, when SW1 is pressed we want to set the Neopixel to full brightness:

```
    pixels.setPixelColor(0, pixels.ColorHSV(neoPixelOneColor, saturation, 255));
```

```
}
```

And when the SW1 is not pressed we want to turn the Neopixel off:

```
  else {
    pixels.setPixelColor(0, pixels.ColorHSV(neoPixelOneColor, saturation, 0));
  }
```

The same process is repeated for the Neopixel on the right, but you'll notice that Neopixel number 1 is addressed instead:

```
  if (digitalRead(SW2) == pressed) {
    pixels.setPixelColor(1, pixels.ColorHSV(neoPixelTwoColor, saturation, 255));

  }
  else {
    pixels.setPixelColor(1, pixels.ColorHSV(neoPixelTwoColor, saturation, 0));
  }
```

Then to display these colors all we need to do is call the **show**() function:

```
  pixels.show(); // Show the Neopixel's color

}
////////////////////////////////////////////////////////
```

And the loop repeats.

To change the color that is being displayed, change the neoPixelOneColor and neoPixelTwoColor variables. This website will give you the HSV and RGB color codes for a desired color:

https://www.rapidtables.com/web/color/color-picker.html

Project 5.04 Fading Neopixels Using Buttons

This project is a bit more complicated than the last. In this project the goal is to slowly fade a Neopixel on while a button is pressed but instead of the Neopixel shutting off when the button is released, it slowly fades off. Again, SW1 will control the left Neopixel, and SW2 will control the right Neopixel.

Project Code:

```
////////////////////////////////////////////////////////
// 5.04 - Fading Neopixels Using Buttons

#include <Adafruit_NeoPixel.h>

byte dataPin = 10;
byte numberOfPixels = 2;
byte SW1 = 1;
byte SW2 = 0;

bool pressed = LOW;

byte saturation = 255;
long neoPixelOneColor = 0;
long neoPixelTwoColor = 45000;
byte neoPixelOneBrightness = 0;
byte neoPixelTwoBrightness = 0;

Adafruit_NeoPixel pixels(numberOfPixels, dataPin, NEO_GRB + NEO_KHZ800);
void setup() {

  pixels.begin();

  pinMode(SW1, INPUT);
  pinMode(SW2, INPUT);
}

void loop() {
  if (digitalRead(SW1) == pressed) {
    if (neoPixelOneBrightness < 255) {
      neoPixelOneBrightness = neoPixelOneBrightness + 1;

    }
  }
  else {
    if (neoPixelOneBrightness > 0) {
      neoPixelOneBrightness = neoPixelOneBrightness - 1;
    }
  }

  if (digitalRead(SW2) == pressed) {
    if (neoPixelTwoBrightness < 255) {
      neoPixelTwoBrightness = neoPixelTwoBrightness + 1;
    }
  }
  else {
    if (neoPixelTwoBrightness > 0) {
      neoPixelTwoBrightness = neoPixelTwoBrightness - 1;
    }
  }

  pixels.setPixelColor(0, pixels.ColorHSV(neoPixelOneColor, saturation,
neoPixelOneBrightness));
  pixels.setPixelColor(1, pixels.ColorHSV(neoPixelTwoColor, saturation,
neoPixelTwoBrightness));

  pixels.show();

  delay(10);
}
////////////////////////////////////////////////////////
```

*If you’re copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

This sketch introduces two new **byte** type variables from the last. These two variables are used to keep track of the individual brightness of each Neopixel:

```
byte neoPixelOneBrightness = 0;
byte neoPixelTwoBrightness = 0;
```

Moving on to the **loop()**, we once again need to make a decision. Is the button pressed? For this we will use an **if-statement**:

```
  if (digitalRead(SW1) == pressed) {
```

We can make increasingly complex decisions with an **if-statement** inside another **if-statement**. In this example we set the color using **ColorHSV**. This makes it very easy for us to set the brightness of an individual pixel. Remember that the last parameter for **ColorHSV** is the “Value” of the Neopixel (brightness). It is important to note that this function takes 8-bit number for value, so it’s numbers between 0-255. If we pass a value over 255 unexpected things can happen. For that reason, we have to check and see if the brightness value for this pixel is less than 255 before we increase the brightness any.

```
    if (neoPixelOneBrightness < 255) {
```

If the button is pressed, we need to increase the brightness of the Neopixel associated with that button (if it is below 255). In this example we increase the brightness variable by one every time that the **loop**() runs and the button is found to be pressed.

```
      neoPixelOneBrightness = neoPixelOneBrightness + 1;
```

If the associated button is not pressed, the **else** part of the **if-statement** executes. Along with not being able to pass numbers above 255 for the value parameter of **ColorHSV** we cannot pass numbers below 0 for the value parameter. It doesn't make sense to have a brightness below 0. For that reason, we have to check and see that the associated brightness variable is greater than 0 before subtracting 1:

```
    if (neoPixelOneBrightness > 0) {
```

If it is above 0 we can subtract 1 from it:

```
      neoPixelOneBrightness = neoPixelOneBrightness - 1;
```

The exact same thing happens with the other button and Neopixel:

```
  if (digitalRead(SW2) == pressed) {
    if (neoPixelTwoBrightness < 255) {
      neoPixelTwoBrightness = neoPixelTwoBrightness + 1;
    }
  }
  else {
    if (neoPixelTwoBrightness > 0) {
      neoPixelTwoBrightness = neoPixelTwoBrightness - 1;
    }
  }
```

Next, we need to update the Neopixels with their associated brightness values. Remember that the first Neopixel is number 0 and the second Neopixel is number 1. All we have to do is address the Neopixels and pass the new brightness values to **ColorHSV**.

```
  pixels.setPixelColor(0, pixels.ColorHSV(neoPixelOneColor, saturation,
neoPixelOneBrightness));

  pixels.setPixelColor(1, pixels.ColorHSV(neoPixelTwoColor, saturation,
neoPixelTwoBrightness));
```

After we update the Neopixels we have to show the new brightness:

```
  pixels.show();
```

Lastly, there is a delay added so that we can see the incremental changes. Try deleting this delay and see what happens!

```
  delay(10);
}
/////////////////////////////////////////////////////
```

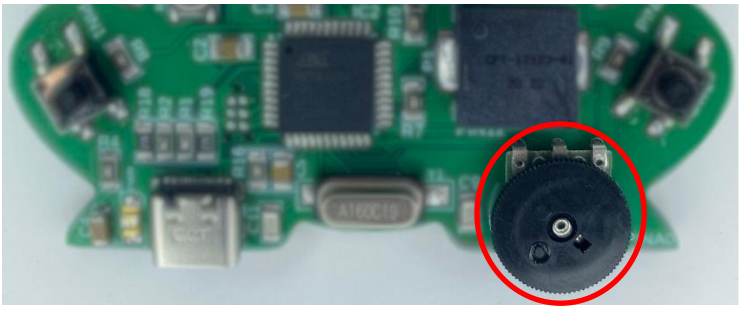
Let’s do a brief recap.

First, we checked to see if a button was pressed. If the button was pressed, we checked to see if the brightness associated with the Neopixel was below 255. If it was, we increased its brightness by 1.

If the button was not pressed, we checked to see if the brightness of the associated Neopixel was greater than 0. If it was, we decreased its brightness by 1.

Next, we updated the Neopixels with the correct brightness variables. Then showed the Neopixel updates and added a small delay.

Lesson 6: Using the Potentiometer



A potentiometer, often referred to as a "pot," is a type of variable resistor. They are commonly used for controlling electrical devices such as volume controls on audio equipment or as control inputs in many types of electronic circuits. It's a three-terminal device that operates as an adjustable voltage divider.

Project 6.00 Using the Potentiometer

This project introduces the potentiometer and shows how to use it. A potentiometer is a special type of resistor that allows us to change its resistance value. This feature makes potentiometers very useful for tasks like adjusting the volume on speakers or controlling the brightness of lights, where we need to vary resistance to control an electrical current.

Project Code:

```
////////////////////////////////////////////////////////////
// 6.00 - Using The Potentiometer

byte potPin = A0;

void setup() {
  Serial.begin(9600);
  pinMode(potPin, INPUT);
}

void loop() {
 int potValue = analogRead(potPin);

 Serial.print("The pot value is at: "); Serial.println(potValue);
 delay(100);
}
/////////////////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

First, we declare the pin that the potentiometer is connected to as potPin. This variable declaration is different from those you've seen so far because it starts with an "A". This "A" means that the pin we are connecting to is an analog pin. Analog pin refers to a pin that can read analog values. A digital value would be a 1 or a 0 (5v or 0v), but an analog value can be anything in between. Not all pins can be used as analog pins on the MC Trainer.

```
byte potPin = A0;
```

In the setup you'll notice that the pin is still an **INPUT** pin. This is because we are using the pin to read a value. It does not matter if it is digital or analog, it is still an input.

```
 pinMode(potPin, INPUT);
```

You can read the analog voltage at an analog pin by using the **analogRead** function. **analogRead** returns a 10-bit value (0 – 1023) that represents the voltage at that pin. 5v would be 1023, ~2.5v would be 512, and 0v would be 0.

```
int potValue = analogRead(potPin);
```

After that we print the value read out into the **Serial** port and **delay** 100 milliseconds. Make sure and open the **Serial** port to see the data being printed.

```
Serial.print("The pot value is at: "); Serial.println(potValue);
delay(100);
```

Try rotating the knob of the potentiometer to see the range of values produced!

Project 6.01 Changing Color with the Potentiometer

This project will demonstrate the use of a potentiometer to control the color of the Neopixels. As the potentiometer rotates, the color of the Neopixels will change.

Project Code:

```
////////////////////////////////////////////////////
// 6.01 - Changing Color with Potentiometer

#include <Adafruit_NeoPixel.h>

byte value = 255;
byte saturation = 255;

byte dataPin = 10;
byte numberOfPixels = 2;
byte brightness = 10;

byte potPin = A0;

Adafruit_NeoPixel pixels(numberOfPixels, dataPin, NEO_GRB + NEO_KHZ800);
void setup() {
  Serial.begin(9600);

  pinMode(potPin, INPUT);

  pixels.begin();
  pixels.setBrightness(brightness);
}

void loop() {

  int potValue = analogRead(potPin);
  long colorValue = map(potValue, 0, 1023, 0, 65535);

  pixels.setPixelColor(0, pixels.ColorHSV(colorValue, saturation, value));
  pixels.setPixelColor(1, pixels.ColorHSV(colorValue, saturation, value));
  pixels.show();

}
////////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

After declaring variables, in the **setup()** function **Serial** is initialized, the **pinMode** of the <u>potPin</u> is set, the Neopixels are initialized and their brightness is set.

```
void setup() {
  Serial.begin(9600);

  pinMode(potPin, INPUT);

  pixels.begin();
  pixels.setBrightness(brightness);
}
```

In the **loop()** function, an **int** variable called potValue is created to hold the analog value read from the pot.

```
int potValue = analogRead(potPin);
```

At this point we have a problem. We have a potentiometer value somewhere between 0 and 1023 (10-bit) but our color range goes from 0 – 65535 (16-bit). This is where the **map** function comes in. The **map** function allows us to map two value ranges together linearly. This is much less scary than it sounds. Essentially, it will make 0 on the potentiometer equal 0 on the color range, and 1023 on the potentiometer equal 65535 on the color range. For example, if the potentiometer value read was 543 the map function would return a value of 34785.

We can check this by using the equation:

1.) (potValue * MaxHueValue) / MaxPotValue = MappedValue
2.) (543 * 65535)/1023 = 34,785

The **map** function takes five parameters. It takes the value to map, the min value of that range, the max value of that range, the min value of the range to be mapped to, and the max value of the range to be mapped to. See the table below:

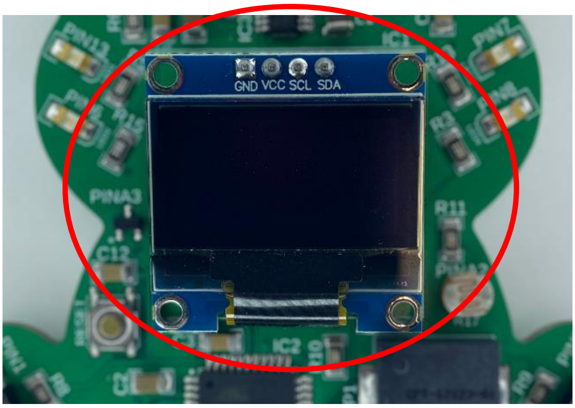
Parameters	Value
Value to map	**analogValue** read from pot
Min value of that range	0
Max value of that range	1023
Max value of the range to be mapped to	0
Min value of the range to be mapped to	65535

```
long colorValue = map(potValue, 0, 1023, 0, 65535);
```

Lastly, the value that was mapped is written to the Neopixels and shown.

```
pixels.setPixelColor(0, pixels.ColorHSV(colorValue, saturation, value));
pixels.setPixelColor(1, pixels.ColorHSV(colorValue, saturation, value));
pixels.show();
```

Lesson 7: Using the OLED



OLED stands for organic light emitting diode. This small screen allows for real-time feedback and interaction with projects.

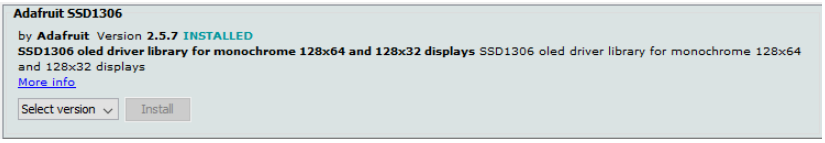
Project 7.00 Using the OLED

The OLED is like a mini-TV screen. In fact, many TVs now-a-days are OLEDs. In this project you'll learn how to setup and use the OLED properly.

Project Code:

```
////////////////////////////////////////////////////////////
// 7.00 - Using The OLED

#include <Adafruit_SSD1306.h>
#include <splash.h>

byte screenWidth = 128;
byte screenHeight = 64;
byte screenAddress = 0x3C;

Adafruit_SSD1306 display(screenWidth, screenHeight, &Wire);
void setup() {
  display.begin(SSD1306_SWITCHCAPVCC, screenAddress);
  display.clearDisplay();
  display.display();

  display.setCursor(0, 0);
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
  display.println("Hello World!");
  display.display();
}

void loop() {
}
////////////////////////////////////////////////////////////
```
*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

Before we do anything, we need to install the OLED Library. You can do this by going to Tools -> Manage Libraries, and type in "Adafruit_SSD1306". It looks like this:


Adafruit SSD1306
by **Adafruit** Version **2.5.7** INSTALLED
SSD1306 oled driver library for monochrome 128x64 and 128x32 displays SSD1306 oled driver library for monochrome 128x64 and 128x32 displays
More info

Click install. If it asks you to install dependencies go ahead and do so.

For reference, you can include it into a project by going to Sketch -> Include Library and click "Adafruit_SSD1306". This does not need to be done in this sketch as it is already included.

The library reference where you can find all of the functions avaliable can be found here:

https://adafruit.github.io/Adafruit_SSD1306/html/class_adafruit___s_s_d1306.html

The first thing that we need to do is tell the define the length and width of our OLED in pixels.

```
byte screenWidth = 128;
byte screenHeight = 64;
```

Next, we have to define the I2C address or our OLED. I2C is a communication protocol that uses unique device addresses to communicate data. You do not need to understand anything about I2C other than the address for the OLED is 0x3C.

```
byte screenAddress = 0x3C;
```

What does 0x3C mean? Well, that number starts with “0x” because it is in hexadecimal. Hexadecimal is a base 16 numbering system that uses letters to represent numbers 10 – 15. These are letters A – F. Our day-to-day numbers are base 10 and are referred to as decimal numbers. 0x3C is 60 in decimal. You could write the address as 60 but I2C addresses are generally done in hex.

```
byte screenAddress = 0x3C;
```

Next is the constructor for the class. The constructor contains a name, the screen’s width, the screen’s height, and a reference to the Wire object. That last one is sort of complicated, so we won’t worry about it. Just know that you need the “&Wire” as the last parameter. To break that down into a table:

Symbol	What it is
display	This is the name of the class we are creating. This is just like the “pixels” in the Neopixel projects.
screenWidth	This is the screen’s width in pixels (128)
screenHeight	This is the screen’s height in pixels (64)
&Wire	This is a reference to the Wire class. Don’t worry about understanding this right now.

```
Adafruit_SSD1306 display(screenWidth, screenHeight, &Wire);
```

A few things need to be done in the **setup()** function. First, we have to initialize the OLED with the **begin** function. The “SSD1306_SWITCHCAPVCC” that is passed is just telling the OLED to create its own IO voltage. It’s basically creating a higher voltage than the MC Trainer can supply to support itself. We also have to pass the I2C address to it.

```
  display.begin(SSD1306_SWITCHCAPVCC, screenAddress);
```

Next, the screen is cleared with the **clearDisplay** function.

```
  display.clearDisplay();
```

Once loaded with data, the OLED needs to be told to display it with the **display** function. This works exactly the same as the **show** function with the Neopixels. In this case we’ve loaded the clear screen data.

```
  display.display();
```

Now we set the cursor to coordinates (0,0). This (0,0) is different from a normal graph. The (0,0) for the OLED starts in the top left corner.

```
  display.setCursor(0, 0);
```

Next, we will set the size of the text to be displayed.

```
  display.setTextSize(1);
```

Lastly for setup we have to tell the OLED what color the text is with the **setTextColor** function. This OLED can only display white.

```
  display.setTextColor(SSD1306_WHITE);
```

We are now ready to print some text! This works exactly the same as the **Serial** functions.

```
  display.println("Hello World!");
```

Once the data is loaded, we have to display it with the **display** function.

```
  display.display();
```

There is nothing in the **loop()** function in this project.

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

Project 7.01 Writing Text to the Screen

In this project you'll learn how to write text to the OLED, change its size, and about printing variables.

Project Code:

```
//////////////////////////////////////////////////////////////////////
// 7.01 - Writing Text to The Screen

#include <Adafruit_SSD1306.h>
#include <splash.h>

byte screenWidth = 128;
byte screenHeight = 64;
byte screenAddress = 0x3C;
byte numberToPrint = 123;

Adafruit_SSD1306 display(screenWidth, screenHeight, &Wire);

void setup() {
  display.begin(SSD1306_SWITCHCAPVCC, screenAddress);
  display.setTextColor(SSD1306_WHITE);
}

void loop() {
  display.clearDisplay();
  display.setCursor(0,0);

  display.setTextSize(3);

  display.println("Hello!");
  display.print(numberToPrint);
  display.display();
}
//////////////////////////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

First, we have to include the correct libraries, declare our standard variables, and include the constructor.

```
#include <Adafruit_SSD1306.h>
#include <splash.h>

byte screenWidth = 128;
byte screenHeight = 64;
byte screenAddress = 0x3C;
byte numberToPrint = 123;

Adafruit_SSD1306 display(screenWidth, screenHeight, &Wire);
```

In the **setup()** function, we need to set the OLED up. The only two things that you absolutely need to do are use the **begin** and **setTextColor** functions.

```
void setup() {
  display.begin(SSD1306_SWITCHCAPVCC, screenAddress);
  display.setTextColor(SSD1306_WHITE);
}
```

In the **loop()** function, the first thing we want to do is clear any text that is currently on the screen and reset the cursor to (0,0).

```
void loop() {
  display.clearDisplay();
  display.setCursor(0,0);
```

Next, we need to set the text size that we would like. Good options are 1-3, but you can go bigger.

```
  display.setTextSize(3);
```

Then we can print some text! First, we print “Hello!” and then we print a variable. You can use the **print** and **println** functions to print all of the same things that you can print using the **Serial** functions.

```
  display.println("Hello!");
  display.print(numberToPrint);
  display.display();
}
//////////////////////////////////////////////////////////////////////
```

Project 7.02 Reaction Game Using OLED

In this project we will make a reaction time game! The game will be able to keep track of the current record in milliseconds and will even be able to tell if the user cheated!

Project Code:

```
////////////////////////////////////////////////////////////////////////
// 7.02 - Reaction Game Using The OLED

#include <Adafruit_SSD1306.h>
#include <splash.h>

byte screenWidth = 128;
byte screenHeight = 64;
byte screenAddress = 0x3C;

byte LED1 = 13;

bool pressed = LOW;
byte SW1 = 1;
byte SW2 = 0;

int timesPlayed = 0;
long record = 0;

bool cheated = false;

Adafruit_SSD1306 display(screenWidth, screenHeight, &Wire);

void setup() {
  display.begin(SSD1306_SWITCHCAPVCC, screenAddress);
  display.setTextColor(SSD1306_WHITE);
  display.setTextSize(2);

  pinMode(LED1, OUTPUT);

  pinMode(SW1, INPUT);
  pinMode(SW2, INPUT);
}

void loop() {
  cheated = false;

  while (digitalRead(SW1) != pressed) {
    display.clearDisplay();
    display.setCursor(0, 0);
    display.println("Press SW1");
    display.println("To Play!");

    if (timesPlayed != 0) {
```

```
    display.println("Record: ");
    display.print(record);
    display.println(" ms");
  }
  display.display();
}

digitalWrite(LED1, HIGH);
int waitTime = random(1000, 4001);
long startWaitTime = millis();

while (waitTime + startWaitTime > millis()) {
  if (digitalRead(SW2) == pressed) {
    cheated = true;
    break;
  }
}

digitalWrite(LED1, LOW);

if (cheated != true) {
  long startTime = millis();

  display.clearDisplay();
  display.setCursor(0, 0);
  display.println("Press the ");
  display.println("button!");
  display.display();

  while (digitalRead(SW2) != pressed) {
    ;
  }

  long endTime = millis();
  long totalTime = endTime - startTime;

  if (timesPlayed == 0) {
    record = totalTime;
  }
  else if (totalTime < record) {
    record = totalTime;
  }

  display.clearDisplay();
  display.setCursor(0, 0);
  display.println("You took");
  display.print(totalTime);
  display.println(" ms");
  display.println("to react!");
  display.display();

  timesPlayed++;

}
```

```
  else {
    display.clearDisplay();
    display.setCursor(0, 0);
    display.println("You");
    display.println("cheated!");
    display.setTextSize(4);
    display.println(":(");
    display.setTextSize(2);
    display.display();
  }

  delay(3000);
}
////////////////////////////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

This program is a bit longer than the others, but don't worry it'll all be broken down.

Let's define our game:

Our game will test reaction time by turning on an LED for a random amount of time, having the player press a button when that LED turns off, timing how long after the LED turned off that the player pressed the button, and displaying that on a screen. It would also be nice if we could keep track of the best reaction time.

As a bonus project, see if you can get the Piezo to Buzz when the user cheats!

Let's break that down a little more:

- Use LED1 as the reaction LED
- Turn LED1 on for a random time between 1 and 4 seconds
- Use the left button to start the game
- Use the right button to react to the LED turning off
- Keep track of the time from when the LED turns off to when the button is pressed
- Display that time on the screen
- Keep track of the best reaction time

Now that we have defined the game, we can program it. First things first, we need to include the proper libraries to use the OLED.

```
#include <Adafruit_SSD1306.h>
```

```
#include <splash.h>
```

Next, we define the height, width, and address of the OLED.

```
byte screenWidth = 128;
byte screenHeight = 64;
byte screenAddress = 0x3C;
```

Then we need to declare some additional variables in our code to define things like the LED that will be used as the reaction indicator, the buttons that will be used, and some general variables that will be used to play the game.

```
byte screenWidth = 128;
byte screenHeight = 64;
byte screenAddress = 0x3C;

byte LED1 = 13;

bool pressed = LOW;
byte SW1 = 1;
byte SW2 = 0;
```

These three variables will be used to record the number of times the game has been played, the record and if the player has cheated.

```
int timesPlayed = 0;
long record = 0;

bool cheated = false;
```

Here we just include the constructor for the OLED.

```
Adafruit_SSD1306 display(screenWidth, screenHeight, &Wire);
```

In the **setup()** function we need to setup our **pinMode**s and OLED. The buttons will be **INPUT**s and the LED will be an **OUTPUT**.

```
void setup() {
  display.begin(SSD1306_SWITCHCAPVCC, screenAddress);
  display.setTextColor(SSD1306_WHITE);
  display.setTextSize(2);

  pinMode(LED1, OUTPUT);

  pinMode(SW1, INPUT);
  pinMode(SW2, INPUT);
}
```

Next in the **loop()** function, we first need to reset the cheated variable before we play the game.

```
  cheated = false;
```

Then we need to wait until the user presses the play button. We can wait by using a **while-loop**.

```
  while (digitalRead(SW1) != pressed) {
```

While the program waits for the user to press the button it can print the “Menu” screen. The menu screen prompts the player to play by pressing SW1.

```
    display.clearDisplay();
    display.setCursor(0, 0);
    display.println("Press SW1");
    display.println("To Play!");
```

This screen is also where the record will be printed. There is no point in printing a record on this screen if the game has not been played, because there is no record. So, by using the timesPlayed variable we can check if the game has been played. If so, print the record.

```
    if (timesPlayed != 0) {
      display.println("Record: ");
      display.print(record);
      display.println(" ms");
    }
```

Now that we have loaded the data into the OLED, we need to display it.

```
    display.display();
  }
```

Once the button has been pressed the program moves on from the **while-loop**.

At this point we know the player is ready to play the game so turn the LED on.

```
  digitalWrite(LED1, HIGH);
```

Next, we need to get a random time between 1 and 4 seconds to keep the LED on for. This can be done with the **random** function. It will return a random value between the first number passed and the second number passed minus 1.

```
  int waitTime = random(1000, 4001);
```

If we are going to wait for some time from now, then we need to know what time it is now. We can get that by using the **millis()** function.

```
  long startWaitTime = millis();
```

To wait we need to keep checking the time to see if it's waitTime past the startWaitTime. For example, if the startWaitTime was 1567 milliseconds and the waitTime was 3000 we need to wait until **millis()** returns 4567 to move on. startWaitTime + waitTime needs to be less than what **millis()** returns.

```
  while (waitTime + startWaitTime > millis()) {
```

While we are waiting for the time to elapse the LED is still on. If the player presses the button while the LED is on that is considered cheating. So, if the player presses the button while we are in this **while-loop** then they are cheating. We can tell if they press the button by monitoring it. In the loop we can put an if statement to check and see if the button is being pressed.

```
    if (digitalRead(SW2) == pressed) {
```

If the button is being pressed, we set the cheated variable to true.

```
cheated = true;
```

Then we use the keyword "**break**" to leave the **while-loop**. **break** can be used to break out of loops completely. There is no need to stay in the loop waiting if the player cheated, so we can move on.

```
      break;
    }
  }
```

Next, we can turn the LED off. At this point, the player has either cheated of the time has properly elapsed.

```
  digitalWrite(LED1, LOW);
```

After the LED is off, if the player did not cheat, we need to record the time that the LED turned off at.

```
 if (cheated != true) {
   long startTime = millis();
```

Then we need to prompt the user to press the button (because the LED is off).

```
   display.clearDisplay();
   display.setCursor(0, 0);
   display.println("Press the ");
   display.println("button!");
   display.display();
```

We then need to wait for the player to press SW2 which indicates a reaction.

```
   while (digitalRead(SW2) != pressed) {
     ;
   }
```

Once the player has pressed the button the **while-loop** breaks. We need to record that time with **millis()**.

```
   long endTime = millis();
```

To find out the total time it took from the LED turning off to when the player pressed the button, we have to subtract the time when the LED turned off from the time the player pressed the button.

```
   long totalTime = endTime - startTime;
```

Then we check to see if this is the first time the player has played.

```
   if (timesPlayed == 0) {
```

If it is the first time, we set the record equal to the totalTime.

```
 record = totalTime;
}
```

If it is not the first time, then we check to see if the totalTime is less than the current record.

```
 else if (totalTime < record) {
```

If it is, then that's the new record time.

```
      record = totalTime;
    }
```

We can then display the time the player took to react on the OLED.

```
    display.clearDisplay();
    display.setCursor(0, 0);
    display.println("You took");
    display.print(totalTime);
    display.println(" ms");
    display.println("to react!");
    display.display();
```

Then we need to increment the timesPlayed variable to indicate that it is not the first time playing the game later.

```
    timesPlayed++;
```

If the player cheated, we need to display that on the OLED.

```
  else {
    display.clearDisplay();
    display.setCursor(0, 0);
    display.println("You");
    display.println("cheated!");
    display.setTextSize(4);
    display.println(":(");
    display.setTextSize(2);
    display.display();
  }
```

After either case (cheated or not), we need to **delay** for three seconds to let the player see their time or see that they cheated.

```
  delay(3000);
```

Project 7.03 Drawing Shapes with the OLED

We can do more than just text on the OLED. In this project you'll learn how to draw shapes too!

Project Code:

```
////////////////////////////////////////////////////
// 7.03 Drawing Shapes with the OLED

#include <Adafruit_SSD1306.h>

byte screenWidth = 128;
byte screenHeight = 64;
byte screenAddress = 0x3C;

Adafruit_SSD1306 display(screenWidth, screenHeight, &Wire);
void setup() {
  display.begin(SSD1306_SWITCHCAPVCC, screenAddress);
  display.setTextColor(SSD1306_WHITE);
}

void loop() {
  display.clearDisplay();
  display.drawPixel(63, 32, WHITE);
  display.display();
  delay(1000);

  display.clearDisplay();
  display.drawLine(0, 0, 63, 63, WHITE);
  display.display();
  delay(1000);

  display.clearDisplay();
  display.drawCircle(63, 31, 12, WHITE);
  display.display();
  delay(1000);

  display.clearDisplay();
  display.fillCircle(63, 32, 12, WHITE);
  display.display();
  delay(1000);

  display.clearDisplay();
  display.drawTriangle(63, 0, 96, 63, 32, 63, WHITE);
  display.display();
  delay(1000);

  display.clearDisplay();
  display.fillTriangle(63, 0, 96, 63, 32, 63, WHITE);
  display.display();
  delay(1000);
  display.clearDisplay();
  display.drawRect(10, 10, 107, 43, WHITE);
  display.display();
  delay(1000);

  display.clearDisplay();
  display.fillRect(10, 10, 107, 43, WHITE);
  display.display();
  delay(1000);

}
/////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

First, we include the library needed for the OLED.

```
#include <Adafruit_SSD1306.h>
```

Then we make some standard variables for the OLED to define screen width, height, and the I2C address.

```
byte screenWidth = 128;
byte screenHeight = 64;
byte screenAddress = 0x3C;
```

After that we use the constructor to make our <u>display</u> object.

```
Adafruit_SSD1306 display(screenWidth, screenHeight, &Wire);
```

Now we can set the OLED up in the **setup()** function.

```
void setup() {
  display.begin(SSD1306_SWITCHCAPVCC, screenAddress);
  display.setTextColor(SSD1306_WHITE);
}
```

Next, we can use specific functions to draw different shapes on the OLED! Don't forget that (0,0) for the OLED is actually in the top left of the screen.

The first shape we will explore is actually just a point. We can draw points on the OLED with the function **drawPixel**. The first parameter is the X position, the second parameter is the Y position, and the last parameter is the color. All of these functions last parameter will be "WHITE" since this OLED can only display white.

```
void loop() {
  display.clearDisplay();
  display.drawPixel(63, 32, WHITE);
  display.display();
  delay(1000);
```

The next shape is a line. We can use the **drawLine** function. These parameters are the starting x position, starting y position, ending x position, ending y position, and the color.

```
  display.clearDisplay();
  display.drawLine(0, 0, 63, 63, WHITE);
  display.display();
  delay(1000);
```

The next shape is a circle. We can use the **drawCircle** function. These parameters are the x position, y position, circle radius, and color.

```
  display.clearDisplay();
  display.drawCircle(63, 31, 12, WHITE);
  display.display();
  delay(1000);
```

The next shape is also a circle. We can use the **fillCircle** function to draw a circle that is filled in. These parameters are the x position, y position, circle radius, and color.

```
  display.clearDisplay();
  display.fillCircle(63, 32, 12, WHITE);
  display.display();
  delay(1000);
```

The next shape is a triangle. We can use the **drawTriangle** function. This function takes a set of three points and draws lines connecting them. The parameters are x0, y0, x1, y1, x2, y2, and color.

```
  display.clearDisplay();
  display.drawTriangle(63, 0, 96, 63, 32, 63, WHITE);
  display.display();
  delay(1000);
```

The next shape is also a triangle. We can use the **fillTriangle** function. This function takes a set of three points, draws lines connecting them and fills in that area. The parameters are x0, y0, x1, y1, x2, y2, and color.

```
  display.clearDisplay();
  display.fillTriangle(63, 0, 96, 63, 32, 63, WHITE);
  display.display();
  delay(1000);
```

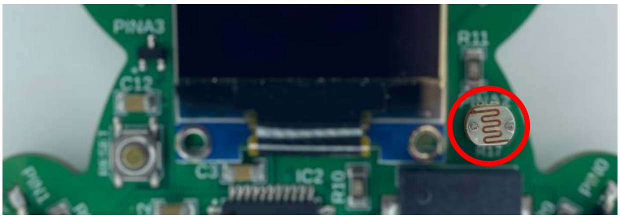
The next shape is a rectangle. We can use the **drawRect** function to draw a rectangle. This function takes the top left starting x position, top left starting y position, width of the rectangle, height of the rectangle and color.

```
  display.clearDisplay();
  display.drawRect(10, 10, 107, 43, WHITE);
  display.display();
  delay(1000);
```

The next shape is also a rectangle. We can use the **fillRect** function to draw and fill in a rectangle. This function takes the top left starting x position, top left starting y position, width of the rectangle, height of the rectangle and color.

```
  display.clearDisplay();
  display.fillRect(10, 10, 107, 43, WHITE);
  display.display();
  delay(1000);

}
//////////////////////////////////////////////////
```

Lesson 8: Using the Light Sensor



A Light Dependent Resistor (LDR), also known as a photoresistor, is a type of resistor whose resistance varies with the amount of light falling on it. In darkness, an LDR exhibits high resistance, and makes it harder for electricity to pass through it. Conversely, when exposed to light, the resistance drops dramatically, making it easier for electricity to flow through it. These are often found in automatic night lights, burglar alarms, streetlights, light intensity meter, and more.

Project 8.00 Light Sensor

In this project you'll learn what an LDR is and how to **analogRead** it to determine light intensity.

Project Code:

```
/////////////////////////////////////////////////
//8.00 - Reading Light Intensity

byte lightSensorPin = A2;

void setup() {
  pinMode(lightSensorPin, INPUT);
  Serial.begin(9600);
}

void loop() {
  Serial.print("The light level is at: "); Serial.println(analogRead(lightSensorPin));
  delay(1000);
}

/////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

Reading the LDR is exactly the same as reading the potentiometer.

The pin that is connected to the LDR is an analog pin.

```
byte lightSensorPin = A2;
```

The LDR is going to have an analog voltage, which is read using **analogRead**. It is then printed out onto the **Serial** monitor. Make sure and open the **Serial** port to see the data being printed.

```
  Serial.print("The light level is at: "); Serial.println(analogRead(lightSensorPin));
```

There is then a **delay** to keep the **Serial** port from being flooded with messages.

```
  delay(1000);
```

The LDR will read more voltage the darker it is. We can use this information to control outputs based on the light intensity.

Project 8.01 Max and Min Brightness

In this project, we'll find out the real-world minimum and maximum readings from our Light Dependent Resistor (LDR) in the room you're in. Unlike the potentiometer, in practice the LDR won't give us the full range from 0 to 1023 that we can read with the **analogRead** function.

Why is that? Well, our LDR is set up in a circuit in series another resistor, which has a resistance of 10,000 ohms. The readings we get depend on the total resistance of this setup.

For us to get a reading of 0, the LDR would have to have no resistance at all - that's 0 ohms, which isn't going to happen because all objects have some resistance.

Similarly, for us to get the maximum reading of 1023, the LDR would need to have an infinite amount of resistance. That's not possible either.

So, the real-world readings we get will be somewhere in between, and that's what we're going to find out!

Project Code:

```
//////////////////////////////////////////////
// 8.01 Max and Min Brightness

byte lightSensorPin = A2;
int maxValue = 0;
int minValue = 0;

void setup() {
  pinMode(lightSensorPin, INPUT);

  Serial.begin(9600);

  maxValue = analogRead(lightSensorPin);
  minValue = analogRead(lightSensorPin);
}

void loop() {
  int currentBrightness = analogRead(lightSensorPin);

  if (currentBrightness > maxValue) {
    maxValue = currentBrightness;
    Serial.print("The new max brightness is: "); Serial.println(maxValue);
  }
  if (currentBrightness < minValue) {
    minValue = currentBrightness;
    Serial.print("The new min brightness is: "); Serial.println(minValue);
  }
}
/////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

We will need a variable to hold the min and max voltage values that come across the LDR.

```
int maxValue = 0;
int minValue = 0;
```

In the **setup()** function, we have to load these variables we created with **analogRead** values. The reason these variables need loaded with real **analogRead** values is because we do not want to load them initially with values more or less than their range will support.

What does that mean? For example, if the lowest in the range of 0 – 1023 that the LDR would read was 20, then if we loaded the minValue with 0 and checked to see if any value went below it, it would never happen.

If we instead load it with an **analogRead** then that is part of the range by default, since it is a reading of the voltage across the LDR.

```
maxValue = analogRead(lightSensorPin);
minValue = analogRead(lightSensorPin);
```

In the **loop()** function, we first analogRead the current voltage value of the LDR

```
int currentBrightness = analogRead(lightSensorPin);
```

After that, we then look to see if the value of the currentBrightness variable is greater than the value of the maxValue variable. If so, we have a new max value and the maxValue variable needs to be reassigned with the value of currentBrightness. Then we print that value for our records.

```
  if (currentBrightness > maxValue) {
    maxValue = currentBrightness;
    Serial.print("The new max brightness is: "); Serial.println(maxValue);
```

```
 }
```

We then do the same thing but with the minValue variable. If currentBrightness variable is less than the minValue variable, we reassign it with its value and print it.

```
 if (currentBrightness < minValue) {
   minValue = currentBrightness;
   Serial.print("The new min brightness is: "); Serial.println(minValue);
 }
```

Make sure and write these values down for the next project. These are the max and min brightness values for the room you're in. Think of this project as calibrating your MC Trainer to get the most out of it.

Project 8.02 Mapping Light

In this project we will use the Min and Max brightness values from the last sketch to vary the brightness of an LED based on the light on the LDR. The darker the room, the brighter LED1 will be.

Project Code:

```
////////////////////////////////////////
// 8.02 Mapping Light

byte lightSensorPin = A2;
int maxValue = 909;
byte minValue = 23;

byte LED1 = 13;

void setup() {
  pinMode(lightSensorPin, INPUT);
  pinMode(LED1, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  int currentValue = analogRead(lightSensorPin);

  int LED1Brightness = map(currentValue, minValue, maxValue,     0,      255);

  LED1Brightness = constrain(LED1Brightness, 0, 255);

  analogWrite(LED1, LED1Brightness);
}

////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

In my case, my maxValue is 909 and my minValue is 23. Yours will probably be slightly different. You'll need to update the variables below for your specific values.

```
int maxValue = 909;
byte minValue = 23;
```

In the **loop**() function, we first have to **analogRead** the voltage on the LDR. The **analogRead** function reads from 0 – 1023 (10-bit) but the **analogWrite** function write values of 0 – 255 (8-bit). The **map** function will take care of this issue. Remember, the **map** function takes five parameters. It takes the value to map, the min value of that range, the max value of that range, the min value of the range to be mapped to, and the max value of the range to be mapped to.

```
int LED1Brightness = map(currentValue, minValue, maxValue,     0,       255);
```

Sometimes the LDR will register values higher or lower than the range we defined in our code. For this reason, we want to “Constrain” our readings. This just keeps our readings from going above or below the range we’ve defined.

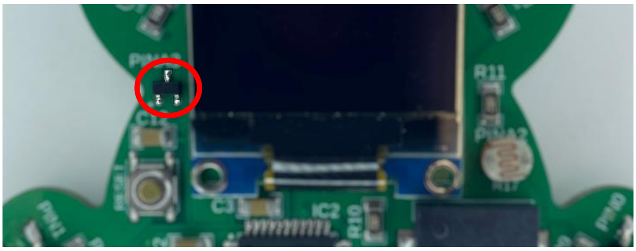
To keep this from happening, what we need to do is use the **constrain** function. The **constrain** function allows us to keep a variable in between certain numbers. For example, if we were to get a value of 300 and our constrain range is 0 – 255, the function would return 255. The same is true for the lower range, but it returns the low constrained value. If the value is instead within the range, it just returns that value.

```
  LED1Brightness = constrain(LED1Brightness, 0, 255);
```

Lastly, the LED1Brightness analog value is written to LED1.

```
  analogWrite(LED1, LED1Brightness);
```

Lesson 9: Using the Temperature Sensor



The temperature sensor on the MC Trainer is a linear active thermistor. These temperature sensors give out a voltage signal that changes directly with temperature. This means the relationship between the voltage output and the temperature is straight and direct, making the data easier to read and use.

Project 9.00 Using the Temp Sensor

In this project you'll learn how to read the voltage produced by the temperature sensor. The temperature sensor on the MC Trainer is tiny! It is the tiny black rectangle on the left of the OLED, above the reset button.

Project Code:

```
/////////////////////////////////////////////////////
// 9.00 - Using the Temperature Sensor

byte tempSensorPin = A3;

void setup() {
  pinMode(tempSensorPin, INPUT);
  Serial.begin(9600);

}

void loop() {
  int currentTemp = analogRead(tempSensorPin);

  Serial.print("The current temp value is: "); Serial.println(currentTemp);
  delay(1000);

}
/////////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

The temperature sensor works a lot like the potentiometer and LDR. The difference is that this temperature sensor actually produces a voltage based on the temperature. The potentiometer and LDR were dropping a certain amount of voltage based on some condition.

The temperature sensor again produces an analog voltage based on the temperature. This means that if we want to read that analog value it needs to be on an analog pin. The temperature sensor is connected to the analog pin A3 on the MC Trainer.

```
byte tempSensorPin = A3;
```

We are reading the voltage, so the pin is an INPUT.

```
  pinMode(tempSensorPin, INPUT);
```

In the **loop()** function, we analog read the voltage produced by the temperature sensor.

```
  int currentTemp = analogRead(tempSensorPin);
```

After that, we print it onto the **Serial** port. Make sure to open the **Serial** port to see the data being printed.

```
  Serial.print("The current temp value is: "); Serial.println(currentTemp);
```

So that we don't flood the screen we add in a one second **delay**.

```
  delay(1000);
```

Try touching the temperature sensor and seeing how the values change.

Project 9.01 Getting an Actual Temperature Reading

In this project you'll learn how to convert the analog value read from the sensor into an actual temperature. This can get a bit complicated but it's how many sensors work in the real world.

Project Code:

```
////////////////////////////////////////////////////
// 9.01 - Getting an Actual Temperature Reading

byte tempSensorPin = A3;

void setup() {
  pinMode(tempSensorPin, INPUT);
  Serial.begin(9600);

}

void loop() {

  float currentTemperature = ReadTemperature();

  Serial.print("The temp in C is: "); Serial.println(currentTemperature);
  Serial.print("The temp in F is: "); Serial.println(CtoF(currentTemperature));
  delay(1000);

}

/*
   This function reads a temperature in degrees C from the MCP9700AT-E/TT and
returns it.
*/
float ReadTemperature() {
  int currentTempReading = analogRead(tempSensorPin);

  float currentTempVoltage = currentTempReading * (5.0 / 1024.0);

  float temperature = (currentTempVoltage - .5) /  .01;

  return(temperature);
}

/*
   This function converts a temperature passed in degrees C to degrees F and
returns it.
*/
float CtoF(float temp) {
  return ((temp * 1.8) + 32);
}
////////////////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

Before we look at the code let's figure out how we are going to go about getting an actual temperature reading. The first thing that we need to find is the equation for converting the analog value read to a temperature. This can be found in the datasheet for the part on page 10.

https://www.mouser.com/datasheet/2/268/MCP9700_Family_Data_Sheet_DS20001942-3132871.pdf

Datasheets can be very intimidating, even for seasoned engineers. Take a look at the datasheet for the Atmega32u4 (The chip you're programming):

https://ww1.microchip.com/downloads/en/devicedoc/atmel-7766-8-bit-avr-atmega16u4-32u4_datasheet.pdf

It's 438 pages!

The equation on page 10 states:

Vout = Tc * Ta + V0degreesC

Let's break that down:

Symbol	What it means
Vout	Voltage produced by the temperature sensor
Tc	Temperature Coefficient
Ta	Ambient Temperature
V0degreesC	Voltage produced at 0 degrees C

So, what do we know? Only the voltage read right now. Let's go back to the datasheet.

On page 4 we can find the temperature coefficient (10mV per degree C), on page 3 we can find the V0degreesC (500mV) and what we are trying to figure out is the ambient temperature.

Now we know the voltage being read (with **analogRead**), the temperature coefficient, and the V0degrees, we can re-arrange the equation for ambient temperature:

1.)	Vout = Tc * Ta + V0degreesC	
2.)	Vout – V0degreesC = Tc * Ta	<- Subtracted V0degrees
3.)	(Vout – V0degreesC) / Tc = Ta	<- Divided by Tc
4.)	Ta = (Vout – V0degreesC) / Tc	<- Just re-written from #3

When we read the analog voltage value from the temperature sensor it will be a value between 0 – 1023. Unfortunately, that is in the wrong form for us. What we need is voltage, not a 10-bit value. In order to convert from a 10-bit number to the voltage it represents we can use the equation:

10bitValue * (5 / 1024)

For example, we would expect a value of about 2.5 with an analog value of 512:

512 * (5 / 1024) = ~2.5v

We can use this equation to convert from 10-bit to voltage.

Now that we have all of the tools we need let's look at how to use them.

In the **loop()** function, we have created a variable of type **float** and assigned it the value that the function **ReadTemperature()** will return. All this does is run the function and assign the value it returns to the currentTemperature variable.

```
  float currentTemperature = ReadTemperature();
```

In the **ReadTemperature()** function, we first take an **analogRead** of the temperature sensor pin.

```
  int currentTempReading = analogRead(tempSensorPin);
```

After that we then convert that analog value read into a voltage and assign it to a variable of type **float**. You may notice the ".0" on the end of the numbers. This specifies these numbers as float type numbers. The number "5" is an **int** and the number "5.0" is a **float**. So, if we're trying to do floating point math, we have to use the ".0" with our numbers. If we did "5 / 1023" the result would be 0!

```
  float currentTempVoltage = currentTempReading * (5.0 / 1023.0);
```

Next, we use the equation we re-arranged earlier to convert to an actual temperature. That value is assigned to a **float** type variable.

```
  float temperature = (currentTempVoltage - .5) /  .01;
```

Then that value is returned.

```
  return(temperature);
```

Going back to the **loop()** function, we then **Serial** print the temperature value out in Celsius and Fahrenheit.

```
  Serial.print("The temp in C is: "); Serial.println(currentTemperature);
  Serial.print("The temp in F is: "); Serial.println(CtoF(currentTemperature));
```

The **CtoF** function is there to convert from Celsius to Fahrenheit.

```
/*
   This function converts a temperature passed in degrees C to degrees F and returns
it.
*/
float CtoF(float temp) {
  return ((temp * 1.8) + 32);
}
```

A one second **delay** is used after the **Serial** prints to prevent flooding of the **Serial** port.

```
  delay(1000);
```

Who knew reading temperature could be so complicated? Luckily, these functions we made can just be copied and pasted into other programs. This is why people create libraries!

Project 9.02 Doing Something Based on Temperature

In this project we will use the temperature data to change the color of the Neopixels based on the temperature. The warmer the temperature to more red the Neopixels will appear. The colder the temperature sensor the more blue the Neopixels will appear.

Project Code:

```
////////////////////////////////////////////////////
// 9.02 - Doing Something Based on Temperature

#include <Adafruit_NeoPixel.h>

byte dataPin = 10;
byte numberOfPixels = 2;
byte brightness = 10;

byte redValue = 0;
byte greenValue = 0;
byte blueValue = 0;

byte tempSensorPin = A3;

Adafruit_NeoPixel pixels(numberOfPixels, dataPin, NEO_GRB + NEO_KHZ800);
void setup() {
  pinMode(tempSensorPin, INPUT);

  pixels.begin();
  pixels.setBrightness(brightness);

  Serial.begin(9600);
}

void loop() {

  float currentTemperature = ReadTemperature();

  Serial.print("The temp in C is: "); Serial.println(currentTemperature);
  Serial.print("The temp in F is: "); Serial.println(CtoF(currentTemperature));

  UpdatePixelColorBasedOnTemp(CtoF(currentTemperature));
  delay(1000);

}

/*
   This function takes a temperature in degrees F and updates the neopixels to
reflect the temperature.
   The hotter the temperature, the more red the neopixels.
   The colder the temperature, the more blue the neopixels.
```

```
*/
void UpdatePixelColorBasedOnTemp(float temperature) {

  int blueValue = map(temperature, 70, 80, 255, 0);
  blueValue = constrain(blueValue, 0, 255);

  int redValue = map(temperature, 70, 80, 0, 255);
  redValue = constrain(redValue, 0, 255);

  Serial.print("Blue: "); Serial.println(blueValue);
  Serial.print("Red: "); Serial.println(redValue);

  pixels.setPixelColor(0, pixels.Color(redValue, 0, blueValue));
  pixels.setPixelColor(1, pixels.Color(redValue, 0, blueValue));
  pixels.show();
}

/*
   This function reads a temperature in degrees C from the MCP9700AT-E/TT and
returns it.
*/
float ReadTemperature() {
  int currentTempReading = analogRead(tempSensorPin);

  float currentTempVoltage = currentTempReading * (5.0 / 1024.0);
  float temperature = (currentTempVoltage - .5) /  .01;

  return (temperature);
}

/*
   This function converts a temperature passed in degrees C to degrees F and
returns it.
*/
float CtoF(float temp) {
  return ((temp * 1.8) + 32);
}
//////////////////////////////////////////////////////////
```
*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

This project is fairly simple thanks to the functions we wrote in the last project. They are simply copied and pasted into this one.

The first thing that we need to do is include the Neopixel library.

```
#include <Adafruit_NeoPixel.h>
```

Next, we declare some standard variables and include the constructor for the Neopixel library.

```
byte dataPin = 10;
byte numberOfPixels = 2;
byte brightness = 10;

byte redValue = 0;
byte greenValue = 0;
byte blueValue = 0;

byte tempSensorPin = A3;

Adafruit_NeoPixel pixels(numberOfPixels, dataPin, NEO_GRB + NEO_KHZ800);
```

After that in the **setup()** function, we set the **pinMode** of the analog pin connected to the temperature sensor, initialize the Neopixels, and setup **Serial** communication.

```
void setup() {
  pinMode(tempSensorPin, INPUT);

  pixels.begin();
  pixels.setBrightness(brightness);

  Serial.begin(9600);
}
```

The **loop()** function starts just like the last project did. First, we get the current temperature.

```
void loop() {

  float currentTemperature = ReadTemperature();
```

Next, the temperature is printed out in Celsius and Fahrenheit.

```
  Serial.print("The temp in C is: "); Serial.println(currentTemperature);
  Serial.print("The temp in F is: "); Serial.println(CtoF(currentTemperature));
```

Let's take a look at the next function in the **loop(), "UpdatePixelColorBasedOnTemp"**. From the function description we know that this function takes a temperature in degrees Fahrenheit and updates the Neopixels to reflect that temperature. This is why the temperature is passed to the function with the **CtoF** function.

```
  UpdatePixelColorBasedOnTemp(CtoF(currentTemperature));
```

Inside the function we can see what is really happening. First the blue value is mapped to the temperature range of 70 – 80 degrees F.

The parameters in this implementation of the **map** function are a bit different than usual. You'll see that the higher end of the output range is mapped to the lower end of the temperature range, and vice versa. This means that when the temperature is 70 degrees the output will be 255 and when the temperature is 80 degrees the output will be 0. This is an easy way to invert the output range to go from high to low. We need this because as the temperature gets lower, we want a larger blue value.

```
  int blueValue = map(temperature, 70, 80, 255, 0);
```

Then the **constrain** function is used to ensure we stay in the proper output range.

```
  blueValue = constrain(blueValue, 0, 255);
```

Next, we will use the **map** function normally to map the temperature range to the red output range. The **constrain** function is used to keep the values in the proper range.

```
  int redValue = map(temperature, 70, 80, 0, 255);
  redValue = constrain(redValue, 0, 255);
```

Then these values are printed out onto the **Serial** port.

```
  Serial.print("Blue: "); Serial.println(blueValue);
  Serial.print("Red: "); Serial.println(redValue);
```

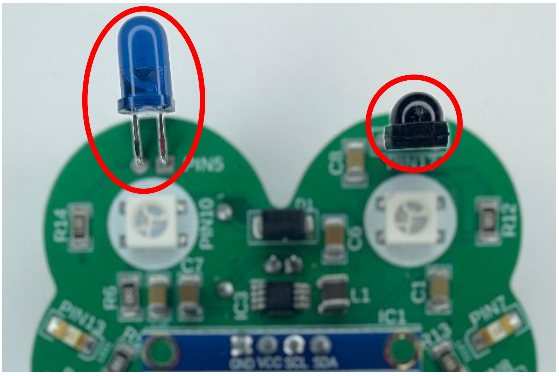
After that, the Neopixels are loaded and updated.

```
  pixels.setPixelColor(0, pixels.Color(redValue, 0, blueValue));
  pixels.setPixelColor(1, pixels.Color(redValue, 0, blueValue));
  pixels.show();
}
```

Going back to the **loop()** function, after the **UpdatePixelColorBasedOnTemp** function, a small **delay** is implemented to keep the **Serial** port from flooding and the **loop()** is ended.

```
  delay(1000);

}
```

Lesson 10: Using the IR Receiver and Emitter



An infrared emitter and detector are paired electronic devices that enable wireless communication and sensing applications using infrared light. These components are found in a wide range of devices, including remote controls, security systems, and obstacle detection systems for robotics.

Project 10.00 Decoding IR

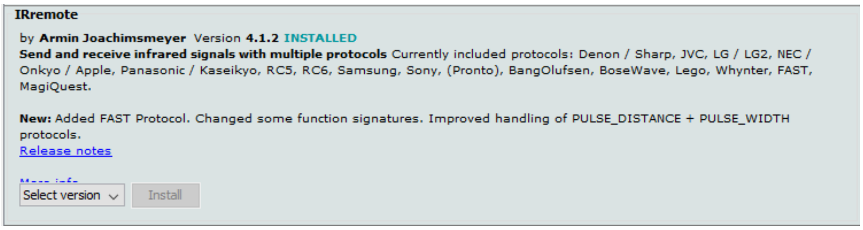
This Arduino sketch is designed to receive and decode infrared signals. When you point a remote at the MC Trainer, the decoded results are printed to the **Serial** Monitor.

Project Code:

```
/////////////////////////////////////////
// 10.00 Infrared Receiver

#include <IRremote.h>

byte IRRecv = 17;

void setup() {
  Serial.begin(9600);
  IrReceiver.begin(IRRecv);
}

void loop() {
  if (IrReceiver.decode()) {
    IrReceiver.printIRResultShort(&Serial);
    IrReceiver.resume();
  }
}
////////////////////////////////////////
```

*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

Before we do anything, we need to install the IRremote Library. You can do this by going to Tools -> Manage Libraries, and type in "IRremote". It looks like this:



IRremote

by **Armin Joachimsmeyer** Version **4.1.2** INSTALLED
Send and receive infrared signals with multiple protocols Currently included protocols: Denon / Sharp, JVC, LG / LG2, NEC / Onkyo / Apple, Panasonic / Kaseikyo, RC5, RC6, Samsung, Sony, (Pronto), BangOlufsen, BoseWave, Lego, Whynter, FAST, MagiQuest.

New: Added FAST Protocol. Changed some function signatures. Improved handling of PULSE_DISTANCE + PULSE_WIDTH protocols.
Release notes

Click install. If it asks you to install dependencies go ahead and do so.

For reference, you can include it into a project by going to Sketch -> Include Library, and click "IRremote". This does not need done in this sketch as it is already included.

The library reference where you can find all of the functions avaliable can be found here:

https://github.com/Arduino-IRremote/Arduino-IRremote

In the **setup()** function we initialize the IR receiver on digital pin 17 using the **begin()** method from the IRremote library.

```
IrReceiver.begin(IRRecv);
```

The **loop()** function continuously checks for incoming IR signals. It uses the **decode()** method to check if a signal has been received. If a signal has been received the **decode()** method then returns a one and the **if-statement** code executes.

```
if (IrReceiver.decode()) {
```

If a signal is received, it then prints a short summary of the decoded data to the **Serial** Monitor using the **printIRResultShort** function. The data printed includes the protocol used, the address, command,

and raw data. Much like the “&Wire” don’t worry too much about what the “&Serial” that we pass is really doing. Just know we need to pass that to print out the data.

```
    IrReceiver.printIRResultShort(&Serial);
```

It then resumes the receiver to keep listening for new signals using the **resume()** function.

```
    IrReceiver.resume();
  }
}
////////////////////////////////////////////
```

Make sure and open the **Serial** monitor to see what is being printed.

Now that you can see what the signal that your remote is sending, you can use the IR emitter to copy that signal.

The data that I got from my remote was:

“Protocol=NEC Address=0x0 Command=0x9 Raw-Data=0xF609FF00 32 bits LSB first”

Let’s break that data down:

Section	Data
Protocol=NEC	This means the infrared signal uses the NEC protocol, a common infrared protocol used in consumer electronics.
Address=0x0	The address is used to specify the target device that should respond to this signal. In this case, the address is 0x0, which usually indicates a generic device.
Command=0x9	The command portion of the signal tells the device what to do. In this case, the command is 0x9.
Raw-Data=0xF609FF00	This represents the raw data of the signal received by the IR receiver. This is typically a sequence of pulse and space lengths that make up the actual signal. The raw data is useful for debugging or for cases where the protocol, address, and command don't adequately represent the signal.

32 bits LSB first	This indicates that the data is 32 bits long and that it's being read from the least significant bit (LSB) first. This is important for correctly interpreting the raw data.

We can use this data to emulate the remote with the IR emitter LED.

Project 10.01 Sending IR

This project aims to replicate the functionality of an IR remote control. We'll be using a push-button switch to trigger the sending of specific IR signals, which can be read by an IR receiver. This can be used to do things like turn on a TV!

Project Code:

```
//////////////////////////////////////
// 10.01 Sending Data

#include <IRremote.h>

byte sCommand = 0x9;
byte ADDR = 0x0;

byte SW1 = 1;
bool pressed = 0;

byte IRPin = 5;

void setup() {
  Serial.begin(9600);
  IrSender.begin(IRPin);
  pinMode(SW1, INPUT);
}

void loop() {
  if (digitalRead(SW1) == pressed) {
    Serial.println();
    Serial.print("Send now: address=0x");
    Serial.print(ADDR, HEX);
    Serial.print(", command=0x");
    Serial.println(sCommand, HEX);

    IrSender.sendNEC(ADDR, sCommand, 0);

    delay(250);
  }
}
////////////////////////////////////////
```

*If you’re copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

In the **setup()** function, the IR emitter is started on pin 5 using **IrSender.begin(**IRPin**)**.

```
IrSender.begin(IRPin);
```

In the **loop()** function, SW1 is monitored to see if it has been pressed.

```
if (digitalRead(SW1) == pressed) {
```

If the button has been pressed, information about what is being sent is printed using the **Serial** methods. You’ll notice that some of the **Serial.print** methods have two parameters. You can use keywords such as **DEC, HEX, BIN, OCT** to print numbers in a specific number base. Data used with the IR protocol is generally in hexadecimal, so we print it in that fashion with the **HEX** keyword.

```
  Serial.println();
  Serial.print("Send now: address=0x");
  Serial.print(ADDR, HEX);
  Serial.print(", command=0x");
  Serial.println(sCommand, HEX);
```

Next, the data is actually sent with the **IrSender.sendNEC** method. There are three parameters to this method. The first parameter is the address that the data should be sent to, the second parameter is the data to be sent, and the last parameter is the number of times the data should be repeated. Sometimes it is useful to repeat the signal a few times to ensure it makes it to the intended device.

```
  IrSender.sendNEC(ADDR, sCommand, 0);
```

What if the data is not the NEC protocol? This library supports a wide range of protocols. They are listed here:

Protocol	Method
NEC	IrSender.sendNEC()
Sony	IrSender.sendSony()
RC5	IrSender.sendRC5()
RC6	IrSender.sendRC6()

Dish	IrSender.sendDISH()
JVC	IrSender.sendJVC()
Samsung	IrSender.sendSAMSUNG()
LG	IrSender.sendLG()
Whynter	IrSender.sendWhynter()
COOLIX	IrSender.sendCOOLIX()
Denon	IrSender.sendDenon()
Sharp	IrSender.sendSharpRaw()
Panasonic	IrSender.sendPanasonic()
Sanyo	IrSender.sendSanyo()
Mitsubishi	IrSender.sendMitsubishi()
Apple	IrSender.sendApple()
Pronto	IrSender.sendPronto()
LEGO Power Functions	IrSender.sendLegoPowerFunctions()
Bose Wave	IrSender.sendBoseWave()
Metz	IrSender.sendMetz()
MagiQuest	IrSender.sendMagiQuest()
RCMM	IrSender.sendRCMM()

Some of these methods may take different parameters so consult the library reference, or simply Google it before using.

The loop is then ended with a **delay** if the button was pressed as a crude debounce.

```
    delay(250);
  }
}
////////////////////////////////////
```

Lesson 11: Emulate Mouse

Project 11.00 Using the Board as a Mouse

The MC trainer can also act like a mouse on a computer! This functionality is unique with microcontrollers. In this project you'll learn how to use it as such!

Project Code:

```
////////////////////////////////////////////////////
// 11.00 - Using The Board As A Mouse

#include "Mouse.h"

byte SW1 = 1;
byte pressed = LOW;

void setup() {
  pinMode(SW1, INPUT);

  Mouse.begin();
}

void loop() {
  if (digitalRead(SW1) == pressed) {

    for (byte x = 0; x < 100; x++) {
      Mouse.move(1, 0, 0);
    }

    for (byte x = 0; x < 100; x++) {
     Mouse.move(0, 1, 0);
    }

    for (byte x = 0; x < 100; x++) {
      Mouse.move(-1, 0, 0);
    }

    for (byte x = 0; x < 100; x++) {
      Mouse.move(0, -1, 0);
    }
  }

  //You can also:
  /*
    Click the mouse:
    Mouse.press(MOUSE_RIGHT);
    Mouse.press(MOUSE_MIDDLE);
    Mouse.press(MOUSE_LEFT);
    Release the mouse:
    Mouse.release(MOUSE_RIGHT);
    Mouse.release(MOUSE_MIDDLE);
    Mouse.release(MOUSE_LEFT);

  */
}
////////////////////////////////////////////////////
```
*If you're copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

Before you write a program that includes mouse functionality it is important to be able to control it. In this project we don't activate the mouse functionality unless a button is pressed. When you plug the MC Trainer in, you don't want your mouse going crazy!

The first thing that we need to do is include the proper library. There is no need to install this library as it comes standard with the Arduino IDE. All you'd have to do is type in " #include "Mouse.h".

```
#include "Mouse.h"
```

Next, we need to include some variables to be used with the button.

```
byte SW1 = 1;
byte pressed = LOW;
```

In the **setup()** function, we need to set the **pinMode** of the button and initialize the mouse functionality. Initializing the MC Trainer as mouse can be done with the **Mouse.begin()** method.

```
void setup() {
  pinMode(SW1, INPUT);

  Mouse.begin();
}
```

In the **loop()** function, we check to see if the button has been pressed before using the mouse functionality.

```
void loop() {
  if (digitalRead(SW1) == pressed) {
```

Once SW1 has been pressed, we can use the **Mouse.move** method to move the mouse! The first parameter is how many pixels to move in the x direction, the second is how many pixels to move in the y direction and the last is how many lines to scroll. The mouse will move instantly so it has been put into a series of **for-loops** and run a certain number of times to make incremental changes instead (So we can see it change).

```
    for (byte x = 0; x < 100; x++) {
      //          x  y  scroll
      Mouse.move(1, 0, 0);
    }

    for (byte x = 0; x < 100; x++) {
      //          x  y  scroll
      Mouse.move(0, 1, 0);
    }
```

To move in the opposite direction all you have to do is pass it a negative value.

```
    for (byte x = 0; x < 100; x++) {
      //          x  y  scroll
      Mouse.move(-1, 0, 0);
    }

    for (byte x = 0; x < 100; x++) {
      Mouse.move(0, -1, 0);
    }
  }
```

Here are some additional functions you can use with the Mouse.h library.

```
  //You can also:
  /*
    Click the mouse:
    Mouse.press(MOUSE_RIGHT);
    Mouse.press(MOUSE_MIDDLE);
    Mouse.press(MOUSE_LEFT);

    Release the mouse:
    Mouse.release(MOUSE_RIGHT);
    Mouse.release(MOUSE_MIDDLE);
    Mouse.release(MOUSE_LEFT);

  */
}
//////////////////////////////////////////////////////
```

Lesson 12: Emulate Keyboard

Project 12.00 Using the Board as a Keyboard

The MC trainer can also act like a keyboard!

Project Code:

```
////////////////////////////////////////////////////////
// 12.00 - Using the Board as a Keyboard

#include "Keyboard.h"

byte SW1 = 1;
bool pressed = LOW;

int pressCounter = 0;

void setup() {
  pinMode(SW1, INPUT);

  Keyboard.begin();
}

void loop() {
  while (digitalRead(SW1) != pressed) {
    ;
    }

  pressCounter++;

  Keyboard.print("You have pressed the button: ");
  Keyboard.print(pressCounter);
  Keyboard.println(" times!");

  delay(500);
}
////////////////////////////////////////////////////////
```

*If you’re copying and pasting the code, or typing from scratch, delete everything out of a new Arduino sketch and paste / type in the above text.

Just like the previous project, we need to put the keyboard on some kind of control. We don’t want thousands of keypresses being sent to the computer at a time!

Also like the other project, to use the keyboard functionality we need to include a library that comes with the Arduino IDE by default. We can do that by simply typing in “ #include "Keyboard.h".

```
#include "Keyboard.h"
```

We then use some standard variables to represent the button.

```
byte SW1 = 1;
bool pressed = LOW;
```

In this program we are also going to use a variable to keep track of the number of times the button has been pressed.

```
int pressCounter = 0;
```

Then in the **setup()** function, all we need to do is set the correct **pinMode** for the button pin and initialize the keyboard with the **Keyboard.begin()** function.

```
void setup() {
  pinMode(SW1, INPUT);

  Keyboard.begin();
}
```

In the **loop()** function, the first thing we do is wait until the button is pressed.

```
void loop() {
  while (digitalRead(SW1) != pressed) {
    ;
   }
```

When the button is pressed, the program moves on from the **while-loop.** Since the button has been pressed, we increment the pressCounter variable.

```
  pressCounter++;
```

After that we can use the MC Trainer to type that information out onto the screen. The **print** and **println** functions here work exactly the same as the **Serial** versions.

```
  Keyboard.print("You have pressed the button: ");
  Keyboard.print(pressCounter);
  Keyboard.println(" times!");
```

A crude debounce is then implemented to keep the button from being read too many times.

```
  delay(500);
}
//////////////////////////////////////////////////
```

To see the text being printed out open a text editor, click on the screen, and press the button.

```
You have pressed the button: 1 times!
You have pressed the button: 2 times!
You have pressed the button: 3 times!
You have pressed the button: 4 times!
You have pressed the button: 5 times!
You have pressed the button: 6 times!
You have pressed the button: 7 times!
You have pressed the button: 8 times!
You have pressed the button: 9 times!
You have pressed the button: 10 times!
```

Electronics

You don’t need to know much about electronics to do the projects in this book. However, it’s a good idea to understand a few concepts that will take you a long way in electronics. They are:

1.) **Electrical potential (voltage).** This is the difference in electrical charge between two points. You can think of electrical potential as the “pressure” that drives electricity through a circuit. When high potential exists between two points, the results can be dramatic; the difference in electrical potential between the sky and ground causes lightning to strike. Electrical potential is measured in volts and is often represented by the letter V.

2.) **Electrical resistance**. This measures how difficult it is for electricity to travel between two points. Take a battery, for instance. When the battery sits on your table by itself, no electricity will flow between the terminals because there isn’t a suitable path between them. You could argue that the terminals are connected by air, but the electrical resistance of air is very high (it takes a *lot* of volts to make lightning travel through air). If you attach a wire to the terminals, electricity will flow. Don’t do this, by the way, because the resistance of a piece of wire is very low, and so much electricity will flow that you will damage the battery. All paths that electricity takes need a fair amount of resistance to avoid damaging the circuit. Resistance is measured in ohms. It is represented by the letter R or the Greek symbol Ω (omega).

3.) **Electrical Current.** Current is the amount of electricity that flows from a point of high potential to a point of low potential. Current is what does the work in electronics. It is measured in amperes, or amps, and is represented by the letter I.

The real beauty of electronics is that these three concepts are related to one another in a simple way called Ohm's law. Ohm's law boils down to an equation that can be written in a few different ways. The most common way is:

I = V/R

This means that the current that will flow through a path equals the difference in electrical potential at the two ends of the path, divided by the path's resistance. For example, if you had a 6-volt battery and you created a path with a resistance of 2 ohms:

I = 6/2 = 3 amps of current

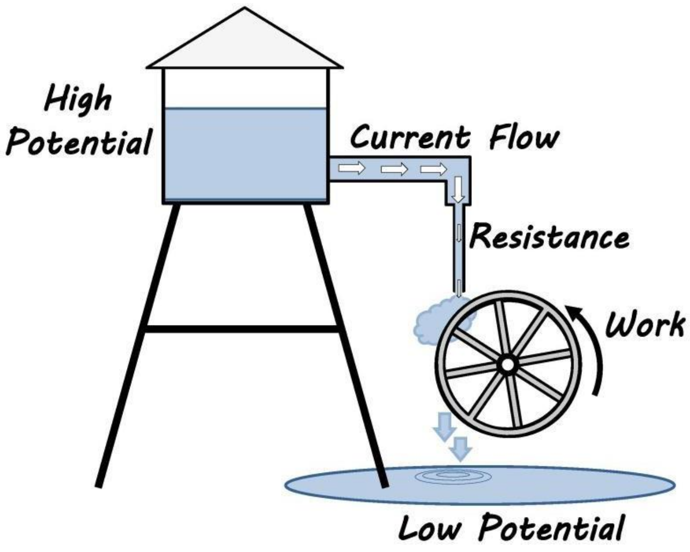
This is a lot of current for an Arduino project! We often create circuits where something like 10 milliamps (mA) of current will flow (1 mA = 1/1000th of an amp).

You can rearrange the parts of Ohm's law to calculate voltage, resistance, or current from the other two parts. One common way to rearrange it is:

V = I x R

If we know how much current flows through a path and its resistance, we can calculate the change in electrical potential between any two points on the path. This is called a "voltage drop." We use this part of Ohm's law to measure things like the position of a dial in a potentiometer.

Electricity flows like Water



It’s easier to wrap our heads around these concepts by thinking of electricity like the flow of water. We can create high potential by filling a water tower. This is analogous to a difference in voltage. And just like charging a battery, it takes energy to fill a water tower (the law of conservation of energy is alive and well!). Water will flow if we create a path between a point of high and low potential, just like an electrical current. The path in our analogy could be a pipe that allows water to flow from the water tower to the ground. In an electrical circuit, the path consists of copper wire and electronic components that electricity will flow through.

The flow rate will depend on the difference in potential and the path's resistance (remember I = V/R?). If we use a big pipe, water will flow quickly; a narrow section of pipe will add resistance and slow the flow rate through the entire circuit. We often use special devices called resistors to slow the rate of electrical current through our circuits.

Like electricity, we can capture the energy of the flowing water to do work. We capture electrical energy to turn on lights, make sounds, turn motors, heat and cool objects, and even do the math!

Circuits

A circuit is a set of electronic components and paths between them that do one or more jobs. All circuits need a voltage source, a path for current to flow, and usually some type of work to do. A very simple circuit could be a battery, a light bulb, and two wires that provide a path between the battery and the bulb. Circuits always contain a way for electricity to flow from the point of high potential to a point of low potential. This means we can trace a path from the positive side of the battery or other power source, through each component, to the negative side.

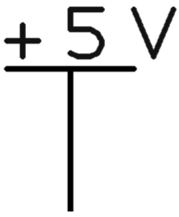
We often need a way to “make” or “break” the circuit. Returning to our simple circuit, if we just put these components together, the light bulb would light until the battery died. If we put a switch in the circuit, we could “break” the circuit (also called an “open” circuit) by interrupting the path on one of the two wires. We could then switch the light on to “make” the circuit (also called a “closed” circuit) when we needed it.

The circuits in most electronic devices are arranged on printed circuit boards (PCBs). PCBs are hard plastic boards with electronic components soldered onto them. Instead of round wires, PCBs use "traces" to connect the electronic components. These are thin, flat strips of copper on the surface of the boards. The traces are commonly covered with a green finish that gives PCBs their characteristic look. The MC Trainer is a PCB with all the circuits needed to complete the projects in this book.

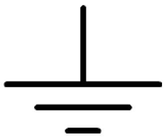
Electronic Components

If you look inside a computer or another electronic device, you'll see components with different shapes, sizes, and colors arranged on a circuit board. It looks like a miniature city arranged on a green field. A circuit board is like a city
– each component has a job and is linked by electronic "roads" (the traces).

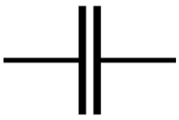
Let's take a look at some electronic components on the MC Trainer and the jobs that they do. Electronic circuits are mapped out in *schematics* that show how all components are connected. Each component has a unique symbol. Schematics are not arranged the same way as the layout of circuit boards. That would be confusing. Instead, schematics are arranged to make it easy to see how parts are connected.



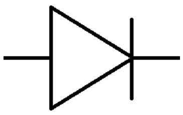
Positive supply voltage: supplies the circuit with high electrical potential. It can originate from the positive terminal of a battery or a source like a USB hub. It is sometimes also called VCC or VSS.



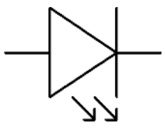
Ground: the negative supply voltage that supplies the circuit with low electrical potential. The ground symbol is often used in circuit schematics instead of showing the connection back to the negative terminal of the battery or power supply. It is sometimes also called GND, VDD, or VEE.



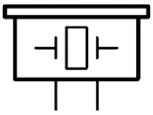
Capacitor: a device holding a small electrical charge, like a tiny battery. Capacitors are often used to avoid rapid changes in voltage as different devices use different amounts of current.



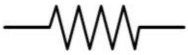
Diode: a device that acts as a one-way door. Current will flow when the anode voltage (left side of the symbol) is higher than the cathode voltage (right side). Current will not flow in the reverse direction.



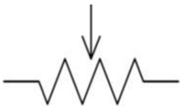
Light-Emitting Diode (LED): a one-way door that lights up when current flows through it. LEDs come in many different colors and sizes. Some LEDs emit infrared light that is invisible to
the human eye but can be sensed by an infrared transistor.



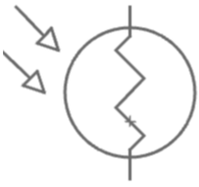
Piezo Element: brings sound to our projects. The piezo element is a special material that changes shape slightly when voltage is applied. We can make it emit different tones by rapidly turning the voltage on and off at different rates. The piezo element also generates a voltage when it is forced to change shape. In some of our projects, we will use this property to sense a finger tapping on it.



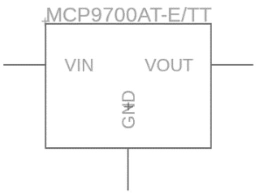
Resistor: reduces current in a circuit (remember I = V/R). Resistors are used to control how much current will flow through different parts of a circuit. We usually want only a few milliamps of current flowing through our microcontroller and the rest of the circuit.



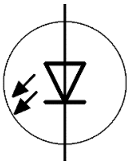
Potentiometer: a voltage divider. It’s a resistor with a “brush” connected to a third terminal. The voltage of the third terminal depends on the position of the brush. They are commonly used as position sensors or in dials. You can learn more about potentiometers and other voltage dividers by looking up “voltage dividers” on Wikipedia.



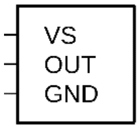
Light Dependent Resistor: LDR, also known as a photoresistor, is an electronic component that changes its resistance based on the amount of light it is exposed to. The resistance decreases as the intensity of light increases, and vice versa.



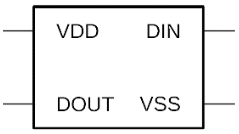
Temperature Sensor: The MCP9700AT-E/TT is a Microchip Technology analog temperature sensor device. It converts temperature from -40°C to +125°C to a linear analog voltage output. The voltage output of the MCP9700AT-E/TT is proportional to the measured temperature, which makes it straightforward to interpret the output signal.



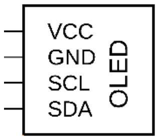
IR Transmitter: is a device that is used for communication. Brief pulses of infrared light do this communication. Infrared light is invisible to the human eye and is a non-intrusive way to send information.



IR Receiver: is a device that allows the detection of infrared light. These are used in televisions, DVD players, air conditioners, and more.

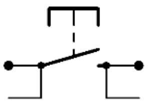


Neopixel: is a unique type of RGB LED that allows for fine control of 3 differently colored LEDs in one. Neopixels allow for easy connection of more LEDs at no more cost than microcontroller pins.



OLED: stands for organic light-emitting diode. This small screen allows for real-time feedback and interaction with projects.

Momentary Switch: when pressed, they either complete a circuit (a normally open switch) or “break” the circuit (a normally closed switch). The button returns to its normal position when it is released.

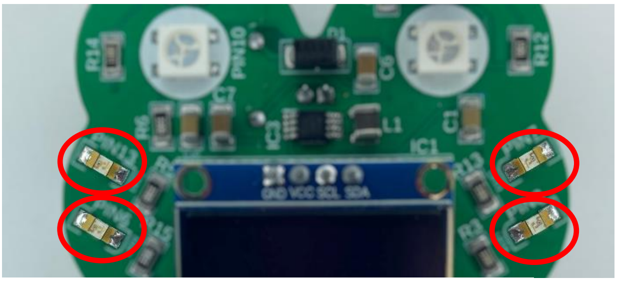


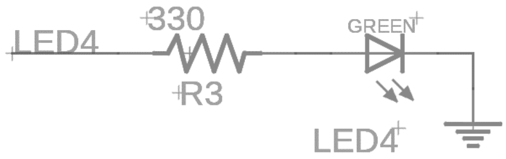
This chapter covered the basics of electronics and introduced some common electronic components. The next chapter examines how these components make the MC Trainer board work.

Circuits

Let’s look closer at the circuits on the board that we can use in our sketches. Each circuit is connected to the microcontroller through one or more I/O pins.

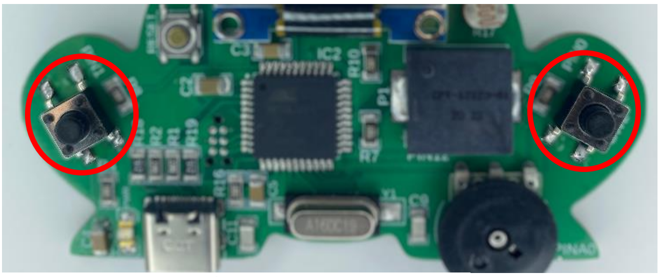
Circuit 1: Single LEDs

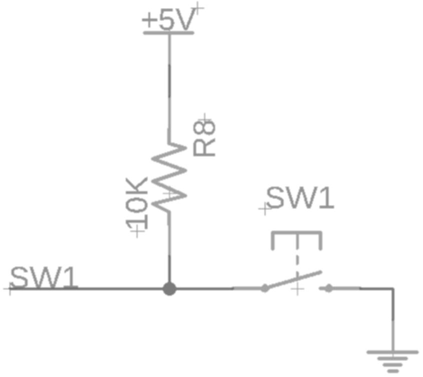




Four single LEDs are controlled by digital pins 8, 13, 6, and 7. There are four separate circuits, but they all do the same thing. Each LED is controlled by programming the pin on the 32U4 chip to enter a **HIGH** or **LOW** voltage state. The LED is turned on when the pin voltage is set to **HIGH,** which creates a difference in electrical potential between the pin and ground. This causes current to flow through the LED.

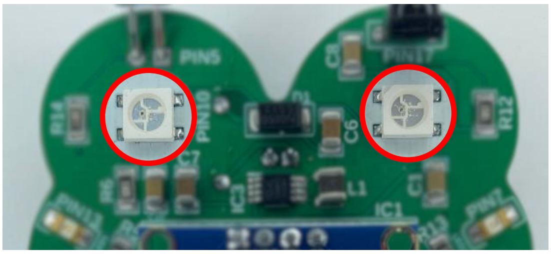
Circuit 2: Momentary Switches (Buttons)

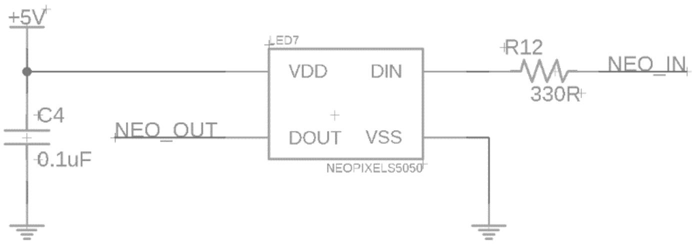




The MC Trainer has two momentary push buttons. SW1 is on the left side of the board and is measured by digital pin 1, and SW2 is on the right and measured by pin 0. The switches are normally open. A 10K resistor acts as a “pull up,” which weakly pulls each pin to 5V so that they consistently will read a digital **HIGH** when the switch is open. Pushing the switch moves the pin to digital **LOW**.

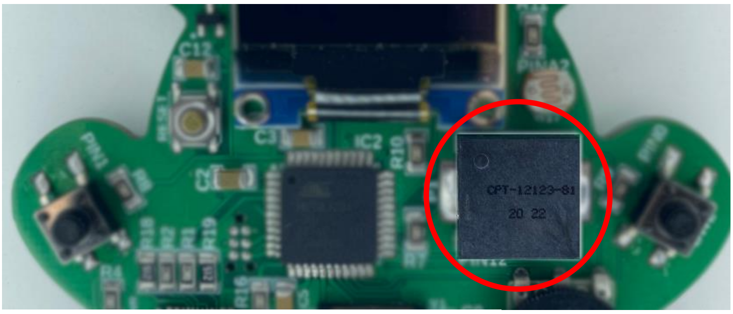
Circuit 3: Neopixels

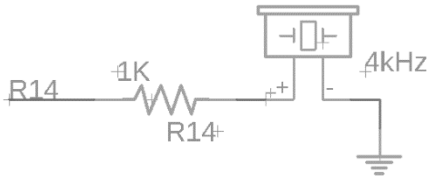




Digital pin 10 on our MC Trainer is interfaced with a WS2812B, an individually addressable RGB LED. The WS2812B is a smart LED incorporating a light-emitting part and a control circuit into a single package. When the MC Trainer sends signals to the WS2812B via digital pin 10, the LED responds by producing colored light, with the color determined by the signals it receives. Because of the integrated control circuit, each WS2812B LED can be individually controlled even when many LEDs are chained together, creating complex, multicolor patterns, and effects.

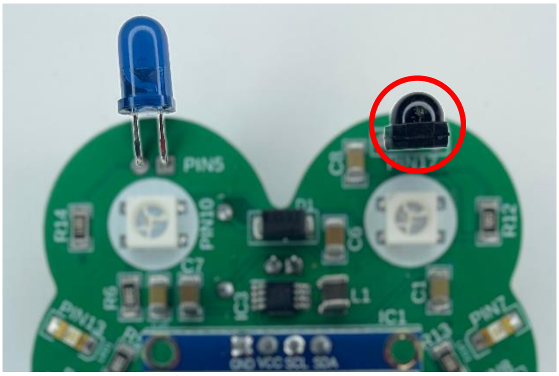
Circuit 4: Piezo Element

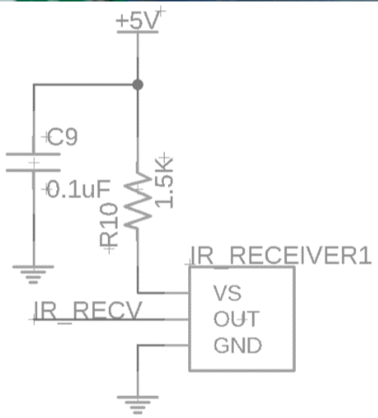




This is a simple circuit. The element is powered by digital pin 12. A 1k resistor limits current to protect the pin and device. The pin is rapidly switched between digital **HIGH** and **LOW** to make it vibrate and emit a tone.

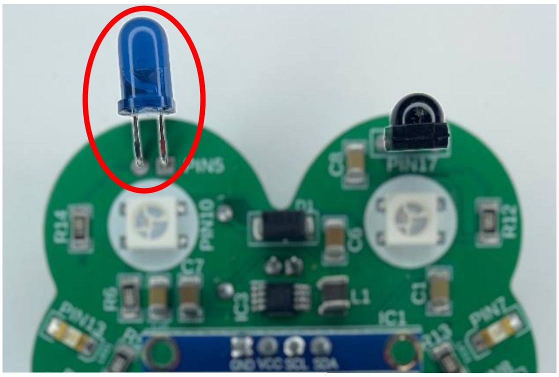
Circuit 5: Infrared Receiver

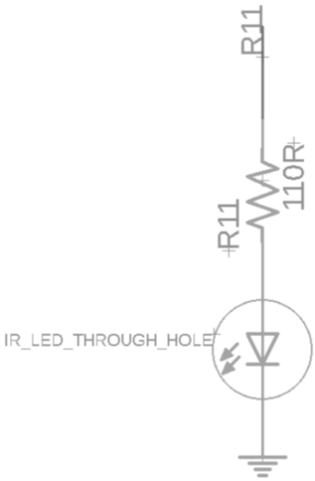




Digital pin 17 on our Arduino Leonardo is connected to an infrared (IR) receiver. The IR receiver is a specialized electronic component designed to sense infrared light which falls beyond the range of human vision. This IR receiver detects infrared signals, transforms them into electrical signals, and forwards them to the MC Trainer. When an IR signal is detected, it influences the output voltage of the receiver.

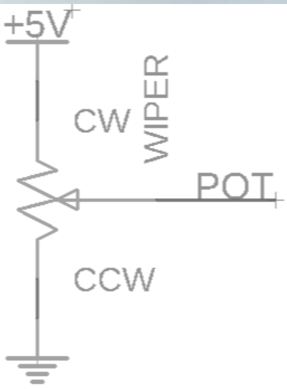
Circuit 6: Infrared Emitter





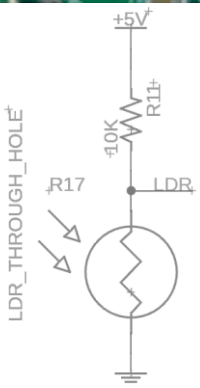
Digital pin 5 on our microcontroller is connected to an infrared (IR) LED. An IR LED, similar to a normal LED, emits light when it is energized. However, unlike a regular LED, the light emitted from an IR LED is invisible to the naked eye as it falls within the infrared spectrum. When digital pin 5 sends a HIGH signal to the IR LED, it turns on and emits infrared light. Conversely, when the pin sends a LOW signal, the IR LED turns off and stops emitting light.

Circuit 7: Thumbwheel Potentiometer



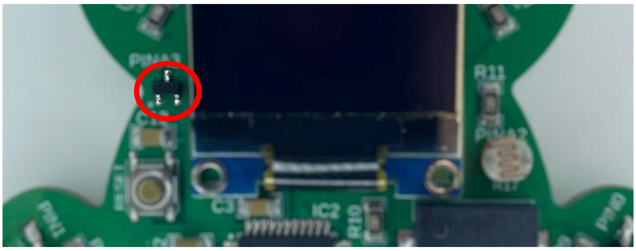
Analog pin A0 on our microcontroller is interfaced with a potentiometer, a type of resistor that allows for adjustable resistance. The unique characteristic of a potentiometer is that its resistance varies as you turn its knob or slide its lever. As the resistance of the potentiometer changes, the voltage observed at pin A0 also adjusts. This change in voltage can be measured and interpreted by the microcontroller to understand the current setting of the potentiometer.

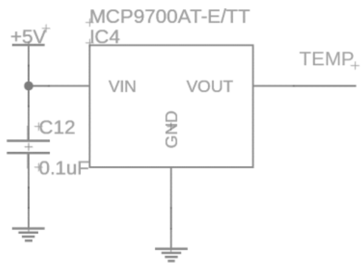
Circuit 8: Light Dependent Resistor



Analog pin A2 is connected to a light-dependent resistor (LDR), a component that varies its resistance in response to changes in light intensity striking its surface. This behavior of the LDR, sometimes referred to as photo resistivity, can be used as a gauge to measure and monitor ambient light conditions. As the resistance of the LDR fluctuates with light intensity, so does the voltage across it. By reading this voltage with analog pin A2, we can gain an insight into the ambient light level.

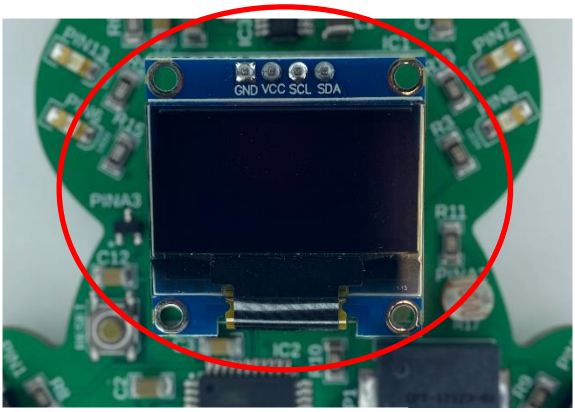
Circuit 9: Temperature Sensor

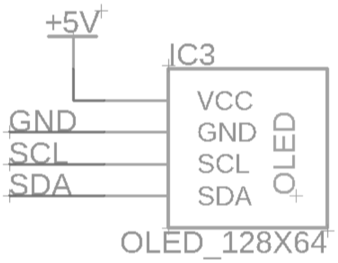




Analog pin A3 on our microcontroller is connected to the MCP9700AT-E/TT, a linear active thermistor IC. Much like how a light-dependent resistor (LDR) changes its resistance with varying light conditions, the MCP9700AT-E/TT alters its voltage output relative to the temperature it detects. We can accurately deduce the ambient temperature by monitoring the voltage change at pin A3 (connected to VOUT). This is possible because the MCP9700AT-E/TT has a linear output, meaning that a predictable and consistent voltage change corresponds to each degree of temperature change

Circuit 10: OLED





The SSD1306 OLED display module is a compact screen that serves as a valuable interface for real-time communication and feedback in your projects. This display is connected to our MC Trainer via two specific lines, SDA and SCL, which are used for data transfer and clock synchronization, respectively, as per the I2C (Inter-Integrated Circuit) protocol. The I2C protocol enables multiple devices to communicate using two wires, simplifying the wiring process and saving precious GPIO pins on the MC Trainer. The unique I2C address for our SSD1306 module is 0x3C in hexadecimal, which allows the MC Trainer to recognize and communicate with it amongst any other I2C devices that might be connected.

Next Steps…

You're not done learning Arduino at this point, you're just beginning. You've learned the basics of electronics and how to write programs. You've gone through the projects that are built-in to the MC Trainer circuit board. Hopefully, you have experimented with these projects, making changes to learn more about how you can use Arduino to interact with the physical world.

Now it's time to *create.* Think of yourself as an artist. Your palette contains computer codes, wires, and components. These are the tools you can now use to create the next project. Undoubtedly, you will still need to learn new things.

Like all artists, you will need supplies. You can do a lot more with the MC Trainer than what we covered in this book. But eventually you will need to get more components and an Arduino board with headers to allow you to plug in wires. There are lots of kits available that include an Arduino board, a solderless breadboard, jumper wires and components. You can also put together your own set of supplies. It's a good exercise to start looking for specific parts rather than just taking whatever happens to be in the kit.

You will also soon need to buy a soldering iron. I helped a friend start with Arduino recently and he said that he enjoyed programming but would never buy a soldering iron. He bought one within a few weeks. You don't need to spend a lot of money on a soldering iron, but you should buy one with temperature control. Again, search Arduino forums and other websites to find recommendations on a soldering iron.

One of the best ways to keep learning is by coming up with a project that you can't do. Not yet at least. Pick something that's over your head. Then start thinking about how you can break that project down into smaller pieces. What sensors, motors, servos, displays, gears, and wheels are available to make your project sense the world and react in an interesting way? Start searching the internet. Talk with other people about what they are doing, or what they would like to do if they had the right knowledge and tools.

Tackle one part of the project at a time. Figure out how to read a sensor and output the results to the serial monitor. Remember to make your code portable and reusable. For example, you can create a function to read the sensor and pass the result to the next part of the sketch where you decide what to do with the information. Now start putting the pieces together. The first completed version of the project will probably fall short of what you imagined (I *still* don't have a hovering robot that can follow me around). But you will keep learning. It won't be long before you'll be amazed by how far you've come.

Finally, don't forget to give back to the Arduino community. You can do this by posting code to forums or videos to YouTube. Functions can be turned into libraries and made available to other users. This is your time to contribute and show off a little. The virtual community of Arduino is also turning into a physical community as people meet for "Arduino Night" at Hacker spaces and other venues. It won't be long before it's your turn to introduce someone new to the world of Arduino.

MC Trainer Pin Key

Pin	Element	Description
0	SW2	Push Button Switch 2
1	SW1	Push Button Switch 1
5	IRPin	Infrared Emitter
6	LED2	Single LED
7	LED3	Single LED
8	LED4	Single LED
10	dataPin	Neopixel Data In
12	piezoPin	Transducer
13	LED1	Single LED
17	IRRecv	Infrared Receiver
A0	potPin	Thumbwheel Potentiometer
A2	LDR	Light Dependent Resistor
A3	tempSensorPin	Temperature Sensor