C

Programming

with

The Programmable Box!

V1.0

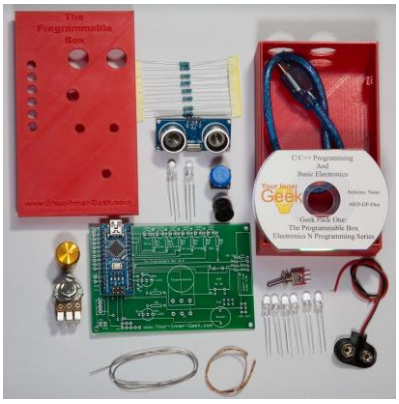
1 CONTENTS

2 WELCOME

2.1 THE GEEK PACK™: ELECTRONICS N PROGRAMMING SERIES

Thank you for purchasing this electronics and computer programming educational product. This is the most "hands-on" of our two series of Geek Pack™:

- Electronics N Programming Series where you solder the electronic components to a Printed Circuit Board (PCB) and then program the micro controller.



- Programming Series where the Programmable Box comes completely assembled and tested – no soldering is required – just plug into your computer and start programming.

Your Geek Pack™ contains the following components:

- 3D printed case + cover
- Printed Circuit Board (PCB)
- Arduino compatible Nano microcontroller
- Ultra-sonic range detector
- 10K or 25K potentiometer, nut, knob
- Business card sized CD with manual and programs
- USB-mini USB cable
- 9v battery clip with wires
- Single Pole Single Throw (SPST) toggle switch
- 6 LEDs (white, green, blue, yellow, orange, red)
- Buzzer
- SPST Momentary Contact switch with round colored push button
- Two dual color (red-blue) LED's
- 8 x 470 ohm resistors
- 24" solder
- 6" solder wick

The microcontroller and range detector come soldered to the PCB

Contained in anti-static plastic bag

2.2 TOOLS YOU WILL NEED WHILE COMPLETING THE ELECTRONICS N PROGRAMMING SERIES

You really only need tow tools: small diagonal cutters and a soldering iron. We provide the solder, and solder-wick (just in case you need to remove a device), and all the components requried to complete construction and program.

2.3 HOW THIS CURRICULUM IS STRUCTURED

In the Electronics and Programming Series are going to move between four distinctly different activities:

1. Learning about electronics devices
2. Building circuits with our electronic devices (PCB assembly)
3. Learning about software design and the C/C++ programming language
4. Writing programs using the C++ language that can interact with the electronic we have constructed.

We try to move from activity to activity fairly quickly so that before you can get bored, you are on to the next activity. Concepts are introduced just before they are used so they can remain fresh in your mind. For example, rather than introducing a long list of data types, we introduce integer (int) data type by itself first, just prior to our using it in a program.

3 LEARNING OBJECTIVES

3.1 C++

- How to set up your C++ development library and connect to your device
- The basic components of every C++ Arduino program
- System functions vs. user defined functions
- Declaring and defining a function vs. executing a function
- Naming conventions that make things less confusing
- Constants and variables
- Declaring vs. initializing variables
- Return values and void
- Data types: Integer, BOOLEAN, long, void, volatile
- Control flow (if, else, for)
- Collecting input from the keyboard, printing output to a screen
- Arrays (one, two, and three dimensional arrays)
- Arithmetic operators, comparison operators
- Interrupts

3.2 ARDUINO

- Analog and digital pins
- Input and output pins
- Defining a pin as either input or output
- Setting the state of an output pin High or Low
- Reading the state of an input pin
- Sketches and shields
- Arduino functions: setup(), loop(), if, if…else, for, pinMode(), digitalWrite(), digitalRead(), analogRead(), analogWrite(), tone(), noTone(), delay(), and more…
- How to enable and handle interrupts

3.3 ELECTRONICS

- Basic electricity concepts E=I/R
- Resistors, what they are and how to read their value
- LED's – what they are, how to connect them
- Drawing and reading schematics
- How to wire switches to the Arduino so you can detect their state
- Connecting a miniature speaker/buzzer so you can play notes using your Arduino
- Ultra-sonic range detector: theory and practice, how to wire it and read distance
- Potentiometers: what they are and how to wire them to the Arduino so you can sense their position

4 LET'S GET STARTED

4.1 THE ARDUINO NANO

The Arduino project has created an entire family of open sourced micro controller products as well as an Integrated Development Environment (IDE) for that family. The entire project is open sourced which means it is available to anyone and numerous manufacturers make Arduino compatible products so the prices are very competitive. Two of the most popular Arduino boards are the Uno and the Nano. In order to keep the size of the box small enough to hold in your hand, we chose the Nano as the processor for this Geek Pack™.

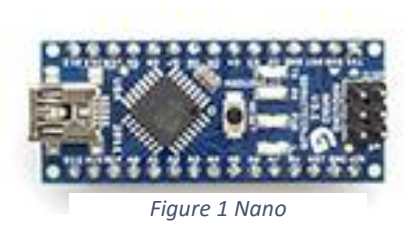


Figure 1 Nano

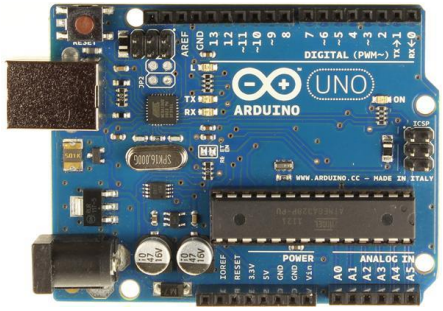


Figure 2 Uno

4.2 SETTING UP THE C++ DEVELOPMENT ENVIRONMENT ON YOUR COMPUTER

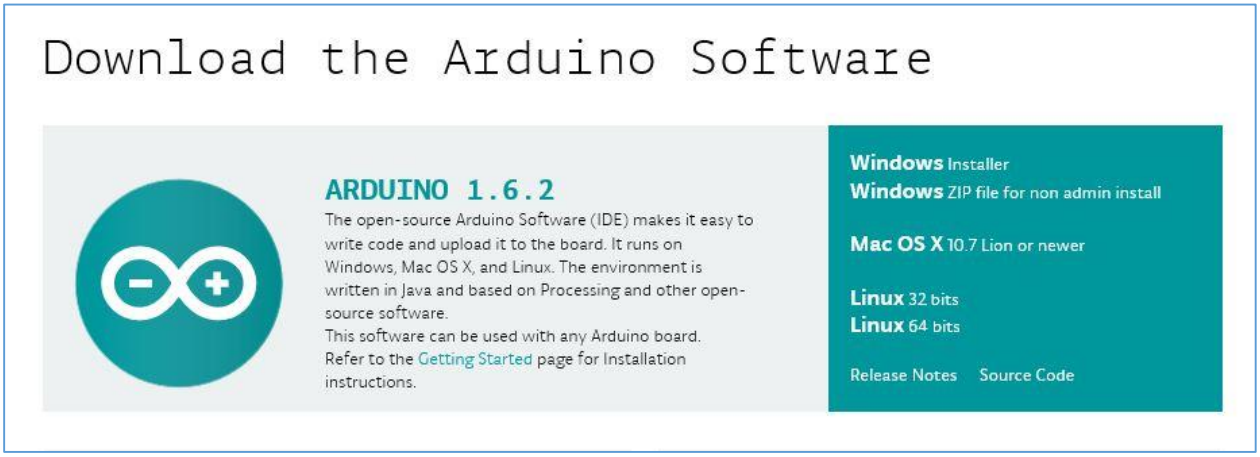
The first thing we need to do is to download and install the Arduino Integrated Development Environment (IDE). This is a visual environment which you will use to write and debug your code. You will also use this environment to transfer your code – in Arduino readable form – to your Arduino microcomputer. In our case, this will be the Nano processor.

After you transfer your code to the Nano, it will run (execute) on the Nano but you can use the IDE to have your program write information to a screen on your computer. This will be useful in making sure that your program is doing what you intended it to do.

The Arduino organization (http://Arduino.cc/) has a wonderful website and with thousands of people all over the world contributing programs, tips, and ideas for stuff to build, you will certainly want to spend some time there. Complete, up to date, instructions for installing the Arduino IDE can be found on this site but the process as of the time of this writing is as follows:

Once you navigated to the Arduino home page, you will see something like the screen below. Click on the "Download" tab so that you can select and download the Arduino IDE.

After selecting “Download” you will be asked to choose between Windows, Mac OS X, or Linux environments. Select your environment, download, and install the IDE.



4.3 CONNECTING YOUR ARDUINO FOR THE FIRST TIME

When the download is complete, plug one end of the USB cable into the Nano and the other end to a USB port on your computer. Verify that the “power” LED is on (RED) and that the “pin 13 LED” is blinking on and off. We ship the Nano pre-loaded with the blink program so this program will begin to run and blink the LED as soon as power is applied via the USB cable.

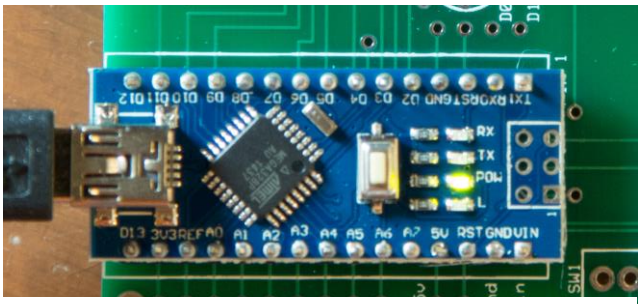


Figure 3 Nano with power LED lit

Next, navigate back to the Arduino home page and select Learning->Getting_started and then click “Arduino Nano” from the list of devices on the right hand side of the page. When following the instructions found there, your device is the Nano 3.0/ATmega328. Depending on what operating system and what version, you may not need to do anything except launch the Arduino IDE.

Launch the IDE using the Arduino icon and we are ready to begin. 

Your IDE screen should look something like the image below (Windows).

Select Tools->Board->Arduino Nano w/ATmega328. Now select Tools->Serial Port and then select the serial port you have plugged into.

Now the bottom right of your IDE screen should say something like “Arduino Nano w/ATmega328 on Com5” or whatever Com port you connected to.

So now you should have:

- Your Arduino Nano plugged into your computer
- You should have the “Power LED” illuminated on your Nano board indicating it has power
- Your IDE is launched and is configured for the Nano and the correct USB port.

Before we start writing our own programs (called Sketches in Arduino land) let’s go through the steps to get an existing program ready and uploaded to the Nano.

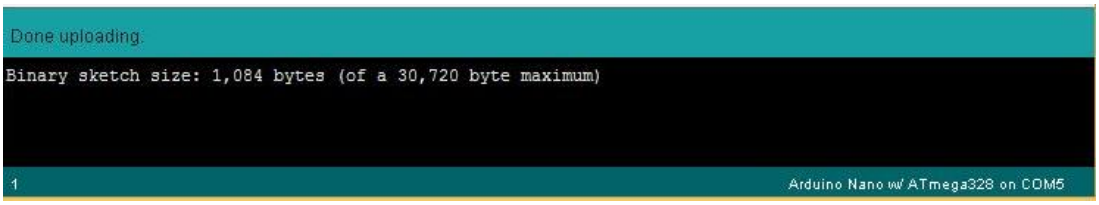
In the Arduino IDE, select Files->Examples->0.1_Basics->Blink. This program will blink the built in LED contained on the Nano board ON for one second and then OFF for one second. This LED is connected to pin 13 on the Nano and is sometimes called the pin 13 LED.

We will go over this program line-by-line shortly but for now we just want to go through the steps of Compiling (called Verify in Arduino land) the code into a form that is understandable by the Nano and then Uploading the resulting code to the Nano and running it.

This will give us positive confirmation that we were able to do all the steps required to successfully compile (verify), upload, and execute our code on the Nano processor.

On the top left of the IDE screen you should see a “check mark”. If you hover your mouse over it a text line should appear saying “Verify”. Clicking this Check Mark will compile (verify) your code into a form that can be understood by the Nano.

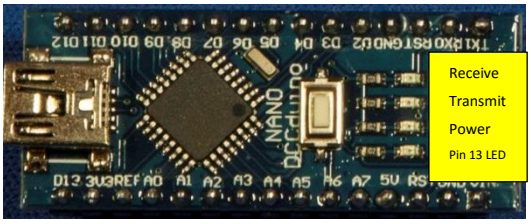
Click the Check Mark now and the bottom window of the IDE should say something like:



If there had been any errors, they would be shown in this bottom window (in color) but since this is a supplied program, there should be no errors.

Next to the Check Mark is a “Right Arrow”. Clicking on the “Right Arrow” will Compile (Verify) and then Upload the Program (Sketch) to the Nano. Click the “Right Arrow” now and watch the LED’s on the Nano.

The Transmit and Receive LED’s should blink on and off as the program is being Uploaded, the Power LED will stay on, and after the program has been Uploaded, the LED to the Right (or Below) the Power LED should blink ON for one second and then OFF for one second.



Before we move on, let’s prove to ourselves that we really are running the Blink program on the Nano. The way we will do that is by making a change to the program, uploading it to the Nano, and seeing the change to the blinking LED.

Look at the Blink Program (Sketch) and find the two lines that say “delay(1000)”. The “delay(value) function tells the Nano to wait (delay) 1000 milliseconds (ms). Since a ms is 1/1000 of a second, delaying a 1000 ms is the same as delaying 1 second.

Let’s change both of those lines to say: delay(2000);

Click on the “Right Arrow” button to compile (verify) and Upload. And now the LED should blink ON for 2 seconds and the OFF for two seconds.

There, we did it! We made a change in the program, Uploaded and ran the program and it did what we wanted. Now let’s learn some things about C++ and what each line of the Blink program is doing.

5 INTRO TO C/C++

As we mentioned previously, the Arduino web site has a wealth of information. For example, if you go to http://arduino.cc/ and then select Learning->Reference you will see the Arduino Language Reference. Although it may appear somewhat intimidating at the moment, keep referring to it and in no time at all it will make complete sense to you.

5.1 STATEMENTS

Programs (Sketches) in C++ are comprised of "digital sentences" called Statements. A statement in computer language is like a sentence in English – there are grammar rules defining a proper English sentence and likewise there are rules for writing a C++ Statement.

In English, a sentence ends with a period (.) and in C++ a statement must end in a semicolon (;). You can continue a C++ statement on more than one line if you need to but you must end the statement with a (;).

```
A Statement must end with a semicolon (;)
```

5.2 COMMENTS

It is quite helpful to add Comments to your code so that you or someone else can understand what you were trying do if you come back to make changes later on. Comments are for humans – not for the computer – they help humans read the code but the computer skips right over them.

In C++ you can start and end Comments in two different ways. If you insert a "slash asterisk" (/*) in your code, then EVERYTHING after that is a Comment until the computer sees "asterisk slash" (*/).

For Example:

```
/*
All of this is a comment
 */
```

The second way to identify a Comment is with "slash slash" (//). This tells the computer that THE REST OF THIS LINE is a Comment.

For Example:

```
Here is a statement;      // this is a comment
```

5.3 DATA TYPES

In a computer, information is represented as Binary Digits (bits). A Binary Digit (Number) can only take on one of two values: zero or one, high or low, +5 volts or zero volts. Data that can only take on two values (HIGH/LOW, true/false, 1/0) are called "boolean" data type and CAN be represented with a single bit[1].

The characters (char) that are on this page are generally represented in a computer using what is called the ASCII code (American Standard Code for Information Interchange) which requires 8 bits to represent. A collection of 8 bits is called a byte and is the most common unit of computer storage.

Although a single byte is all that is needed to represent a character (char), multiple bytes are required to represent numbers. The more accurate we want the number to be, the more bytes will be required to store and represent it.

A third very popular data type is the integer (int) data type. An integer is a whole number (1, 2, 5, 100) not a real or floating point number (1.2, 5.6, 8 ½, 3.1415).

Before we can use a computer to store and manipulate data, we need to inform the computer what data type we intend to use so that the proper amount of memory can be reserved to store our data. It takes a lot more memory to store a high precision floating point number than it does to store a boolean variable that can only take on two states: true and false.

In this Knowledge Pax, we will be using the following data types[2]:

Data types: boolean, int, char, void

This last type, void, is used to indicate that no memory needs to be reserved because we are not going to store anything. (This will make more sense as we progress.)

5.4 DECLARING VARIABLES

Variables are the lifeblood of computer programming. We use variables anytime we need to keep track of something that will change.

For example, if we want to do some task 10 times, we need a counter variable to keep track of how many times we have already done the task. Since the value of such a counter would only need to take on values of 1, 2, 3,…10 we would want the variable to be a int data type.

Before we can use this variable, we have to declare its name and that its data type is going to be int. We do that as follows:

[1] Although a boolean variable CAN be represented using a single bit, in most cases an entire 8 bit byte is used.
[2] There are many more data types but these are the ones we will be using in this Knowledge Pax.

```
//      declare the variable taskCounter as an int
int     taskCounter;
```

We should stop at this point and talk about "programming conventions". Programming conventions are not rules. Your program will still work fine if you don't follow the conventions BUT other people will have a harder time reading your code.

For example, convention says that a variable name should be descriptive and can use more than one word if required with no spaces between the words.

Our variable "taskCounter" clearly tells people that it is a Counter and it is counting the number of times a task is completed.

Convention also dictates that the first letter of all words except the first be Capitalized so that words stand out. See how taskCounter is easier to read than taskcounter?

Likewise, if we wanted a variable to indicate that the task completed successfully or not we could declare:

```
boolean     success;        // true or false
```

The four data type names we just introduced (boolean, int, char, void) are called Language Keywords. They are reserved words and can-not be used by the programmer for other purposes. Also note that CASE MATTERS when using these words. These words are all lower case.

Convention also says that CONSTANTS are named using all CAPITAL LETTERS so BUZZERPIN would be assumed to be a constant whereas buzzerPin would be assumed to be a variable that can change.

5.5 ASSIGNING A VALUE TO A VARIABLE

So far we have picked a name for some variables and determined what data type they should be but we have not assigned a value to them. The compiler can reserve memory for them since it knows their type and hence how much memory they require. And the compiler has assigned a name (our variable name) for that memory location so it knows where to put a value when it gets one. Let's look at how we can assign a value to a variable.

In computer programming, it is important to assign an initial value to all variables. The act of Declaring a variable and data type DOES NOT initialize the variable to any particular starting value.

The easiest way to initialize a variable is to do so at the time of Declaration. For example[3]:

```
int         taskCounter = 0;
boolean     success = false;
```

The first statement Declares that the variable named taskCounter is an int data type and ALSO Assigns the value of 0 to it using the Assignment Operator "=".

There is a very important distinction between the equals sign in math and the Assignment Operator (=) in C++.

In math, the equation:

```
i = i +1
```

would not make any sense. There is no value of "i" where the statement would be true.

In C/C++ (and other programming languages), the statement:

```
i = i + 1;
```

makes perfect sense and is quite useful. In C/C++, this statement says:

"Perform the calculation on the right hand side of the Assignment Operator (=) and store the result in the variable on the left hand side of the Assignment Operator."

In other words, the computer will take the value in the memory location named "i", add "1" to it, and put it back in the same memory location named "i".

5.6 C/C++ PROCEDURES AND FUNCTIONS PROVIDED SPECIFICALLY FOR THE ARDUINO

Before we move on to understanding each line of the "Blink" program, there are a few Arduino specific items we need to cover.

Throughout this Geek Pack™, we will be referring to Procedures and Functions interchangeably. Both contain one or more Statements, may require input variables/parameters, and may return a value when they complete their execution. Although there are some subtle differences between a Procedure and a Function, those distinctions will be covered in a more advanced Geek Pack™.

[3] Note: Spaces, Tabs, white space are all ignored and can be used by the programmer to make the code more readable.

In Arduino C++, every Program (Sketch) must contain two Procedures:

setup()	loop()

The “setup()” procedure is used to do things that only need to be done once. For Arduino programmers, setting up the I/O (Input/Output) pins as either Input or Output would be a good example of something that only needs to be done once, at the start of the program.

The second procedure, “loop()” does just as the name implies; the microcontroller continuously executes the statements contained within the “loop()” procedure. When the end of the “loop()” procedure is reached, it starts over again at the top[4].

We saw earlier that Variables need to be Declared and Assigned an initial value (initialized). In the case of Procedures, they also have to be Declared and instead of initialized, they need to be defined (e.g. the Statements that are to be executed within the Procedure need to be included as part of the Procedure Definition. Although these two activities CAN be performed separately (e.g. you can Declare a Procedure now and Define it later), in this Geek Pack™, we will stick to performing both operations in a single step – much as we did for Declaring and Initializing our variables earlier.

In order to Declare a Procedure, the programmer must specify what if any value is returned (using the Data Types we previously discussed), the name of the Procedure must be provided, and any input Parameters must be specified. In the case of the two Procedures we are discussing (setup(), and loop()), no values are returned (void) and no input Parameters are allowed. For example in order to Declare AND Define the Procedure “setup()”, we could do the following:

```
void    setup() {
        /*
              The Definition of “setup()” consists of all
              the Statements between the two curly
              braces “{“ and “}”
        */
        statementOne;
        statementTwo;
}
```

[4] It is possible to end the looping and Exit the program but we will consider that possibility in a future Knowledge Pax.

The name of this Procedure is “setup”. We can tell that it is a Procedure and not a Variable because after the name is has an opening “(“ and closing “)” parenthesis, e.g. setup(). The fact that there is nothing between the two parentheses indicates that this Procedure does not have any input Parameters.

The use of the Language Keyword “void” in front of the Procedure name indicates that it returns no value. The Procedure Definition begins with the opening curly brace “{“ and ends with the closing curly brace “}”. Notice that there is no semicolon “;” after the closing curly brace. This is because it is NOT a Statement. This is a Procedure Declaration and Definition, not a statement.

If this was some other Procedure that we were Declaring and Defining, it could accept input Parameters AND could return a value. For example:

```
boolean        myProcedure(int parameter1, char parameter2) {
               statement1;
               statement2;
               statement3;
}
```

Does the following;

- Declares the Procedure called “myProcedure”
- Indicates that myProcedure returns a boolean (true/false) value
- Accepts two input parameters: an int called parameter1 and a char called parameter2

In addition to setup() and loop(), there are a number of other definitions that are provided in the Arduino version of C++ as indicated on the Arduino Language Reference discussed earlier.

(http://arduino.cc/en/Reference/HomePage)

You should bookmark this page in your browser and refer to it often. It lists all of the predefined key words, internal functions and what parameters they accept/return and is quite valuable while writing your code.

Let’s look at the center column “Variables”, the first item “Constants”.

Many times we either want to set the value of a variable equal to a constant such as HIGH, LOW, true, false, or test to see if the variable has that value. For this reason, the Arduino version of C++ includes the following Constants Definitions which the programmer can use simply by typing their name[5]:

[5] The | symbol means “or” in this context.

```
HIGH | LOW                          // "1" or "0", 5v or 0v
INPUT | OUTPUT | INPUT_PULLUP       // INPUT_PULLUP will be
                                    // covered later in this
                                    // Geek Pack™
LED_BUILTIN                         // digital pin number for the built in LED = D13
true | false
```

Notice the right hand column marked "Functions". Since the Arduino microcontroller family contains many specialized I/O (Input/Output) pins, Functions (Procedures) have been included to control these pins. For the "Blink" program example, we will be using two of these hardware control functions: pinMode and digitalWrite. The pinMode function is used to specify that a particular pin should be programmed to act as an INPUT or OUTPUT (also INPUT_PULLUP to be covered later). The digitalWrite function is used to write either a HIGH or LOW to a pin that has been previously been programmed to be a digital output pin.

pinMode(pin, mode)

where:

- "pin" can be any valid Digital pin number for the Ardunio microcontroller being used (pin = 13 specifies the internal LED)
- "mode" can be INPUT, OUTPUT, or INPUT_PULLUP which connects the input to an internal pullup resistor tied to 5 volts

```
int       led = 13;              // the internal LED connected to pin D13
pinMode(led, OUTPUT);            // make pin D13 an output pin
```

Since "LED_BUILTIN" is defined as part of the Arduino C++ language, we could accomplish the same thing with:

```
pinMode(LED_BUILTIN, OUTPUT);          // make pin D13 an output pin
```

Now let's move on and cover the digitalWrite function:

digitalWrite(pin, value)

where:

-"pin" can be any valid digital pin number for the Ardunio microcontroller being used

- "value" can be HIGH or LOW

```
pinMode(LED_BUILTIN, OUTPUT);        // make pin D13 an output pin
digitalWrite(LED_BUILTIN, HIGH);     // make pin D13 go to the HIGH state (5v)
```

5.7 CHAPTER REVIEW

5.7.1 Questions:

Q1. What does a Program consist of?

Q2. If you want to make a note within the code to make it easier to read later, what is that called and how do you do that?

Q3. How would you declare and initialize a variable named “loopCount” that would only take on the values 1, 2, 3 and initialize it to the value of 1?

Q4. How would you declare and initialize a variable called “success” that can only take on the values “true” and “false” and initialize it to the value “false”?

Q5. How would you declare and initialize a variable called “firstInitial” and initialize it to the value of ‘a’?

Q6. What are the two procedures that are required in every Arduino C++ program?

Q7. When you are defining a procedure, what specifies the start and end of the definition?

Q8. How would you declare and define a procedure named “myName” that accepts two input parameters, a char named myChar and an int named myInt, and returns a boolean return value?

Q9. How would you set pin 13 to be an OUTPUT pin and set it to the HIGH value?

5.7.2 Answers:

A1. A Program consists of a collection of digital sentences called Statements. Each Statement may span one or more lines but must end in a semicolon (;).

A Statement can be Declaring a variable

```
int  i;
```

or can be Assigning a value to a variable

```
i = i + 1;
```

or can be calling (invoking) a previously defined Function (Procedure)

```
letsCallThisFunction();
```

A2. Making a note within a program is called adding a Comment or Commenting. There are two ways to do this in C++.

```
// the rest of this line is a comment
```

Or,

```
/*
Everything between the /* and the */ is a comment
*/
```

A3. `int loopCount = 1;`

A4. `boolean success = false;`

A5. `char firstInitial = 'a';`

A6. setup(), and loop() are required in every Arduino C++ program. Neither procedure returns any value (void).

A7. The start of a procedure definition is indicated by the open curly brace "{" and the end is indicated by the closed "}" curly brace.

A8.
```
boolean myName(char myChar, int myInt) {
  Statements;
  }
```

A9.
```
pinMode(LED_BUILTIN, OUTPUT);
digitalWrite(LED_BUILTIN, HIGH);
```

6 UNDERSTANDING THE BLINK PROGRAM

- Open the development environment by clicking on the Arduino IDE icon.

- Navigate to the blink program: File->Examples->01.Basics->Blink

6.1 LINE-BY-LINE ANALYSIS OF "BLINK"

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
 */

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:

void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:

void loop() {
  digitalWrite(led, HIGH);          // turn the LED on (HIGH is the voltage level)
  delay(1000);                      // wait for a second
  digitalWrite(led, LOW);           // turn the LED off by making the voltage LOW
  delay(1000);                      // wait for a second
}
```

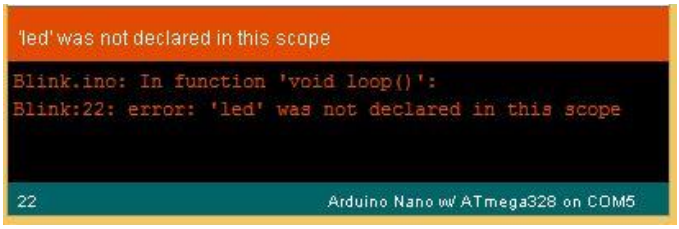
We recognize the items that are in light grey text as all being comments so let's focus on the actual code.

Just below the comment that talks about pin 13 being connected to an LED on the Arduino boards is the Statement:

```
int       led = 13;
```

We recognize this as a Statement since it ends with a semicolon (;). We also know that this is Declaring the Variable “led” to be a Data Type “int” and we see that in addition to Declaring “led” it is also Initializing it to the value of 13.

Notice that the declaration of “led” is done OUTSIDE of any procedure. A variable that is declared outside of any procedure is GLOBAL and can be used by any procedures within the program. If we had declared “led” inside the setup() procedure, we would have received an error like the following when we tried to compile (verify) because the loop() procedure would not be aware of the variable “led”.



Next we see (in addition to more comments) the Declaration AND Definition of the Procedure “setup”.

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

The Declaration part: void setup() declares that the procedure setup() returns no value (void) and accepts no input parameters setup().

The setup() procedure definition only has one statement in its body (between { and }) and that this statement sets the mode of pin 13 (the pin with the LED) to OUTPUT. Since this would only need to be performed once, this statement is placed in the setup() procedure.

We know that every Arduino C++ program has at least two procedures: setup() and loop(). The next block of code we encounter is declaring and defining the loop() procedure.

```
void    loop() {
        digitalWrite(led, HIGH);
        delay(1000);
        digitalWrite(led, LOW);
        delay(1000);
}
```

This procedure simply turns on the LED attached to pin 13, waits one second, turns off the LED and waits one second. This action is repeated indefinitely until power is removed from the Nano.

6.2 A SIMPLE MODIFICATION TO THE BLINK PROGRAM

Let's make some modifications to this program before moving on. Let's have the LED stay on a bit longer each time the loop() procedure executes – say 1/10 of a second each time through the loop. The length of time that the LED stays ON is governed by the first delay(1000); statement – the one right after digitalWrite(led, HIGH);

In order to make that delay increase in value by 1/10 of a second each time through the loop, we need to change the "1000" into a variable and then increment the value of that variable by 100 (1/10 sec) each loop.

Lets name the variable onTime and we will need to declare and define it outside of any procedure so lets add the following statement right after declaring "led".

```
int     onTime = 1000;
```

Now we need to add 100 to onTime every time through the loop. We could do this anywhere after we use it the first time in the loop but let's just have this be the last statement before we start the loop over again:

```
onTime = onTime + 100;
```

Now our program would look as follows:

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly
 */

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;
int onTime = 1000;        // onTime variable will be used to modify the ON delay time

// the setup routine runs once when you press reset:

void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:

void loop() {
  digitalWrite(led, HIGH);          // turn the LED on (HIGH is the voltage level)
  delay(onTime);                    // wait for a second
  digitalWrite(led, LOW);           // turn the LED off by making the voltage LOW
  delay(1000);                      // wait for a second
  onTime = onTime + 100;            // increase ON time by 1/10 of a second
}
```

Figure 4 Ch_6_2_blink_longer

Now verify and upload this sketch to the Nano using the "right arrow" on the IDE. Watch the pin 13 LED and notice that it is staying ON for a longer and longer time. If we let the sketch continue to run, the LED would continue to stay on longer and longer each time through the loop.

6.3 USING THE "IF" STATEMENT TO MODIFY THE BLINK PROGRAM

What if we wanted to modify the program such that:

- The LED starts by being ON for a period of time onTime = 1000ms
- The LED is always OFF for a period of time offTime = 1000ms
- Each time through the loop, the period of onTime is increased by increaseTime = 200ms UNTIL onTime reaches a value of 3000ms. After reaching onTime = 3000ms and the LED has been on for

3000ms, the next time through the loop, change onTime to 1000ms and start increasing onTime all over again.

So how would we do this?

Give this some thought before continuing. Some of this you may already be able to do. For example, from the text above we know we should add the following statements (outside of any function).

```
int     offTime = 1000;         // set offTime to 1 second
int     OnTime= 1000;           // initial value of onTime
int     increaseTime = 200;     // increment value of 200ms
```

But how do we determine when onTime = 3000ms? For this we need to introduce the concept of Control Flow and one of the Control Structures that we can use to perform different actions based upon the value of a variable.

The Control Structure we are going to learn first is the "if" statement and it's close relative the "if", "else" statement.

The basic syntax for an "if" statement is as follows:

```
If(test) {
        // if the test is true, execute these statements
}

        /*      if the test is false, do not execute the statements
                between { and }
        */
```

The "test" can take many forms. Since you are testing to see if the "test" is true, you can test a boolean variable simply by placing the name of the boolean variable between the ().

```
boolean          success = true;

if(success) {

// execute these statements if success is true

}
```

You can also compare two variables and test if they are:

x == y (true if x is equal to y)
x != y (true if x is not equal to y)
x < y (true if x is less than y)
x > y (true if x is greater than y)
x <= y (true if x is less than or equal to y)
x >= y (true if x is greater than or equal to y)

Note that "==" is the TEST to SEE if two variables are equal. This is completely different than the "=" assignment operator that ASSIGNES the value on the right to the variable on the left.

Let's look at the code that turns the LED on and off:

```
digitalWrite(led, HIGH);                // turn the LED on
delay(onTime);                          // wait for a onTime ms

digitalWrite(led, LOW);                 // turn the LED OFF
delay(offTime);                         // wait for a offTime ms

onTime = onTime + increaseTime;         // increment onTime by increaseTime ms

// since we just incremented onTIme, if it had been 3000, it would now be > 3000

If(onTime > 3000) {
        onTime = 1000;
}
```

Your completed code should look something like what is shown on the next page. Verify and upload your program and confirm that the LED stays on longer and longer until it reaches three seconds and then starts back at one second.

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.
*/

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int  led = 13;

int  onTime = 1000;              // initial value of time to stay ON
int  offTime = 1000;             // initial value of time to stay OFF
int  increaseTime = 200;         // increase onTime by 200 ms each time through the loop

// the setup routine runs once when you press reset:

void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:

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

  onTime = onTime + increaseTime;        //  increment onTime by increaseTime ms

  if(onTime > 3000) {
    onTime = 1000;
  };
}
```

Figure 5 Ch_6_3_blink_with_if

6.4 USING THE "IF ELSE" STATEMENTS FOR MORE PRECISE CONTROL

When using the "if" statement by itself (without the "else" statement), we can execute some code if the "test" is true but we don't have the ability to execute some other code only if the test is false. This is why we had to increment onTime BEFORE we did the "if" statement and then test to see if we had gone too far.

Sometimes what you really need is to execute one block of code if the "test" is true and a different block of code if "test" is false. In this case, we need the "if", "else" statements working together as follows.

```
If(test) {
        trueStatement1;         // execute these statements if the
        trueStatement2;         // test is true
}
else {
        falseStatement1;        // execute these statements if the
        falseStatement2;        // test is false
}
```

Using the "if", "else" control structure, we could re-write the test for 3000 ms as follows:

```
digitalWrite(led, HIGH);                // turn the LED on
delay(onTime);                  // wait for a onTime ms

digitalWrite(led, LOW);                 // turn the LED OFF
delay(offTime);                 // wait for a offTime ms

If(onTime == 3000) {            // notice "==" NOT assignment "="
        onTime = 1000;
}
else {
onTime = onTime + increaseTime;         // increment onTime by
increaseTime ms
}
```

Figure 6 Ch_6_4_blink_with_if_else

Before we move on to other topics we should mention something about syntax. So far, we have always used left and right curly brackets around the statements to be executed if “test” is true or false:

```
if(test) {
        // test true statements
      } else {
      // test false statements
      }
```

In the case where there is only ONE statement to be executed, that statement need not be bracketed by curly braces. For example:

```
if(test)  doThis();
else      doThat();
```

6.5 CHAPTER REVIEW

6.5.1 Questions

Q1. If we want to have a variable available for use within the “loop()” procedure, where would we declare and initialize it?

Q2. What does the statement “onTime = onTime + 100;” do?

Q3. When will the statements contained within the curly braces be executed?

```
if(test) {
    Statements;
}
```

Q4. Do you always have to have curly braces when using the “if” and “else” control structure?

6.5.2 Answers

A1. You must declare and initialize it outside of any procedure. If we declare it inside setup() it will NOT be available from within loop(). We could declare it inside loop() but if we initialize it within loop() it will get re-initialized each time through the loop.

A2. The right hand side of the assignment operator "=" is evaluated (e.g. add 100 to onTime) and the result is stored in the variable on the left hand side of the assignment operator (e.g. onTime).

A3. When the evaluation of "test" is true. The test can be to see if a single boolean variable is true or can compare two equations to see if they are equal, not equal, greater than, less than, greater than or equal to, less than or equal to.

A4. No. If there is a single statement to be executed for either the "if" or the "else", it can be placed after the "test" without the use of the curly braces.

```
if(test)    doThis;

else        doThat;
```

7 ASSEMBLING THE PRINTED CIRCUIT BOARD

7.1 COMPONENT LAYOUT ON THE PCB

Hold the PCB so that the Nano processor is in the upper left corner and the ultra-sonic range detector is on the right. The top center of the PCB will contain the 6 different colored LED's and between each LED is a resistor labeled R1 through R6. The on/off power switch is located in the lower left corner of the board and the 2 dual-colored LED's are located to the right of the on/off switch with resistors R7 and R8 between them. In the center of the board is the potentiometer and to the right of the potentiometer is the momentary contact switch. In the lower right corner is the buzzer and the right end of the board contains the ultra-sonic range detector (already soldered to the PCB).

Figure 7 Completed board showing component locations

7.2 SOLDERING THE BATTERY CLIP

The 9v battery clip will be installed next. The battery clip leads are inserted from BENEATH the circuit board with the RED lead inserted into the hole by the on/off switch labeled +9v and the BLACK lead being inserted into the hold marked Gnd. Be sure to install the wire from the bottom of the circuit board and solder the wires from the top of the board. This will enable the battery clip wire to be stored in the space beneath the circuit board when you insert the board into the case.

7.3 SOLDER THE RESISTORS

Insert each of the 8 resistors into the locations marked R1 – R8 by first bending them into a "U" and inserting the leads into the PCB from the printed side of the board. Spread the leads slightly on the back of the board to keep the resistors from falling out before you solder them. To make the board look professional, line up all the colors on the resistors the same way. Resistors are not directional so it will not affect the operation of the board but it looks nice to have all the resistor colors the same direction.

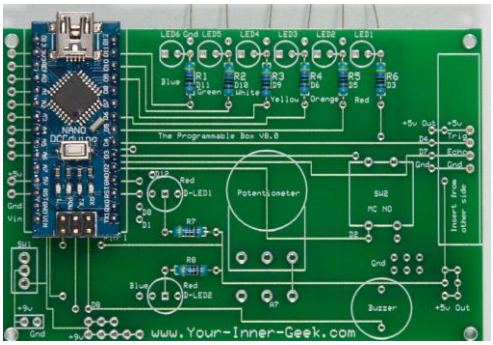


Figure 8 Resistor location - top of board

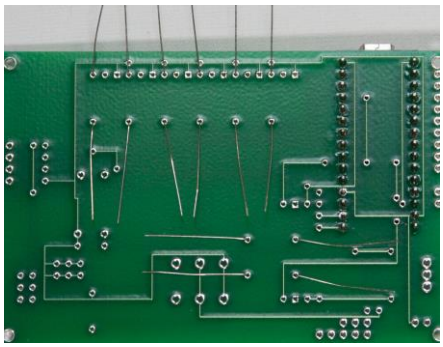


Figure 9 Bending resistor leads prior to soldering - underside of board

After you solder in the 8 resistors, cut off the excess leads from the back of the board and make sure that you don't have any solder-shorts where the solder is bridging to another pad. Now we are ready to add the LED's.

7.4 INSTALLING THE ON/OFF SWITCH

The HEIGHT of all of the components that attach to the printed circuit board and protrude through the top of the box is **very important**. The top has four stand-off pins that serve to set the spacing between the circuit board and the top so that all of the components are installed at the correct height.

When installing the on/off switch, you can either:

(1) Mount the switch to the top (photo to the right) with the nut tightened just enough to come flush with the threaded shaft. Then place the on/off switch pins into the circuit board and solder. The circuit board will be held away from the top by the four stand-off pins.

(2) Solder the on/off switch into the circuit board with the pins just barely flush with the bottom of the circuit board (see incorrect and correct photos below). There will be a gap between the bottom of the switch and the PCB when installed correctly.

Figure 10 Mount switch to top so that nut is just barely threaded completely on

Figure 11 INCORRECT - switch is too LOW

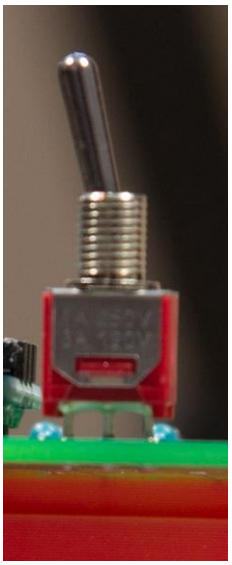


Figure 12 CORRECT - note the gap between the bottom of the switch and the PCB. The switch leads should be flush with the bottom of the PCB

When you have soldered the switch in place, verify that it is at the correct height by holding the PCB up to the top with the switch protruding through the mounting hole and verify that there is sufficient thread for the nut to be screwed on.

7.5 ADDING THE MOMENTARY CONTACT PUSH BUTTON SWITCH

Locate the Normally Open Momentary Contact (NOMC) switch and notice that in one direction the pins are spaced closer together than in the other direction. The narrow spaced pins are at the top and bottom. Also notice that there is a small curve at the end of the switch pins. These pins are only pushed into the PCB slightly – just until you reach the curve in the leads. Carefully insert the four pins slightly into the PCB and solder from the TOP of the PCB. You want the NOMC switch as high as possible from the PCB so don't force the pins into the PCB.

7.6 DETERMINE THE COLOR OF EACH LED

In order to assist in determining the color of each LED, we are going to first load a hardware test program into the Nano. On the web site or from the enclosed CD, locate the "LED-test-program" program, open it with the Arduino IDE, and upload it to the Nano. One by one, insert the LED's with two leads into a LED location LED1 – LED6 on the PCB with the long lead to the RIGHT and the short lead to the LEFT. Note what color the LED is and relocate it to the correct position on the PCB. The color of the LED that goes into each location is printed on the PCB.

Now take the two LED's that have three leads (Dual-color LED's) and insert one into each location (LED7 & LED8) with the longest outside lead to the RIGHT. All the LED's may not stay lit constantly since they are not yet soldered into place but if you wiggle the they should light up.

If you still have the PCB oriented with the Nano in the upper left corner, the dual color LED's should have the BLUE to the LEFT and the RED to the RIGHT.

Your top row LEDs should be (from left to right) Blue, Green, White, Yellow, Orange, Red. **DO NOT SOLDER ANYTHING YET.**

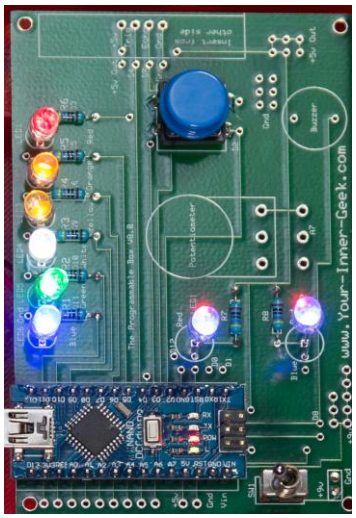


Figure 13 LED's all in correct locations - BUT NOT SOLDERED

7.7 SOLDERING THE LED'S

Now that the LED are all in their correct positions, we are ready to solder them once we get them at the right height. Place the top over the circuit board so that the on/off switch, NOMC switch, and the 8 LEDs are all protruding through the top. Hold the two pieces together with a rubber band and turn the assembly upside down so that the top is facing down.

Press each of the LED's firmly into place so that they are all pushed into the top as far as they will go and then solder them in this position.

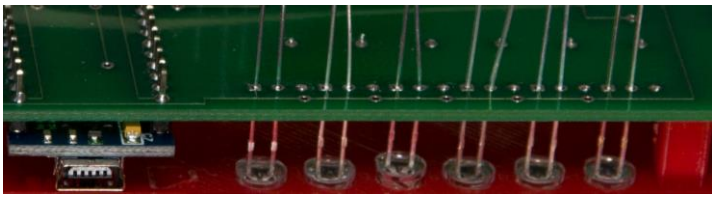


Figure 14 Push LED's until flush with TOP

Figure 15 Hold top and PCB together with rubber band while soldering LED's

Follow the same procedure to solder the dual color LED's in place flush with the top.

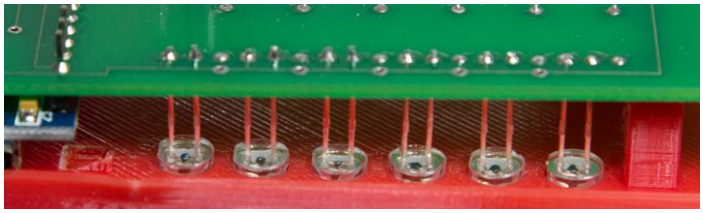


Figure 16 Solder LED's flush with TOP

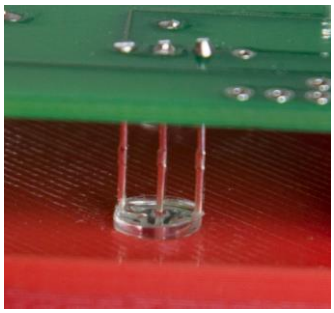


Figure 17 Dual color LED's flush with TOP

7.8 INSTALLING THE POTENTIOMETER AND BUZZER

When installing the potentiometer, we want it to be as close to the circuit board as possible. Push the pins on the potentiometer into the PCB until the narrow ends of the leads are as far into the PCB as they can go. Solder the potentiometer in place.

Figure 18 Completed board (ultra-sonic range detector not shown)

The buzzer needs to be as HIGH as possible off the PCB. On the front of the buzzer, locate the + sign (this is also the longest of the two leads) and insert the buzzer into the PCB with the + lead toward the edge of the PCB.

The short lead should only be about ½ way through the PCB so that the buzzer stands off the PCB as much as possible. The longer + lead will protrude through the PCB a bit. If the leads on your buzzer a too short, use the pin extenders included in the plastic bag to raise the buzzer up a bit. The two extenders come connected as shown in figure 19. Carefully cut them apart as shown in figure 20 and solder them into the two buzzer pin locations on the PCB. The buzzer pins will fit into those extenders and raise the buzzer up a bit.

Figure 19 Dual pin extenders

Figure 20 Pin extenders separated

Before proceeding, place the top on the circuit board and verify that all of the components line up. The NOMC switch should be free to be pressed and release and the buzzer should be aligned with the hole in the top.

7.9 TESTING THE BOARD

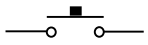
Before we start programming the Nano, let's run a test program to satisfy ourselves that everything is working correctly. From either the web site or the enclosed CD, locate and open the file "hardware_test_program". Compile and upload the program into the Nano and verify the following:

- Plays do-re-mi on the buzzer from low to high notes, reverses sequence upon button push
- Speed of playing do-re-mi determined by distance detected by ultra-sonic detector
- Lights the 6 LED's from Blue to Red, reverses sequence upon button push
- Speed of sequencing through the LEDs determined by distance detected by ultra-sonic detector
- Brightness of 6 LEDs determined by potentiometer
- Lights the dual-color LEDs Red, Red, Blue, Blue – speed determined by distance detected by ultra-sonic detector

8 WRITING PROGRAMS USING THE BUZZER AND PUSH BUTTON SWITCH

8.1 SCHEMATIC DIAGRAM FOR THE BUZZER AND PUSH BUTTON SWITCH

A schematic is a drawing that shows how your electronic components are wired up. Most electronics components have an agreed to symbol that is used to represent them when drawing a schematic. For example, this is the symbol for a Normally Open (NO), Momentary Contact (MC) switch.



The normally open part means that unless you push on the switch, the contacts are normally not connected and so no electricity will flow. You can see from the symbol that as long as you hold the switch button pushed in, the contacts will touch and electricity will flow – hence momentary contact.

For completeness, the symbol for a Normally Closed (NC) Momentary Contact (MC) switch is as follows.

The schematic symbol for a buzzer is:

8.2 PROGRAMMING - HOW DO WE CONTROL THE BUZZER

Before we proceed further, let's talk about some good programming habits for this Geek-Pack course. Whenever you are about to start writing a program, open up the IDE and create three sections:

(1) Some space before the setup() function for you to insert declarations for variables that you want to be global,
(2) the setup() function (empty for now) and,
(3) the loop() function (also empty for now). This will give you a great framework for placing the variables and statements in the proper location.

```
                                // space for declaring global variables
void setup() {
                                // space to insert statements in the setup() function
}
void loop() {
                                // space to insert statements in the loop() function
}
```

Figure 21 Create a structure like this in the Arduino IDE for every program

Now that we have this structure in our IDE, let's get started making some noise with the buzzer.

8.3 WRITING A PROGRAM TO PLAY TONES ON THE BUZZER

Now that we are going to start writing programs to control the various Input/Output (I/O) devices in our box, we are going to need to know what Input or Output port they are connected to on the Nano. The Appendix shows all these connections and we can see that the buzzer is connected to the Digital pin D8.

Since the buzzer is connected to Digital pin D8 and since we need to send a signal OUT to the buzzer, we know that we will need the following statements:

```
int         buzzer = 8;                 // the buzzer is on pin D8
                                        // define this outside of any function so that it is GLOBAL
void setup() {
            pinMode(buzzer, OUTPUT);    // set the pin D8 as an OUTPUT pin
                                        // Any other setup statements
)
void loop() {
                                        // space to insert statements in the loop() function
}
```

In order to supply a tone to the buzzer, we need to call the tone(digitalPin, frequency) function. The tone() function requires two arguments: the digital pin number, and the frequency of the tone to be played. If we wanted to play a tone of 1,000 Hz (1 Kilo Hertz) on the buzzer connected to pin D8, we would use the following statement.

tone(buzzer, 1000);

The tone() function works by making the digital pin specified (in our case buzzer = 8, so D8) HIGH and then LOW the number of times per second specified by frequency (in our case, 1000 times per second or 1 Kilo Hz).

The tone() function will continue to play a tone until told to stop by using the noTone() command. In order to not drive ourselves (and others) crazy, let's just have our program play the tone for three seconds and then stop – and not start again.

Here are the requirements for our program:

frequency = 1000 Hz

duration = 3 seconds = 3000 ms

buzzer on pin 8

only turn on the buzzer once for three seconds and then silence

Give some thought to how to approach this. What variables or constants do we need to declare and define? Where do we put statements that we want to only execute once not over and over?

Remember, so far we have learned about three places we can put statements: outside of any function, inside the setup() function, and inside the loop() function. Based upon the above requirements, what would we put outside of any function to make global? What should go in the setup() function to be performed once? What do we want to do over and over again?

Try writing this program yourself and then check the code below for a way to do this. Compile and upload your code to the Nano and see if your buzzer sounds for three seconds and then stops. If you get stuck, then, from either our web site (www.Your-Inner-Geek.com) or the business card CD, load the program Ch_8_1_Basic_buzzer_1KHz and run it on your Nano.

```
//      play a tone of 1000 Hz on the buzzer connected to pin 8 for 3 seconds and then silence
int     buzzer = 8;                 // buzzer is on Digital pin 8
int     frequency = 1000;           // play frequency of 1000 Hz
int     duration = 3000;            // duration of 3 seconds
void setup() {
        pinMode(buzzer, OUTPUT);    // set the pin D8 as an OUTPUT pin
        tone(buzzer, frequency);    // apply tone of frequency “frequency” to pin “buzzer”)
        delay(duration);            // play tone for time = duration ms
        noTone(buzzer);             // stop playing the tone on pin “buzzer”
}
void loop() {
                                    // nothing to do over and over again
}
```

Figure 22 Ch_8_3_basic_buzzer_1khz

8.4 PLAYING A MUSICAL SCALE ON THE BUZZER

Now as cool as it is to play one tone for 3 seconds, how about we play a song or something more interesting. In order to do that, we need to first introduce a new concept, an ARRAY.

An ARRAY is used to hold a group of objects. These can be characters, integers, or any other data type that the programmer desires.

An array of 6 integers named myArray can be declared as follows:

```
int     myArray[6];          // declares an array myArray that has memory to store 6 integers
```

To reference a particular item within the array myArray, we specify myArray[i] where i is the element we are interested in.

> Note: arrays go from zero to one minus the size of the array so in our example, myArray[0] through myArray[5] would reference the 6 different integers stored in myArray.

We can initialize the values in an array in a number of ways, for example:

```
myArray[3] = 10;
```

would store the value of 10 in the 4th location (remember, we start at zero) of myArray.

We can also declare AND initialize an array at the same time as follows:

```
int     myArray[] = {2,4,6,8,10,12};     // declare myArray as an array of integers and
                                         // initialize it
```

Notice that in this case, we do not have to say the size of myArray[6]. The compiler will count the number of items we initialized the array with and make that the size of the array. Handy!

So if we were to create an array where each entry in the array contained frequency for the next note in the musical scale, then by sequencing through each entry of the array we could play the musical scale.

Let's call our variables: Do, Re, Mi, Fa, Sol, La, Te, Do2 representing the notes of the scale and their values are:

Do = 440, Re = 494, Mi = 554, Fa = 587, Sol = 659, La = 740, Ti = 831, Do2 = 880

We are going to store these variables in an array called toneArray[] and we are going to declare and initialize this array at the same time.

Where and how would you declare and assign values to those variables? Using your Arduino IDE, do so now – then continue and see how you did.

One way you could have accomplished this would look like this:

```
// declare and initialize outside of any function so they are global
int Do = 440;
int Re = 494;
int Mi = 554;
int Fa = 587;
int Sol = 659;
int La = 740;
int Ti = 831;
int Do2 = 880;
int toneArray[] = { Do, Re, Mi, Fa, Sol, La, Ti, Do2};
setup() {
                    // space to insert statements in the setup() function
}
loop() {
                    // space to insert statements in the loop() function
}
```

Let's start by writing the remainder of the program such that it plays the musical scale contained in toneArray[] from the lowest note to the highest note, one time only and then silence (so don't forget to use noTone(buzzer) at the end).

What do we need to setup() in order for this to happen. Do we need to set any pins as INPUT or OUTPUT? That would be the kind of thing to do in the setup() function.

After determining what setup() stuff you need to get done once you are ready to actually start sending tones to the buzzer. What we need to do is FOR every integer value of i between i = 0 and i<8 (e.g. 0,1,2,3,4,5,6,7) we need to call the tone() function and send the frequency located at toneArray[i] to the buzzer located on pin 8.

How do you think we should accomplish that? How about using the "for" statement as follows:

Since, in this example, we only want to play the scale once, let's put this in the setup() function along with any pin setup stuff (e.g. pinMode(buzzer, OUTPUT)).

```
                                    // declare and initialize outside of any function so they are global
int Do = 440;
int Re = 494;
int Mi = 554;
int Fa = 587;
int Sol = 659;
int La = 740;
int Ti = 831;
int Do2 = 880;

int toneArray[] = { Do, Re, Mi, Fa, Sol, La, Ti, Do2};

int buzzer = 8;                                     // the buzzer is on pin 8

void setup() {
  pinMode(buzzer, OUTPUT);
        for(int i = 0 ; i < 8 ; i++) {
                tone(buzzer, toneArray[i]);         // sent the frequency toneArray[i] to the pin "buzzer"
                delay(400);
                noTone(buzzer);
                delay(100);                         // play tone for one second
        }
}

void loop() {

// space to insert statements in the loop() function

}
```

Figure 23 Ch_8_4_basic_musical_scale_array

8.5 CONSTANTS VS. VARIABLES – USING THE #DEFINE COMPILER DIRECTIVE

Before we continue with our buzzer programming, let's catch our breath with a few style and good-practice items. So far, we have declared global variables when we wanted to define items such as a pin

```
for(int i = 0 ; i < 8 ; i++) {
        tone(buzzer, toneArray[i]);       // sent the frequency toneArray[i] to the pin "buzzer"
}
noTone(buzzer);                           // stop the darn buzzer before we all go crazy…
```

number or a frequency or a delay time. Items such as frequency and delay time could truly be variables in that we could really want to change their value during the execution of our program.

For example, every second, we could increase the frequency of the tone by 500 Hz or increase the delay time. For items such as the pin number, this is probably NOT the case and these items are really constants not variables.

There are three things that programmers consider when declaring variables and constants. The first is memory use, the second is compiler type checking, and the third is called “scope”. We will explain all three issues here but will not solve the scope issue until later in this curriculum.

When you declare a variable of a certain type (int, BOOLEAN, etc.) the compiler must reserve enough memory so that the variable can be stored and accessed by the program. Sometimes, when you are running on a small microcontroller like the Nano, memory space might be at a premium so you should know that there is another way to declare items like pin numbers – constants – that do not take up extra memory space.

The way to do this is with what is called a “compiler directive’’ and one of the most popular is the #define. Using it will shed some lite on the situation.

```
#define BUZZER 8                    // notice there is no “=” sign and no semicolon “;”
```

Later in the program:

```
pinMode(BUZZER, OUTPUT);
```

In this case, the compiler makes a list of the #defines and knows that “buzzer” gets replaced with “8”. No memory is assigned for a variable. When the compiler comes across the statement

pinMode(BUZZER, OUTPUT);

it replaces it with: pinMode(8, OUTPUT);

The code is easier to read because you have all your #defines up front where you have your variable declarations but you don’t use any extra memory.

Earlier, when we introduced the array toneArray[] we showed that we could specify the size as in toneArray[6] or we could initialize the array and the compiler would count the items to compute the size. What if we did the following:

```
int size = 6;

int toneArray[size];        // trying to compile this would give an error
```

The reason is that although “size” is an int, it is also a variable – it can change – and the compiler needs a fixed size in order to allocate memory for toneArray[]. Using the #define construct, we could do the following:

#define size 6

toneArray[size]; // this will work fine since “size” is an int and also a constant

In our previous example:

This	Becomes this
int Do = 440;	#define DO 440
int Re = 494;	#define RE 494
int Mi = 554;	#define MI 554
int Fa = 587;	#define FA 587
int Sol = 659;	#define SOL 659
int La = 740;	#define LA 740
int Ti = 831;	#define TI 831
int Do2 = 880;	#define DO2 880
Int buzzer = 8;	#define BUZZER 8
Program compiled to 2,834 bytes	Program compiled to 2,660 bytes

Notice that when we changed from int to #define we also changed the punctuation to ALL CAPS. This is the common convention (not required) for constants being declared and defined with the #define directive.

If you make these changes to our previous program, you will see that it actually takes a bit less space.

The second reason NOT to call a constant a variable is what is called “type checking”. If you use:

#define BUZZER 8

And then later on in your program you say:

BUZZER = BUZZER + 1;

You will receive a compiler error since you are trying to change the value of a constant. Strong type checking is very helpful in a language and can save the programmer a lot of headaches by not allowing stupid mistakes like trying to change the value of a constant.

The third item is “scope”; that is, how widely known is the variable? Is it global – know by all functions/procedures in our program or is it known only within the particular function where it is declared and used? In general, we only want variables to be known by the actual function that uses them and we really want to limit “global” variables to the minimum possible.

```
// declare and initialize outside of any function so they are global
#define DO      440
#define RE      494
#define MI      554
#define FA      587
#define SOL     659
#define LA      740
#define TI      831
#define DO2     880

#define BUZZER          8                       // the buzzer is on D8

int toneArray[] = { DO, RE, MI, FA, SOL, LA, TI, DO2};

void setup() {
  pinMode(BUZZER, OUTPUT);
  for(int i = 0 ; i < 8 ; i++) {
        tone(BUZZER, toneArray[i]);             // sent the frequency toneArray[i] to "BUZZER"
        delay(400);
        noTone(BUZZER);
        delay(100);
  }
}

void loop() {

// space to insert statements in the loop() function

}
```

Figure 24 Ch_8_5_basic_musical_scale_array_define

8.6 CREATE A SEPARATE PROGRAM TO PLAY THE MUSICAL SCALE

Now that we have our program successfully playing the musical scale, let's package the part of the program that actually plays the scale into a separate function/procedure/sub-routine. This way, we can use this function in other programs as well.

Let's name our new function playScale() and for now it will return no value (void) and take no parameters. We can copy all the code in our setup() function EXCEPT the pinMode statement into our new function playScale(). The reason we are going to leave the pinMode command is that (a) it really only needs to be performed once, and (b) it is nice to have all the Input/Output (I/O) assignments in one place for easy

reference for future programmers. We also have to "call" the playScale() function within the loop() function, otherwise it will never get run.

So here is what our program could look like:

```
#define DO      440             // declare and initialize outside of any function so they are global
#define RE      494             // later we will reconsider where these should be placed
#define MI      554
#define FA      587
#define SOL     659
#define LA      740
#define TI      831
#define DO2     880
#define BUZZER  8               // the buzzer is on D8

int toneArray[] = { DO, RE, MI, FA, SOL, LA, TI, DO2};   // probably move this to playScale function
                                                         // so other functions can't change the values
// ----------------------------------------------------------------------------
void setup() {
  pinMode(BUZZER, OUTPUT);
}
// ----------------------------------------------------------------------------
void loop() {
        playScale();                                     // call the playScale function to play the scale
}
// ----------------------------------------------------------------------------
void playScale() {
          for(int i = 0 ; i < 8 ; i++) {
                  tone(BUZZER, toneArray[i]);          // sent the frequency toneArray[i] to the pin "buzzer"
                  delay(400);
                  noTone(BUZZER);
                  delay(100);
          }
}
```

Figure 25 Ch_8_6_basic_musical_scale_array_separate_function

There are still several improvements we could make to this program. First, the statement:

```
int toneArray[] = { DO, RE, MI, FA, SOL, LA, TI, DO2};
```

is currently declared and initialized outside of any function so it is a global variable. It is probably not a very good idea to let all the other functions have the opportunity to mess with the notes that are in that array and for that reason it would be best to move that statement INTO the playScale function so that it becomes a private, local variable that no other function can mess with.

We could also consider moving the definition of the notes into the playScale function and perhaps even reconsidering placing the pinMode statement into the playScale function but we will discuss that further in this curriculum.

8.7 PLAY THE SCALE ONCE WHENEVER WE PRESS THE NO MC SWITCH (THE BUTTON)

Now we want to play the scale once every time we press the button. In this case, we don't want to play the scale when we first start our program, we want to wait until the button is pressed and then play the scale once and wait for the button to be pressed again.

Since we already wrote a separate function called playScale() perhaps it would be a good idea to write a new function that will read the switch and return the state of the switch. Let's call our new function switchPushed() and the two states could be true for pushed and false for not pushed. If we define our function in this manner, it might be a useful function that other programs could use if they want to know if the button is pushed or not.

```
boolean switchPushed() {
// returns true if pushed and false if not pushed
// no working code yet, just return a boolean value for now
        return false;      // return false for now
}
```

So far, this is all we know about our new function switchPushed(). Notice several things. There is a new statement "return" that we have not seen before. This is a control structure key word. If you use "return" by itself, it simply returns (passes control) from a function to the previous function that called it. So if in our loop function we called switchPushed, when the switchPushed function comes across the statement "return", it simply returns to the calling loop function.

But there is another way to use the return statement – that is with a constant or variable after the word return. In this case, the value of the constant or variable is returned to the calling program. Since we declared that the switchPushed function returned a boolean value, any boolean constant (true, false) or any variable that was declared as a boolean variable could be returned to loop.

The reason we show the code returning false is we wanted to create what is called a "stub" function. Since we don't know how to write the switchPushed function yet, we just put a comment inside the function reminding us what the function needs to do and returned a valid boolean value so we can test our loop function even before we have written the switchPushed function. We can test our program for switchPushed == false and then change the return value to true and test our program again. Writing stub functions is quite useful so that you can test the main program ASSUMING that the other functions you are going to write return known values and then write the stub programs to do their jobs.

Within our main loop function, we know that we want to call switchPushed and IF the return value == true (e.g. the switch is pushed) then we want to call the playScale function, otherwise we will do nothing. We will keep calling the switchPushed function over and over again, and whenever it returns true, we will call playScale one time.

We could program this as:

```
loop() {
        boolean pushed;                 // declare variable to hold return value of switchStatus()
        pushed = switchPushed();        // get switch status
        if(pushed == true) {    // if switch is pushed, play the musical scale, notice "==" not "="
                playScale();
        }
}
```

We really don't need the new boolean variable "pushed". We can call switchPushed and have the "if" statement evaluate the return value directly as follows:

```
loop() {
        if(switchPushed() == true)      playScale();
}
/*      if we only have a single statement to be executed, we don't need the {} that are used to
        include multiple statements. Having everything on one line makes it easier to read and
        fewer lines of code
*/
```

If you look at the code on the next page (Ch_8_7_musical_scale_if_switch_pushed_stub) you will notice that, in addition to adding the switchStatus function declaration and definition, we have also move the declaration and initialization of the notes into the playScale function, changed them back to integer variables (rather than using #define) and put several other word (static, const) in front of the int declaration.

Now that we are starting to use our playScale function in another program, it would be nice if it was self-contained and did not require that the variables it uses be defined elsewhere. In addition, now would be

a good time to start addressing “scope” and not letting other functions know anything about the variables used by playScale. We can’t just move the #defines into the playScale function because #defines are ALWAYS global no matter where you put them. So we will go back to declaring these as int,

By moving the declaration/definition of variables into the playScale function, these variables are not known to any other function outside of playScale. If we simply move these variables into the playScale function, they will only be known within the playScale function but they will also be “dynamic” – that is, they will be created each time the playScale function is called and will disappear each time you exit the playScale function.

There is a modifier, “static” that can be placed in front of a declaration/definition that tells the compiler to create the variable only once and keep it around as long as the program is running. The scope remains the same – that is, the variable is known ONLY within the playScale function but it acts as if it had been created within the setup() function in that it is performed only once.

So now our declaration/definition of the buzzer pin would be: static int buzzer = 8;

This is nice and addresses the scope issue as well as the speed issue of only having to perform the operation once, but what about the protection of not being able to change the constant that was provided by the #define directive? For this, we use another modifier, “const”, which stands for constant. A variable marked as “const” will give you a compiler error if you try to change it. We can use several modifiers at once so now our declaration/definition of the buzzer pin is: static const int buzzer = 8;

Take a look at Figure 23 Ch_8_7_musical_scale_if_switch_pushed_stub and see how our program has changed. Let’s make a few test to verify that the scope is really local to playScale() and that we have protection against accidently trying to change the value of a constant.

To test scope, add something like the following to the setup() or loop() program:

int testScope = Do;

Try to compile and notice that you get an error that says Do is not defined within this function. Great, that works!

Now, within the playScale() function, try to change the value of the buzzer pin number. For example, add the statement: buzzer = buzzer + 1; and notice that you also get a compiler error saying you are trying to change the value of a constant. Good to know that we have the protection we wanted!

Starting with our IDE (Integrated Development Environment) template from several chapters ago, we have:

```
                                // space for declaring global variables

void setup() {

                                // space to insert statements in the setup() function

}

// ---------------------------------------------------------------------------
void loop() {
        if(switchPushed() == true)          playScale();

}
// ---------------------------------------------------------------------------
void playScale() {
        static const int DO = 440;
        static const int RE = 494;
        static const int MI = 554;
        static const int FA = 587;
        static const int SOL = 659;
        static const int LA = 740;
        static const int TI = 831;
        static const int DO2 = 880;

        static const int BUZZER = 8;                                    // the buzzer is on pin 8

        pinMode(BUZZER, OUTPUT);

        static const int toneArray[] = {DO, RE, MI, FA, SOL, LA, TI, DO2};      // moved this into playScale function
        for(int i = 0 ; i < 8 ; i++) {
                tone(BUZZER, toneArray[i]);                             // send the frequency toneArray[i] to the pin "buzzer"
                delay(400);
                noTone(BUZZER);
                delay(100);
        }
}
// ---------------------------------------------------------------------------
boolean switchPushed() {

// returns true if pushed and false if not pushed

        return false;           // return false for now

}
```

Figure 26 Ch_8_7_musical_scale_if_switch_pushed_stub

Compile and run this program with switchPushed returning false and then change the return value to true and run it again. Does your program work as expected? You should only play the musical scale if return == true and have nothing but silence if return == false.

8.8 WRITING THE SWITCHPUSHED() FUNCTION

Now we are ready to actually write the switchPushed function and return the actual status of the switch rather than always returning either true or false.

Looking at the Appendix, we see that the NOMC switch is connected to Digital pin D2 and since we are going to read the value of the switch (open or closed) we are going to need to configure this pin as an INPUT but if we were to follow the traces on the PCB we would find that one side of the switch is connected to Nano D2 and the other side to ground.

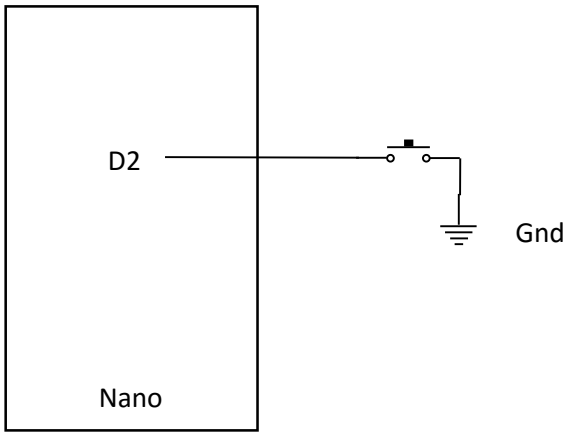
There is something not quite right here. Where would any voltage come from? We have one side of the switch connected to ground (0 volts) but the other side only goes to the Nano pin D2. If we simply read the voltage, it will ALWAYS be zero volts.

This is where we are going to use the INPUT_PULLUP mode of the pinMode function. When we specify:

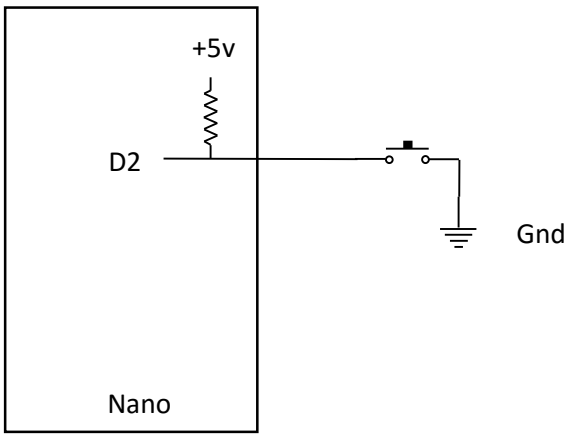
```
pinMode(pin, INPUT_PULLUP);
```

then the pin will be connected to +5 v through a resistor. Now when the switch is open, we will read/detect 5 volts but when the switch is closed we will read 0 volts.

pinMode(2, INPUT);



pinMode(2, INPUT_PULLUP);



Now that we have pin D2 programed to pull the switch up to 5v, all we need to do is perform a standard digitalRead to determine if the switch is open or closed. But wait! The digitalRead will return HIGH if there is 5v and LOW if there is 0v on pin D2 so we are going to need to convert LOW to true (since if the switch is pressed, there will be 0v or LOW).

```
boolean switchPushed() {
        static const int pushButtonSwitch = 2;                  // switch is on pin D2, declare as constant, hide from other functions
         pinMode(pushButtonSwitch, INPUT_PULLUP);// put pin D2 in the INPUT mode and attach a PULLUP resistor
         if(digitalRead(pushButtonSwitch) == HIGH) return false;          // if 5v (HIGH) switch is NOT pushed
         else return true;                                                // if 0v (LOW) switch IS pushed
}
```

Now if you add the code above to replace the existing code within the switchPushed() function and compile and upload it. Now your buzzer should be quiet until you press the push button switch and then it should play the scale once.

```
void setup() {

}

// ----------------------------------------------------------------------
void loop() {
        if(switchPushed() == true)          playScale();

}
// ----------------------------------------------------------------------
void playScale() {
          static const int DO = 440;
          static const int RE = 494;
          static const int MI = 554;
          static const int FA = 587;
          static const int SOL = 659;
          static const int LA = 740;
          static const int TI = 831;
          static const int DO2 = 880;

          static const int BUZZER = 8;                                  // the buzzer is on pin 8

          pinMode(BUZZER, OUTPUT);

          static const int toneArray[] = { DO, RE, MI, FA, SOL, LA, TI, DO2};    // moved this into playScale function
                    for(int i = 0 ; i < 8 ; i++) {
                              tone(BUZZER, toneArray[i]);       // send the frequency toneArray[i] to the pin "buzzer"
                              delay(400);
                              noTone(buzzer);
                              delay(100); }
                              noTone(BUZZER);                                   //turn off buzzer at the end of playing scale
}
// ----------------------------------------------------------------------
boolean switchPushed() {
        static const int pushButtonSwitch = 2;                  // switch is on pin D2, declare as constant, hide from other functions
        pinMode(pushButtonSwitch, INPUT_PULLUP);                // D2 INPUT mode and attach a PULLUP resistor
        if(digitalRead(pushButtonSwitch) == HIGH) return false; // if 5v (HIGH) switch is NOT pushed
        else return true;                                       // if 0v (LOW) switch IS pushed
}
```

Figure 27 Ch_8_8_musical_scale_if_switch_pushed

9 COMMUNICATIONS BETWEEN YOUR COMPUTER AND THE NANO

9.1 HAVING YOUR PROGRAMS WRITE TO YOUR COMPUTER SCREEN

So far, once we have uploaded our programs to the Nano, we have no more contact from our computer. The only thing we can do to see if our program is running the way we want it to look at our sensors and Input/Output (I/O) devices. (e.g. is the buzzer buzzing or the pin 13 LED blinking)

As we write more complex programs, we really need to be able to have our program send back some information to our computer screen so we can see if our program is acting the way we want it to. This information could be a simple "I am here at this location in the program" message or a "I'm about to call this function" message, or even the value of a variable like "I am about to play a tone of 2,000 Hz". These messages will be quite helpful as we debug our code.

Additionally, we may want to control the behavior of our program by using key strokes on our keyboard. Fortunately, the Arduino family comes with support for just such communication between your microcontroller board and your computer.

Go back to the Arduino reference page (http://arduino.cc/en/Reference/HomePage) and look at the far right column, toward the bottom where it says "Communication". Click on "Serial" and you will find a list of the serial communications functions that are available. We are going to focus on the following functions:

```
Serial.begin(speed);      // configures the serial communications channel at the selected speed

Serial.println(value);    // prints the variable or string with a "return" to start a new line

Serial.read();            // reads input from the keyboard
```

In order to access either the Serial.println or Serial.read, we must first set up the communications channel using the Serial.begin function. If you are going to want to have access to these functions from within your program, place the Serial.begin function in your setup() function as follows:

```
Serial.begin(9600);       // set up the serial communications channel at 9600 bits per second
```

Later in your main loop function or any of your own functions, if you want to print the value of a variable named "tone", all you would do is the following:

```
Serial.println(tone);
```

Many times, just printing the value of a variable is not enough. You probably want to print something like:

The value of tone is: 2000

In order to do that, you need to use two statements: Serial.print() and Serial.println(). The Serial.print() does NOT generate a new line while the Serial.println() does.

```
Serial.print("The value of tone is: ");
Serial.println(tone);
```

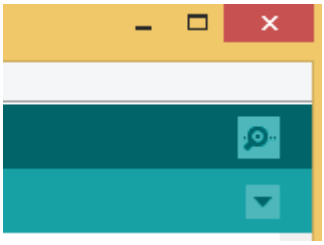
No go back to your program "Ch_8_8_musical_scale_if_switch_pushed" and add the Serial.begin(9600) to the setup function and add a Serial.print and Serial.println to the playScale function just before the delay

```
tone(buzzer, toneArray[i]);          // send the frequency toneArray[i] to the pin "buzzer"

Serial.print("playing the following tone: ");
Serial.println(toneArray[i]);

 delay(1000);                        // play tone for one second
```

Now compile and upload that modified program and press the button to hear the musical scale being played. In order to see the messages on your computer screen, you need to open the Serial Monitor on your desktop. To do that, click on the Serial Monitor button on the top right of your IDE screen – looks like a magnifying glass.



This will open up a new window and you should see a line for each tone saying: playing the following tone: 440 (440 will be replaced with each separate tone as they are being played).

Now that we can get OUTPUT to our screen, let's turn our attention to having our program receive INPUT from our keyboard.

9.2 HAVING YOUR PROGRAMS RECEIVE INPUT FROM YOUR KEYBOARD

Now we want to be able to send input to our program while it is running on the Nano using our keyboard on our computer. For this we will use two new functions: Serial.available() and Serial.read().

Serial.available() will return the number of characters that are available to read. So if we were to call this function and the return value is > zero, then we know we have at least one character to read. We will need a char variable to hold the value that we read so we will need to declare one. For this example, we are going to look for either a capital or lower case P (for Play) and if we receive either of these characters, we are going to call the playScale() function.

You are going to need a few other pieces of information before we proceed. First, in C/C++ there is a huge difference between a single character and a string of characters. A single character can be represented in a single 8 bit byte and would be represented by single quotes around the single character. For example, 'P' is the single character capital P and 'p' is the single character lower case p. If we want to represent a string of characters, for example "Play scale command received from keyboard" we would enclose the string in double quotes "character string".

One other item that will make our lives easier is the logical OR test. If we want capital OR lower case 'p' we can use the OR operator || (two pipe symbols, NOT !). For example:

```
if(inputChar == 'P' || inputChar == 'p') {

// do the following

}
```

Would be true for either capital P OR lower case p.

Program Requirements:

Declare a char variable called inputChar to hold the character from the keyboard

Call the playScale() function IF you receive either a capital or lower case p

Make sure to initialize the serial channel to 9600 bits per second

Only try to read a character from the keyboard IF there is one AVAILABLE

When you receive either a 'P' || 'p', print the following character string "Play scale command received from keyboard"

Make these changes to the program: Ch_8_6_Musical_scale_if_switch_pushed

Try to work this on your own before you go to the next page. When you get the program to compile, upload it and see if it works. Remember to open the Serial Monitor window so you can see what gets printed. You have to enter either the P or the p and then hit "enter" in order to send the character from your keyboard to the Nano.

```
                                // space for declaring global variables

void setup() {

        Serial.begin(9600);}

// ------------------------------------------------------------------------
void loop() {
        if(switchPushed() == true)          playScale();

        char  inputChar;
          if(Serial.available() > 0) {
                inputChar = Serial.read();
                if(inputChar == 'p' || inputChar =='P') {
                        Serial.println("Play scale command received from keyboard");
                        playScale();
                }
        }
}
// ------------------------------------------------------------------------
void playScale() {
        static const int DO = 440;
        static const int RE = 494;
        static const int MI = 554;
        static const int FA = 587;
        static const int SOL = 659;
        static const int LA = 740;
        static const int TI = 831;
        static const int DO2 = 880;

        static const int BUZZER = 8;                                    // the buzzer is on D8
        static const int toneArray[] = { DO, RE, MI, FA, SOL, LA, TI, DO2};    // moved this into playScale function

        for(int i = 0 ; i < 8 ; i++) {
                tone(BUZZER, toneArray[i]);       // send the frequency toneArray[i] to the pin "buzzer"
                delay(1000);                              // play tone for one second
        }
        noTone(BUZZER);                           //turn off buzzer at the end of playing scale
}// ------------------------------------------------------------------------
boolean switchPushed() {

        const int pushButtonSwitch = 2;  // switch is on pin D2, declare as constant, hide from other functions

      pinMode(pushButtonSwitch, INPUT_PULLUP);          // D2 INPUT mode and attach a PULLUP resistor

      if(digitalRead(pushButtonSwitch) == HIGH) return false;          // if 5v (HIGH) switch is NOT pushed

      else return true;                                                // if 0v (LOW) switch IS pushed

}
```

Figure 28 Ch_9_2_read_keyboard_play_scale

10 PROGRAMMING THE SIX COLORED LIGHT EMITTING DIODES (LED'S)

A diode is a two terminal device that allows current to flow in only one direction. A LED will emit a specific color when the current is flowing and will be dark when no current is flowing. The symbols for a diode and a LED are shown below:

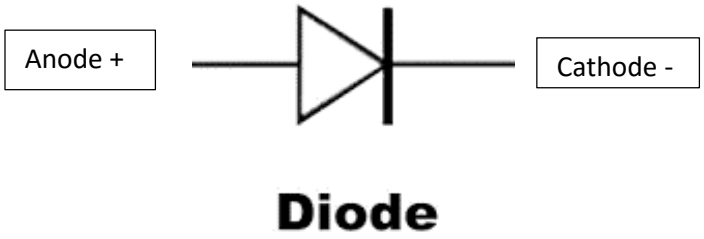


Figure 29 Diode Schematic Symbol

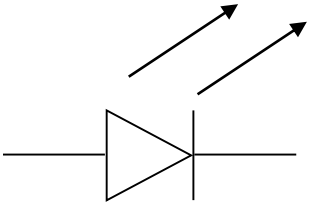


Figure30 Light Emitting Diode (LED) Symbol

Current is the flow of electrons and since electrons are negatively charged, they are repelled by the negative terminal of the battery and attracted to the positive terminal. So electrons move from the Cathode to the Anode. The absence of an electron, called a "hole", moves in the opposite direction of the electrons – from positive to negative. The arrow on the diode symbol points in the direction of the "hole" movement.

To the right is a photo of the six different colored LED's included in your Geek Pack™. Notice that these LED's all have two leads and that one lead is longer than the other. The longer lead is the Anode and would be connected to the POSITIVE terminal.

There are two other LED's included in your Geek Pack™ that have THREE leads – these devices actually have TWO separate LED's in one package with the two Anode's tied together to create a three lead device. We will be using these special LED's later so set them aside for now.

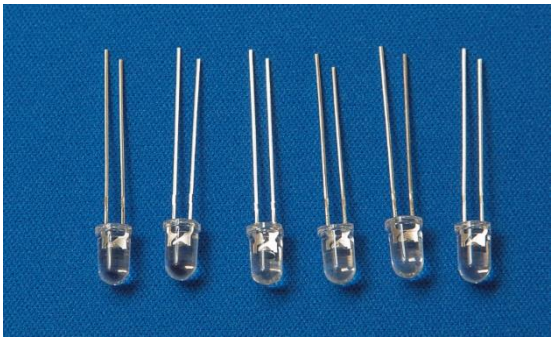


Figure 31 Six different colored LED's - long lead is +

If we were to connect an LED directly to our 9v battery or even directly to our microcontroller (Ardunino Nano), we would either burn out our LED or destroy the microcontroller – or both. This is because too much current would flow and the devices would overheat and burn out. We need to limit the amount of current that flows through the LED by using a RESISTOR. A resistor restricts, or resists the flow of current and we can calculate how much current will flow using the following equation:

Voltage = Current * Resistance (voltage in Volts, Current in Amps, Resistance in Ohms)

This is often written as either V = IR or E = IR (E stands for Electromagnetic force or Voltage). The Arduino Nano specifications say that each 5v output pin can supply 40 mA (one milliamp = 1/1000 Amp) so we know we must limit the current to less than 40 mA. The specifications for the LED's indicate 5 – 20 mA and in order to maximize the battery life, let's target something close to 6 mA.

Before we can calculate the value of the resistor, we need one more item; the amount of voltage across the LED. Different colored LEDs operate at different voltages but 2.0 volts is a good average value for the different colored LEDs in this Geek Pack™.

Referring to the figure below, if we start with 5 volts and there are 2 volts across the LED, then the resistor will have 3 volts across it. What value of resistance would cause 3 volts to be across the resistor with 6 mA flowing?

R = E/I = 3volts/0.006A = 500 Ohms

Resistors come in fixed values and 470 ohms is a common value so we will use this value resistor for our LED's.

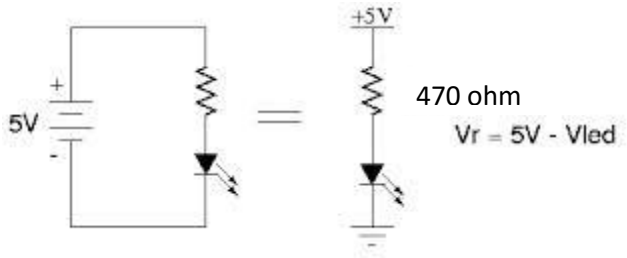
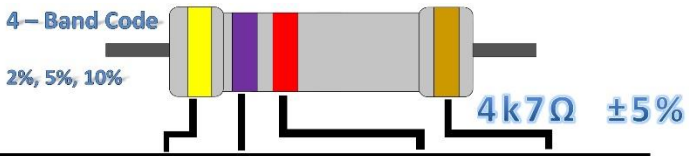


Figure 32 LED Circuit Schematic

Resistors are color coded to indicate their value. The resistors included in this Geek Pack™ use the 4-band code as shown below. With the three bands that are close together pointed to the left, the first to bands are the digits (Yellow, Violet = 47. The next band indicates how many "zeros" are after the two digits. Brown indicates one zero so Yellow, Violet, Brown = 470 ohms. At the other end of the resistor is the TOLLERANCE band. Gold equals 5% tolerance so the resistors included in your kit are 470 ohms, 5% tolerance.



Color	1st Band	2nd Band	3rd Band	Multiplier	Tolerance
Black	0	0	0	1Ω	
Brown	1	1	1	10Ω	± 1%
Red	2	2	2	100Ω	± 2%
Orange	3	3	3	1kΩ	
Yellow	4	4	4	10kΩ	
Green	5	5	5	100kΩ	± 0.5%
Blue	6	6	6	1MΩ	± 0.25%
Violet	7	7	7	10 MΩ	± 0.1%
Grey	8	8	8		± 0.05%
White	9	9	9		
Gold				0.1Ω	± 5%
Silver				0.01Ω	± 10%

Figure 33 Resistor Color Code

10.1 STARTING TO PROGRAM THE SIX COLORED LED'S

From the Appendix we can see that the six colored LEDs are connected to the following pins on the Nano:

Red	D11
Orange	D10
Yellow	D9
White	D6
Green	D5
Blue	D3

These six LEDs are wired so that the Cathode (negative end) of each LED is connected to Ground and the Anode is connected to the Nano pin through a 470 Ohm resistor. This means that in order to light the LED, we need to send +5v (a HIGH) to the digital output pin.

Here are the requirements for this program. Try writing the program on your own before you continue reading the step by step directions.

Program Requirements:

- Program name Ch_10_1_6leds
- LED pin numbers as shown above
- Sequence through each LED in the following order (Red, Orange, Yellow, White, Green, Blue)
- For each LED, turn LED on for time ledOn and then off for time ledOff

By now you should have a pretty good idea of the items we need to accomplish outside of any function (global) and in the setup() function. We need to set the pinMode() of each of the LED pins to OUTPUT and we should probably do a digitalWrite(LOW) to make sure all the LED's are off.

We could do this by doing the following for each LED:

```
#define RED     11              // outside of any function to make global

pinMode(RED, OUTPUT);           // in setup()

digitalWrite(RED, LOW);         // in setup()
```

That would certainly get the job done but would require a lot of typing.

Let's think about what we might want to do with the LEDs before we determine how to perform the setup() activities. Let's start by lighting each LED, in sequence, for a period of time ledOn and then off for a period of time ledOff. To begin, let's do them in the order of: Red, Orange, Yellow, White, Green, and Blue. Later we know we may want to change the order so that should factor into our coding decision as well.

If we were to use an Array[] that contained the LED pin numbers in the sequence that we wanted to light them, then if we wanted to change the sequence, all we would need to do is change the order of the LED pin numbers in the array. This sounds like a much better plan than “hard coding” the sequence as in the above example.

What if we did the following:

```
int red = 3;
int orange = 5;
int yellow = 6;
int white = 9;
int green = 10;
int blue = 11;

int ledArray[] = {red, orange, yellow, white, green, blue};

void setup() {
        // make LED pins OUTPUTs and turn off all LEDs
        for(int i = 0 ; i < 6 ; i++) {
                pinMode(ledArray[i], OUTPUT);
                digitalWrite(ledArray[i], LOW);
         }
}
```

Then, if we want to change the order of the LEDs being lit, we simply change the order of the colors in the array ledArray[].

This would take care of the setup() items for the LED’s but if we want to turn them ON for time = ledOn and OFF for time = ledOff and sequence through the list of LEDs in ledArray[], we would need to add something like the following. We have many choices of where to add these statements. We could place them outside the setup() or loop() functions if we want to make them global or we could put them inside the loop() function if we were going to use them within the loop function or with a separate blinkLeds() function if that is where we are going to use them to blink the leds. For now, let’s make them global and place them outside of any function definition.

```
int     ledOn = 250;

int     ledOff = 100;
```

We would also have to add something like the following to the loop() function in order to turn each LED on for time ledOn and off for ledOf time.

```
loop() {
        for(int i = 0 ; i < 5 ; i++) {
                digitalWrite(ledArray[i], HIGH);
                delay(ledOn);
                digitalWrite(ledArray[i], LOW);
                delay(ledOff);
        }
}
```

Putting all of this together, we would have the following:

```
int     ledOn = 250;
int     ledOff = 100;

int red = 3;
int orange = 5;
int yellow = 6;
int white = 9;
int green = 10;
int blue = 11;

int ledArray[] = {red, orange, yellow, white, green, blue};

void setup() {
        // make LED pins OUTPUTs and turn off LEDs
        for(int i = 0 ; i < 6 ; i++) {
                pinMode(ledArray[i], OUTPUT);
                digitalWrite(ledArray[i], LOW);
        }
}

void loop() {
        // turn on each LED, one at a time for time ledOn
        // and then off for time ledOff
        for(int i = 0 ; i < 6 ; i++) {
                digitalWrite(ledArray[i], HIGH);
                delay(ledOn);
                digitalWrite(ledArray[i], LOW);
                delay(ledOff);
        }
}
```

Figure 34 Ch_10_1_6leds

10.2 Two Dimensional Array and Nested "for" loops

What if we want to create a more complex sequence of patterns for the LEDs? How could we do that? We are going to use this program to introduce two new concepts: nested "for" loops (having one "for" loop inside another "for" loop) and two dimensional arrays.

A two dimensional array looks like a table with rows and columns. In our case, each column is an LED so we will have 6 columns and each row is a different pattern of on/off for the LEDs. If an element of the array is a "1", we will light the LED and if the element is a "0" we will turn it off.

Starting with the first row, we will sequence through all the columns, turning the corresponding LED on or off based upon whether that element is a 1 or a 0. Then we will wait a time = ledOn, point to the next row, and start again on this new row. In this way, each row becomes an on/off pattern for the 6 LEDs that is illuminated for a time = ledOn, and then the pattern in the next row is illuminated. The result of lighting each LED according to the value 1 or 0 in a row, waiting, and then performing the same action for the next row will create a sequence of patterns for the LED's.

When you declare a two dimensional array without initializing it, you need to specify both the number of rows and the number of columns as follows:

```
#define numRows 6
#define numCols 6

int       sequenceArray[numRows] [numCols];
```

This would declare a two dimensional array with 6 rows and 6 columns. Normally, you will want to initialize the array at the same time as follows:

```
int sequenceArray [] [numCols] {
  {1, 0, 0, 0, 0, 1},
  {0, 1, 0, 0, 1, 0},
  {0, 0, 1, 1, 0, 0},
  {1, 0, 0, 0, 0, 1},
  {0, 1, 0, 0, 1, 0},
  {0, 0, 1, 1, 0, 0}
};
```

Notice that in this case, we did not provide a value for the number of rows – the compiler will calculate this based upon the data that we provided to initialize the array. In general, for an "n" dimension array, you must specify all but the last dimension – that will be calculated.

When you reference a two dimensional array, you must specify both the row and the column you are interested in. For example, sequenceArray[1][4] would return '1' since the second row (starting from 0, remember), the fifth element (again, starting from 0) is a '1' (colored orange above).

One more item to point out before we jump into coding. Notice that if we use a loop counter "i" to point to the column in the two dimensional array, then the value of the "i"th element on the current row tells us whether to turn on or off the LED associated with that column – but what LED is that?

From our last program, we used the one dimensional array ledArray[] to point to the pin numbers of the LEDs in the sequence they are installed in our box. So, the loop counter "i" that is used to identify the column in the two dimensional array will also point to the correct pin number in ledArray[i].

An example might make this clearer.

```
int red = 3;
int orange = 5;
int yellow = 6;
int white = 9;
int green = 10;
int blue = 11;

#define numCols 6
#define numRows 6

// this is the sequence for the 6 LEDs
int ledArray[] = {red, orange, yellow, white, green, blue};

int sequenceArray [] [numCols] {
  {1, 0, 0, 0, 0, 1},
  {0, 1, 0, 0, 1, 0},
  {0, 0, 1, 1, 0, 0},
  {1, 0, 0, 0, 0, 1},
  {0, 1, 0, 0, 1, 0},
  {0, 0, 1, 1, 0, 0}
};
```

If i = 3, then sequenceArray[2][i] would return "1" (see red "1" above) and using the same loop counter "i", ledArray[i] would return the pin associated with the "white" LED or the pin number 9. Putting this all together, the "1" in sequenceArray[2][3] says turn on that LED and the 9 returned from ledArray[3] says that LED is on digital pin 9.

Program Requirements:

- Name your program Ch_10_2_two_dim_array_nested_for_loops
- Create a two dimensional array containing integers named sequenceArray[] []
- sequenceArray has numCols = 6 columns (one for each LED) and numRows = your choice
- Continue to use ledArray[] to contain a list of the colors/pin numbers for the 6 LED's
- Use a nested "for" loop with two loops: the outer loop selects the row and the inner loop selects the columns
- If the entry in sequenceArray[j] [i] is a 1 then turn on the led ledArray[i], otherwise turn it off
- Continue to display each row for time ledOn = 300ms
- Continue to sequence through all the rows and then repeat forever

Take a shot at programming this before you proceed to the next page.

The nested “for” loops will contain two loops: an outer loop that sequences through the rows (with each row being a different pattern that will be displayed) and an inner loop that sequences through each of the six columns (one for each of the 6 LED’s).

```
for(int j = 0 ; j < numRows ; j++) {                // outer loop increments through the ROWS

        for(int i = 0 ; i < numCols, i++)           // inner loop increments through the COLUMNS
                if(sequenceArray [j][i] == 0)       digitalWrite(ledArray[i], LOW);
                else                                digitalWrite(ledArray[i], HIGH);
        }
        delay(ledOn);
 }
```

So the program would look as follows:

```
/* Ch_10_2_two_dim_array_nested_for

  Using a two dimensional array and nested "for" loops to control the 6 LED light pattern

   copy and paste this program into the Arduino IDE
   name this program Ch_10_2_two_dim_array_nested_for
*/

//  ---------------------------------------------------------------------------------
Int ledOn = 300;
int red = 3;
int orange = 5;
int yellow = 6;
int white = 9;
int green = 10;
int blue = 11;

// this is the sequence for the 6 LEDs
int ledArray[] = {red, orange, yellow, white, green, blue};

/*  the 6 x 6 array represents one entire display pattern sequence. The columns represent the 6 LEDs (0 - 5)
    and the rows represent the the pattern that the 6 LEDs will display. A zero means the LED is off and a
    one means ON.
    So, the first row in the 6x6 array says LED 0 and 5 will be on, all other off, the second row indicates
    that LEd 1 and 4 will be on and so forth.
 */
#define numCols 6
#define numRows  6

int sequenceArray [] [numCols] {
  {1, 0, 0, 0, 0, 1},
  {0, 1, 0, 0, 1, 0},
  {0, 0, 1, 1, 0, 0},
  {1, 0, 0, 0, 0, 1},
  {0, 1, 0, 0, 1, 0},
  {0, 0, 1, 1, 0, 0}
};
```

Figure 35 Figure 14 Ch_10_2_two_dim_array_nested_for - part 1

```
//  ---------------------------------------------------------------------------
void setup() {
// configure the LED pins as outputs and turn off the LEDs
  for(int i = 0 ; i < numCols ; i++) {
              pinMode(ledArray[i], OUTPUT);
              digitalWrite(ledArray[i], LOW);
  }
}

//  ---------------------------------------------------------------------------
void loop() {
/* each row represents one LED pattern, display that pattern for time ledOn
   then go to next row and display that pattern
 */
  for(int j = 0; j < numRows ; j++) {                          // outter loop - ROW pointer
    for(int i = 0 ; i < numCols ; i++) {                       // inner loop - COLUMN pointer
      if(sequenceArray [j][i] == 0)          digitalWrite(ledArray[i], LOW);
      else                                   digitalWrite(ledArray[i], HIGH);
    }
    delay(ledOn);
  }
}
```

Figure 36 Figure 14 Ch_10_2_two_dim_array_nested_for - part 2

The LED pattern sequence that we programmed starts with the top and bottom LED illuminated, then the next two LEDs closer to the center are illuminated, and finally the center two LEDs are illuminated. Try changing the values in the sequenceArray[] to create your own pattern sequence before you proceed to the next section.

10.3 Three Dimensional Array and Button Push

Now that you have played around with several different pattern sequences, what if you have several favorites and you would like to select which one to display without having to go back to your computer and change the values in the sequenceArray[]? How can we program it such that each time you press the NOMC (Normally Open, Momentary Contact) switch, you select your next pattern sequence?

For this, we are going to use a three dimensional array where each plane (each 2 dimensional array) represents one entire LED pattern sequence. Think of the three dimensional array as a cube. If you look at a face of the cube, you are looking at a two dimensional array where the columns represent the individual LEDs and the rows are the various illumination patterns that go with this pattern sequence – just like in the last example. If you go back into the cube one layer, you are at another two dimensional array that represents another pattern.

The syntax for declaring and initializing a three dimensional array is very similar to that of a two dimensional array.

```
Int nameOfThreeDimArray [],[numberOfColumns] [numberOfRows] {

        {                                                       // start of first sequence
                {1, 0, 0, 0, 0, 1},     // first pattern to display
                {0, 1, 0, 0, 1, 0},     // second pattern to display
                {0, 0, 1, 1, 0, 0},     // third pattern to display
                {1, 0, 0, 0, 0, 1},     // fourth pattern to display
                {0, 1, 0, 0, 1, 0},     // fifth pattern to display
                {0, 0, 1, 1, 0, 0}      // sixth pattern to display
        },                                                      // end of first sequence, notice the comma

        {                                                       // start of second sequence
                {1, 0, 0, 0, 0, 0},
                {0, 1, 0, 0, 0, 0},
                {0, 0, 1, 0, 0, 0},
                {0, 0, 0, 1, 0, 0},
                {0, 0, 0, 0, 1, 0},
                {0, 0, 0, 0, 0, 1}
        },                                                      // end of second sequence

        {                                                       // start of third sequence
                {1, 0, 0, 0, 0, 0},
                {1, 1, 0, 0, 0, 0},
                {0, 1, 1, 0, 0, 0},
                {0, 0, 1, 1, 0, 0},
                {0, 0, 0, 1, 1, 0},
                {0, 0, 0, 0, 1, 1}
        }                                                       // end of third sequence

};      // semicolon at end of statement
```

Here are the requirements for the program we are going to write using a three dimensional array. Spend some time creating a program that will meet these requirements before you continue and see how we did it.

Program Requirements:

- Name your program Ch_10_3_three_dim_array
- Create a three dimensional array sequenceArray [numPatterns] [numRows] [numCols]
- numPatterns is an int and has a value of 3 (three patterns) a counter would go 0, 1, 2
- The pointer to the current pattern being displayed is an int called ledPatternNo
- Reuse the function we created called switchPushed() to determine if you should go to the next pattern
- Sequence through the patterns each time the switch is pushed and when you get to the last pattern, start with the first all over again
- Do this continuously as long as power is applied

```
/* Ch_10_3_three_dim_array

  Using a three dimensional array to control the 6 LED light pattern
  depressing the push button switch selects the next pattern

   copy and paste this program into the Arduino IDE
   name this program Ch_10_3_three_dim_array
 */

//  ---------------------------------------------------------------------------
#define numCols 6          // one column for each LED
#define numRows 6           // number of patterns

int    ledPatternNo = 0;   // current pattern to display
int    numPatterns = 3;    // number of pattern sequences

int            ledOn = 300;       // lenght of time to display each pattern

// pin numbers for each LED
int red = 3;
int orange = 5;
int yellow = 6;
int white = 9;
int green = 10;
int blue = 11;

// this is the sequence for the 6 LEDs
int ledArray[] = {red, orange, yellow, white, green, blue};

/*  each 6 x 6 array represents one entire display pattern sequence. The columns represent the 6 LEDs (0 - 5) and the rows
    represent the the pattern that the 6 LEDs will display. A zero means the LED is off and a one means ON.
    So, the first row in the first 6x6 array says LED 0 and 5 will be on, all other off, the second row indicates
    that LEd 1 and 4 will be on and so forth.
 */
int sequenceArray [] [numRows] [numCols] {
 {                        // light LEDs from outside -> center
  {1, 0, 0, 0, 0, 1},
  {0, 1, 0, 0, 1, 0},
  {0, 0, 1, 1, 0, 0},
  {1, 0, 0, 0, 0, 1},
  {0, 1, 0, 0, 1, 0},
  {0, 0, 1, 1, 0, 0}
 },
 {                        // light LEDs from top to bottom
  {1, 0, 0, 0, 0, 0},
  {0, 1, 0, 0, 0, 0},
  {0, 0, 1, 0, 0, 0},
  {0, 0, 0, 1, 0, 0},
  {0, 0, 0, 0, 1, 0},
  {0, 0, 0, 0, 0, 1}
 },
 {                        // light two LEDs and have them travel
  {1, 0, 0, 0, 0, 0},
  {1, 1, 0, 0, 0, 0},
  {0, 1, 1, 0, 0, 0},
  {0, 0, 1, 1, 0, 0},
  {0, 0, 0, 1, 1, 0},
  {0, 0, 0, 0, 1, 1}
 }
};
```

Figure 37 Three dimensional array - part 1

```
void setup() {
// configure the LED pins as outputs and turn off the LEDs
  for(int i = 0 ; i < 6 ; i++) {
          pinMode(ledArray[i], OUTPUT);
          digitalWrite(ledArray[i], LOW);
  }
}
//  ---------------------------------------------------------------------------
void loop() {
  for(int j = 0; j < numRows ; j++) {

    for(int i = 0 ; i < numCols ; i++) {
      if(sequenceArray [ledPatternNo][j][i] == 0)    digitalWrite(ledArray[i], LOW);
      else                                           digitalWrite(ledArray[i], HIGH);
    }
    delay(ledOn);

    if(switchPushed()== true){          // select next pattern if switch is pressed
      ledPatternNo++;
      if(ledPatternNo >= numPatterns) ledPatternNo = 0;
    }
  }
}
// ---------------------------------------------------------------------------
boolean switchPushed() {
  const int pushButtonSwitch = 2;                    // switch is on pin D2, declare as constant,
  pinMode(pushButtonSwitch, INPUT_PULLUP);           // D2 INPUT mode and attach a PULLUP resistor
  if(digitalRead(pushButtonSwitch) == HIGH)  return false;        // if 5v (HIGH) switch is NOT pushed
  else                                       return true;         // 0v if IS pushed
}
```

Figure 38 Ch_10_3_Three_Dim_Array_Button_Push - part 2

Notice that we are reusing our switchStatus() function and using it to move to the next plane in the 3D cube – e.g. selecting the next pattern to run. In this program, ledPatternNo points to the plane we are going to use (the pattern we want to use). Each time we press the button, we increment ledPatternNo and point to the next plane, when we get to the last plane, we reset ledPatternNo to zero and start again.

11 USING THE ULTRA-SONIC RANGE DETECTOR

Now we are going to turn our attention to using the ultra-sonic range detector. This is a really cool device and operates like sonar. The range detector sends out a series of pulses and listens for an echo (a return) where the pulses strike an object and returns. The length of time it takes for the pulse to return is used to determine the distance to the object.

There are four pins on the range detector: power, ground, trigger, echo. Operation is quite simple, have the trigger pin (D4) go from LOW to HIGH and stay HIGH for at least 10 uSec then return to LOW. Then monitor the echo pin (D7) and measure the length of the return pulse in uSec.

The Arduino comes with a command to measure the length of a pulse in uSec (micro seconds, millionth of a second): pulseIn(pinNumber,triggerHighorLow);

The pulseIn() function is general purpose and can be used to measure a "return" signal that goes from LOW to HIGH and then back to LOW (measuring the time in the HIGH state) or the other way around. If triggerHighorLow = HIGH, then we are measuring the time the "return" pulse is in the HIGH state – which is what we want to do.

So we would call pulseIn(7, HIGH); to read a HIGH pulse on digital pin D7.

> Remember to keep referring to: http://www.arduino.cc/en/Reference/HomePage
>
> This page contains the Language Reference for the Arduino and lists all the functions and how to use them. This is a VERY useful page to bookmark.

The speed of sound in air depends upon the humidity and temperature but for our needs, we will assume it is constant at 1,127 feet/second or 0.013524 inches/uSec. When using pulseIn(), remember that the pulse traveled out AND back so the time returned is twice the time we need to determine distance so we will need to divide the return time by 2 and multiply by 0.013524. For example:

```
#define TRIG_PIN                4       // trigger pin on range detector is digital pin D4
#define ECHO_PIN                7       // echo pin is D7
#define TRIG_PULSE_WIDTH        15      // use 15 uSec for trigger pulse width

int     distance;
pinMode(TRIG_PIN, OUTPUT);              // define trigger pin as an output pin
pinMode(ECHO_PIN, INPUT);               // define echo pin as an input pin
digitalWrite(TRIG_PIN, LOW);            // set trigger pin to LOW

digitalWrite(TRIG_PIN, HIGH);           // send 15 microsecond pulse
delayMicroseconds(TRIG_PULSE_WIDTH);    // delay 15 microseconds
digitalWrite(TRIG_PIN, LOW);

distance = pulseIn(ECHO_PIN, HIGH) *2/.0135135          // wait for return pulse to go high
```

11.1 DETERMINE DISTANCE AND PRINT TO SCREEN

In this program, you are going to read the distance from the ultra-sonic range detector SAMPLES times and take the average distance and print that value in inches to the computer screen. You will also use that value to send a tone to the buzzer. The smaller the distance, the higher the tone. The highest tone will be MAX_TONE and the lowest tone will be MIN_TONE. We will arbitrarily limit the range of the distance measurement from 0 – 100 inches.

In order to do this last part, converting distance to frequency, we will need to convert a distance value from 0 – 100 into a frequency from MAX_TONE to MIN_TONE. Fortunately, the Arduino C++ environment contains a function to do just that. It is called map() and you use it as follows:

```
map(valueToMap, fromLow, fromHigh, toLow, toHigh);
```

In our case, we are going to be using

```
map(distanceInches, 0, 100, MAX_FREQ, MIN_FREQ);
```

to take the distance we measured from the ultra-sonic range detector (distanceInches) and map it from its range of 0 – 100 inches to the frequencies MAX_FREQ to MIN_FREQ. Notice that in this case, since the highest frequency is associated with the smallest distance the value we used for “toLow” is actually the highest frequency (MAX_FREQ) and the value we used for “toHigh” is actually the lowest frequency.

In practice, if we have already calculated the value of distanceInches, we could do something like the following:

```
int buzzerFrequency;

buzzerFrequency = map(distanceInches, 0, 100, MAX_FREQ, MIN_FREQ);

tone(BUZZER_PIN, buzzerFrequency);
```

A more efficient (and probably easier to read) version would be:

```
tone(BUZZER_PIN, map(distanceInches, 0, 100, MAX_FREQ, MIN_FREQ));
```

Which does not require the creation of the buzzerFrequency variable.

Program Requirements:

- Name your program Ch_11_1_range_screen
- Use the variable names and values as shown above
- 15 uSec trigger pulse, 3 samples and obtain the average, limit distance from 0 – 100 inches
- At 0 inches, play a tone of 4 KHz and at 100” play 60 Hz
- Print the distance in inches as well as the frequency on the computer screen

Take a shot at programming this before you continue.

Hint 1: Remember to initialize the serial channel with Serial.begin(9600) and to include the #defines before the setup() routine.

Hint 2: Remember to set any output pins to the output mode using pinMode() and setting their initial values to either LOW or HIGH using digitalWrite(). For example, the trigger pin should be set low to start so you can generate a positive pulse.

Here is an example of the definition and initialization sections of our program based upon the requirements given to us. There is no loop() program written yet – that is next on our "to-do" list.

```
/* Ch_11_1_range_screen

  copy and paste this program into the Arduino IDE
  name this program Ch_11_1_range_screen

  uses the ultra-sonic range detector to measure the distance in inches
  to an object and print distance on screen

  also plays a tone associated with the distance from MIN_FREQ (farthest away) to MAX_FREQ (closest)
 */
// ----------------------------------------------------------------------------

#define TRIG_PIN            4                   // trigger pin on range detector is digital pin D4
#define ECHO_PIN            7                   // echo pin is D7
#define BUZZER_PIN          8                   // buzzer is on pin D8

#define TRIG_PULSE_WIDTH          15            // use 15 uSec for trigger pulse width
#define SAMPLES             3                   // number of distance samples to average
#define  MIN_FREQ             60                // 60 Hz minimum
#define MAX_FREQ            4000                // 4 KHz maximum

int distanceInches = 0;                         // used to hold the returned value of distance in inches


// ----------------------------------------------------------------------------
void setup() {
  pinMode(TRIG_PIN, OUTPUT);                    // set up trigger pin as an output pin
  pinMode(ECHO_PIN, INPUT);                     // set up echo pin as an input

  digitalWrite(TRIG_PIN, LOW);                  // make sure trigger pin is low to start

  Serial.begin(9600);                           // setup serial I/O to display
}
// ----------------------------------------------------------------------------
void loop() {

}
```

Figure 39 Ch_11_1_range_screen, part 1

Within the loop program we are going to have to do the following:

- Send a trigger pulse of length TRIG_PULSE_WIDTH
- Listen for and measure the echo using pulseIn(ECHO_PIN, HIGH)
- Take the value we receive from pulseIn() which is time in uSeconds and convert to distance = pulseIn(ECHO_PIN, HIGH) * .0135135/2
- Test to be sure that the distance is between 0 and 100 inches
- Take number = SAMPLES measurements and take the average (add them together and divide by SAMPLES) to get distanceInches
- Map this distanceInches to frequency with 0 inches = MAX_FREQ and 100 inches = MIN_FREQ using the map() function.
- Print distanceInches to the computer screen using Serial.println() function
- Play a tone() on the BUZZER_PIN using the frequency we got back from map()

Spend some time writing the body of loop() before you move on and see how we did it. By now you should be able to make great progress on this on your own.

```
void loop() {

  int duration, freq;                       // duration is uSec echo duration, freq is frequency to send to BUZZER_PIN
  int distance = 0;                         // final averaged distance over SAMPLES measurements
  int accumDistance = 0;                    // the total of SAMPLES distance measurements, reset accumulated distance to zero

  for(int i = 0 ; i < SAMPLES ; i++) {      // measure the distance SAMPLES times and use average distance
   // send trigger pulse
    digitalWrite(TRIG_PIN, HIGH);
    delayMicroseconds(TRIG_PULSE_WIDTH); // wait TRIG_PULSE_WIDTH micro seconds
    digitalWrite(TRIG_PIN, LOW);

    // wait for echo and measure it
    duration = pulseIn(ECHO_PIN, HIGH);
    distance =(duration/2)*.0135135;        // get current distance measurement

    if(distance > 100)  distance = 100;     // largest distance allowed is 100 inches (completely arbitrary)
    if(distance <= 0)   distance = 0;       // just to be sure you dont get a negative number

    accumDistance = accumDistance + distance;       // add current distance to accumulated distance
    delay(100);                                     // wait 1/10 second before taking another distance measurement
   }
   distance = accumDistance/SAMPLES;        // calculate the average distance

  freq = map(distance, 0, 100, MAX_FREQ, MIN_FREQ);  // map the measured distance from 0 - 100 inches

  Serial.print("distance = ");
  Serial.print(distance);

  Serial.print("       frequency = ");
  Serial.println(freq);                     // to a number between MAX_FREQ and MIN_FREQ

  tone(BUZZER_PIN, freq);
}
```

Figure 40 Ch_11_1_range_screen, part 2

11.2 USING DISTANCE TO CREATE A BAR GRAPH WITH OUR SIX COLORED LED'S

In this program we will build on our previous range-detector program by lighting the row of 6 LED's based upon the distance from an object; an object nearby will light all the LED's while an object 100" away will only light one LED.

Since we will now have the LED's to tell us how far away the object is, let's also add the capability to turn off the buzzer so we don't go crazy. Each time the button is depressed, we will toggle between playing and buzzer and silencing the buzzer.

Since we are going to be turning stuff ON and OFF, how about we define the following to make our code easier to read:

```
#define ON true
#define OFF false
```

We are also going to need a global variable to keep track of whether we should be buzzing the buzzer or not. Let's call this variable playBuzzerFag and declare it as follows:

```
int playBuzzerFlag = OFF;
```

One last item we need to discuss is how to light from one to all six LED's based upon the distance we read from the ultra-sonic range detector. Since we have already learned about one, two, and three dimensional arrays for controlling the LED's, let's build upon that.

In our case, a two dimensional array where each row contains the pattern of LED's to light would work just fine. We will have six rows each having six entries. For objects that are very close, the row will be {1,1,1,1,1,1} and for objects that are far away (100" or more) the row would be {0,0,0,0,0,1}. So our total array, named sequenceArray, would look as follows:

```
int sequenceArray [] [6] {
   {1, 1, 1, 1, 1, 1},       // all lights on, closest
   {0, 1, 1, 1, 1, 1},
   {0, 0, 1, 1, 1, 1},
   {0, 0, 0, 1, 1, 1},
   {0, 0, 0, 0, 1, 1},
   {0, 0, 0, 0, 0, 1}       // only one light on, farthest
};
```

Let's combine all the "setup" information for this program by using the information we used in the last program and adding the stuff we just discussed:

```
/* Ch_11_2_range_leds

  copy and paste this program into the Arduino IDE
  name this program Ch_11_2_range_leds

 This program uses the ultra-sonic range detector to measure the distance in inches to an object
 And light from one to six LEDs based upon the distance (all 6 closest)
 play a tone from MIN_FREQ = 60Hz to MAX_FREQ = 4KHz based upon distance (4HKHz is closest)
 turn on/off tone using NOMC push button switch
 */
// ---------------------------------------------------------------------------
#define ON            true
#define OFF           false

#define TRIG_PIN                4         // trigger pin on range detector is digital pin D4
#define ECHO_PIN                7         // echo pin is D7
#define BUZZER_PIN    8         // buzzer is on pin D8

#define TRIG_PULSE_WIDTH      15        // use 15 uSec for trigger pulse width
#define SAMPLES                         3         // number of distance samples to average
#define MAX_DISTANCE          100     // set maximum distance to 100 inches (quite arbitrary)

#define MIN_FREQ      60        // lowest frequency is 60 Hz
#define MAX_FREQ      4000      // highest frequency is 4KHz

int    playBuzzerFlag = OFF;      // global variable - changed by NOMC push button switch
int    distanceInches = 0;        // used to hold the returned value of distance in inches

#define  RED          3   // six colored LED pins
#define  ORANGE               5
#define  YELLOW       6
#define  WHITE        9
#define  GREEN        10
#define  BLUE         11

// this is the sequence for the 6 LEDs
 int   ledArray[] = {RED, ORANGE, YELLOW, WHITE, GREEN, BLUE};

int numCols           =           6;
int numRows           =           6;

int sequenceArray [] [6] {
   {1, 1, 1, 1, 1, 1},      // all lights on, closest
   {0, 1, 1, 1, 1, 1},
   {0, 0, 1, 1, 1, 1},
   {0, 0, 0, 1, 1, 1},
   {0, 0, 0, 0, 1, 1},
   {0, 0, 0, 0, 0, 1}      // only one light on, farthest
};
// ---------------------------------------------------------------------------
void setup() {
 Serial.begin(9600);
 pinMode(TRIG_PIN, OUTPUT);   // set up trigger pin as an output pin
 pinMode(ECHO_PIN, INPUT);    // set up echo pin as an input

 digitalWrite(TRIG_PIN, LOW);  // make sure trigger pin is low to start

 // configure the LED pins as outputs and turn off the LEDs
 for(int i = 0 ; i < numCols ; i++) {
            pinMode(ledArray[i], OUTPUT);
            digitalWrite(ledArray[i], LOW);
 }
}
```

Figure 10 CH_11_2_range_leds, part 1

Now let's turn our attention to the tasks to be performed within the loop() program itself. In addition to the activities we performed in the previous program we will need to:

- Monitor the push button switch using switchStatus() and toggle the global variable playBuzzerFlag from true to false and vice-versa every time the switch is pressed.
- Since we will be lighting one LED at 100" and 6 LED's at 0" we have 5 LED's to control over a distance of 100" or 20" per LED. We will need an index that will be used to select the row within our array sequenceArray[index][] and this index will be zero when the distance is between 0" – 20", 1 when the distance is >20" – 40", and so on up to 5 when distance >= 100".
-

For boolean variables (ones that can only be true or false) there is a useful operation that can be performed in C and C++. The NOT operator (!) will toggle the value of a boolean variable. This means that if the variable was true, it is changed to false and if it was false, it is changed to true. This is very useful since you don't need to know the current state of the variable if all you want to do is toggle it.

So if we want to toggle the state of the playBuzzerFlag every time the push button switch is pushed, all we need to do is:

```
if(switchStatus() == true)      playBuzzerFlag = !playBuzzerFlag;  // toggle the playBuzzerFlag
```

Take a shot a completing this program and then look at the next page to see one way to accomplish this.

```
void loop() {

  int pattern = 0;              // current pattern of LEDs to display based upon distance, index into sequenceArray[pattern][]
  int duration, freq;           // frequence for buzzer from 60Hz to 4KHz
  int distance = 0;             // initialize starting disance as zero
  int accumDistance = 0;        // reset accumulated distance to zero

  if(switchStatus() == true)     playBuzzerFlag = !playBuzzerFlag;  // toggle the playBuzzerFlag

  for(int i = 0 ; i < SAMPLES ; i++) {
   // send trigger pulse
   digitalWrite(TRIG_PIN, HIGH);
   delayMicroseconds(TRIG_PULSE_WIDTH);              // wait TRIG_PULSE_WIDTH micro seconds
   digitalWrite(TRIG_PIN, LOW);

   // wait for echo and measure it
   duration = pulseIn(ECHO_PIN, HIGH);
   distance =(duration/2)*.0135135;                  // get current distance measurement

   if(distance > MAX_DISTANCE)  distance = 100;      // largest distance allowed is MAX_DISTANCE inches
   if(distance <= 0)   distance = 0;                 // just to be sure you dont get a negative number

   accumDistance = accumDistance + distance;         // add current distance to accumulated distance
   delay(60);                                        // wait 60ms before taking another distance measurement
  }
  distance = accumDistance/SAMPLES;          // calculate the average distance

  // average distance now constrained from 0" to 100"
  freq = map(distance, 0, 100, MAX_FREQ, MIN_FREQ);       // translate from 0 - 100 to 4KHz to 60Hz

  if(playBuzzerFlag == ON)         tone(BUZZER_PIN, freq);
  else                              noTone(BUZZER_PIN);

  pattern = map(distance, 0, 100, 0, 5);

/* each row represents one LED illumination pattern, display the pattern for each LED in the row
  pointed to  by "pattern"
 */
  for(int i = 0 ; i < numCols ; i++) {
    if(sequenceArray [pattern][i] == 0)        digitalWrite(ledArray[i], LOW);
    else                                       digitalWrite(ledArray[i], HIGH);
 }

 }
// ---------------------------------------------------------------------------
boolean switchStatus() {
 static const int pushButtonSwitch = 2;                    // switch is on pin D2, declare as constant, hide from other functions
 pinMode(pushButtonSwitch, INPUT_PULLUP);                  // D2 INPUT mode and attach a PULLUP resistor

 if(digitalRead(pushButtonSwitch) == HIGH)          return false;          // if 5v (HIGH) switch is NOT pushed
 else                                               return true;           // if 0v (LOW) switch IS pushed
}
```

Figure 42 Ch_11_2_range_leds, part 2

12 CHALLENGE PROGRAMS

What follows are a number of challenge programs to increase your expertise and create additional cool ways for your Programmable Box to interact with the environment. You will be given the requirements and, if there is additional concepts, those will be covered. Go to the web at www.Your-Inner-Geek.com and look under DOWNLOADS for hints as well as the full solutions to these problems.

12.1 IMPROVING THE SWITCHPUSHED() FUNCTION

In Chapter 10, lesson 3, we introduced using the push button switch to move from one LED pattern to another. You may have noticed that when pressing the button, sometimes your program acted as if you had pressed the button more than once – and skipped one of the patterns. The reason for this is that when you push the button, the two pieces of metal that come together don't just come together and stay together, they bounce back away from each other for a fraction of a second and then come back together again. Sometimes they do this more than once before finally coming together and staying together.

This action is called switch bounce and ordinarily, if the switch is wired directly to an LED our eye can't see the LED going on and off that fast so we don't even realize that this is happening. But when we use a computer to detect the state of the switch, the computer can detect each of these bounces and count each one as a separate switch press. We want to get rid of this by adding "de-bounce" software so we only count one button-push not several.

Program Requirements:

- Name your program Ch_12_1_challenge_debounce_leds
- Modify the switchPushed() function as follows
- After the initial push is detected, wait 10 ms and read the switch status again. If no longer pushed, return not pushed if pushed then…
- As long as it stays pushed (while loop) sample, wait 10 ms, sample again. Each time the sample is "pushed" increment a loop counter. Exit the while loop when switch no longer pushed
- If loop counter >= 3, return "pushed"
- e.g. switch must remain pushed for 30 ms to count as "pushed"

12.2 VARY THE INTENSITY OF THE LEDS USING PULSE WIDTH MODULATION (PWM)

Another enhancement we can make to the three dimensional array program is to have the intensity of the LEDs controlled by the potentiometer. This is quite simple to do thanks to a built in Pulse Width Modulation (PWM) functionality contained in the Arduino.

In PWM, rather than have an output pin that drives an LED be always HIGH or always LOW, we can have is switch from HIGH to LOW very rapidly. For the Nano, the PWM frequency 490 Hz. To apply a PWM signal to a pin, simply use:

analogWrite(pinNumber, dutyCycle)

The dutyCycle specifies what portion of the time the pin is HIGH vs. LOW. For example, a dutyCycle of 0 means the pin is always LOW, a dutyCycle of 255 means the pin is always HIGH and a dutyCycle of 127 means the pin is HIGH 50% of the time and LOW 50% of the time.

So now that we know how to make the LED's brighter and dimmer using analogWrite(), we need to figure out how to determine the position of the potentiometer so we can use that information to control the brightness.

The potentiometer is connected to digital pin 7 so we know we are going to either use:

```
int       potPin  = 7;
```

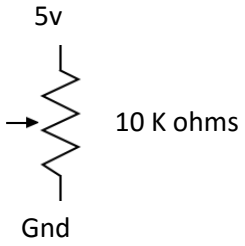
or

```
#define POTPIN 7
```

So that we can reference the correct pin.

The schematic symbol for a potentiometer is shown below. A potentiometer has three leads. The outer two leads are at the ends of the resistor (in our case, a 10K resistor) and the center lead goes to the "wiper". The wiper is a contact that can move from one end of the resistor to the other as you turn the potentiometer shaft.

In our case we have one end of the resistor connected to 5v and the other end to ground. As the wiper moves from one end of the resistor to the other, the voltage measured at the wiper will go from zero volts (when the wiper is at the ground end of the resistor) to 5v (when the wiper is at the 5v end of the resistor).



In order to read the value of the voltage on a pin, we use the Arduino function call analogRead(pinNumber).

The analogRead() function returns a value from 0 to 1023 with 1023 representing 5 volts.

So by now we have a pretty good idea of what we need to do:

Read the value of the potentiometer using analogRead(potPinNumber) and use that value to control the intensity of the LED using analogWrite(ledPinNumber, potValue). But wait a minute. When we read the

potentiometer value we get a value from 0-1023 and when we write a value to control the PWM we are limited to 0-255 so we are going to have to scale our potentiometer value by dividing by 4. That way the potentiometer value will be limited to 0-255. Just what the PWM function required.

Program Requirements:

- Name your program Ch_12_2_challenge_pwm_intensity_leds
- Use analogRead(potPinNumber) to get potentiometer position
- Scale that value by dividing by 4 to get the range to 0-255
- Use analogWrite(ledPinNumber, potValue/4) to control LED intensity

Note: In this case the scaling was easy, just divide by 4 but for more complicated scaling, there is a scaling function called:

map(valueToMap, fromLow, fromHigh, toLow, toHigh)

In our case we could have used this function as: map(potValue, 0, 1023, 0, 255);

13 APPENDIX:

Device	Analog/Digital pin number	Notes:
Red LED	Digital pin D3	HIGH to light
Orange LED	Digital pin D5	HIGH to light
Yellow LED	Digital pin D6	HIGH to light
White LED	Digital pin D9	HIGH to light
Green LED	Digital pin D10	HIGH to light
Blue LED	Digital pin D11	HIGH to light
Dual Color LED7 Blue Red	 Digital pin D12 Digital pin D14	 Low to light Low to light
Dual Color LED8 Blue Red	 Digital pin D13 Digital pin D15	 Low to light Low to light
Buzzer	Analog pin A8	tone()
Ultra sonic range Trigger Send Echo Receive	 Digital pin D4 Digital pin D7	
Potentiometer	Analog pin A7	
Push Button Switch	Digital pin D2	pinMode(2, INPUT_PULLUP)