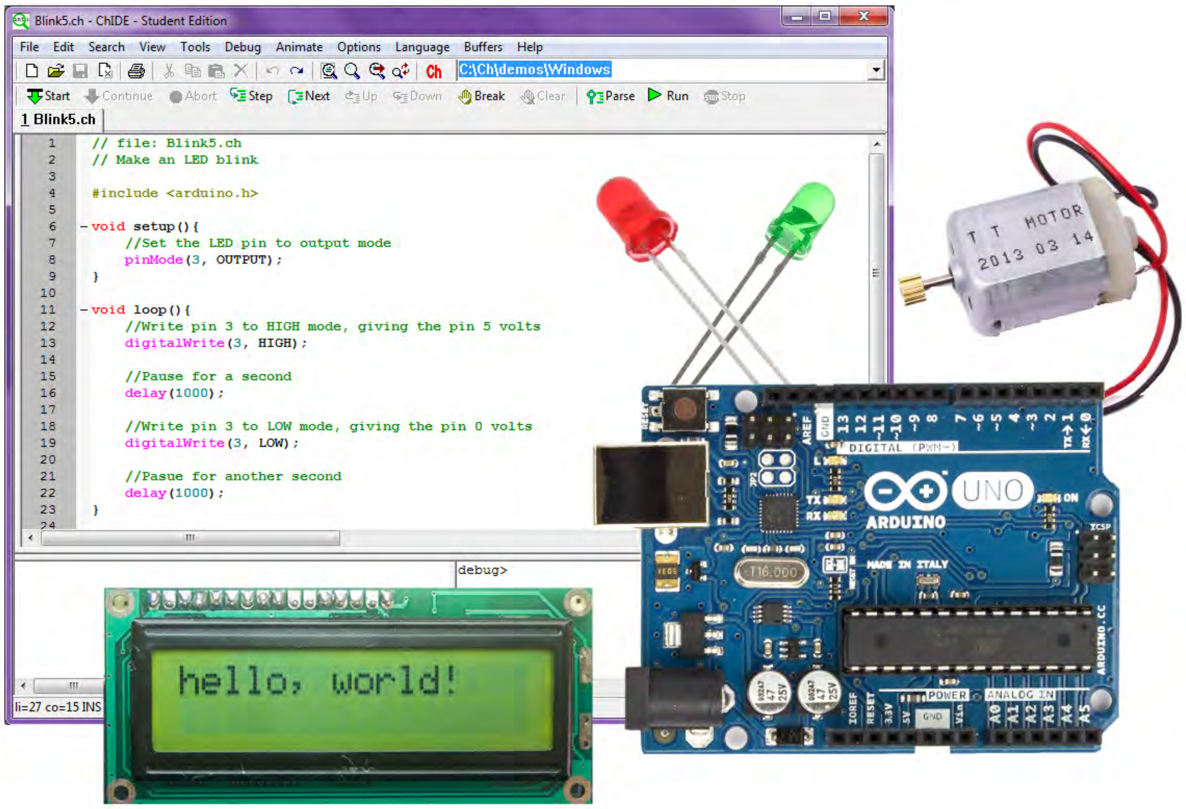
Learning Arduino with C Programming

Version 1.5


Blink5.ch - ChIDE - Student Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\Ch\demos\Windows
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 Blink5.ch
debug>
li=27 co=15 INS
hello, world!
T T MOTOR 2013 03 14
ARDUINO UNO

```
// file: Blink5.ch
// Make an LED blink

#include <arduino.h>

void setup(){
    //Set the LED pin to output mode
    pinMode(3, OUTPUT);
}

void loop(){
    //Write pin 3 to HIGH mode, giving the pin 5 volts
    digitalWrite(3, HIGH);

    //Pause for a second
    delay(1000);

    //Write pin 3 to LOW mode, giving the pin 0 volts
    digitalWrite(3, LOW);

    //Pasue for another second
    delay(1000);
}
```

Contents

1 Introduction

For engineers, artists, and students alike, the Arduino single-board microcontroller is one of the most popular of its kind in the world. The rise of the Arduino, and similar boards, has brought the diverse functionality and power of microcontrollers into the realm of the everyday person by not only displaying near unlimited usefulness in household tasks, but also by bringing to light the fun and creative side to engineering. This document will be a walk-through in how to program a microcontroller to interact with electronic components using ChIDE and the Ch programming language.

Ch and ChIDE provide the ability to enter a line-by-line debugging mode which can be invaluable to the absolute beginner programmer and breed a greater intuition about how a program actually works. As an interpreter, ChIDE also does not need to be re-uploaded to the Arduino after every change in the code, unlike with the Arduino IDE, meaning that Ch has a faster transition time between the editing and the execution of code, keeping students engaged and giving them more time in a classroom setting to troubleshoot or experiment. The Ch code is written in such a way that it is nearly identical to the what the Arduino code would be such that the code from a Ch program can be copied and paste into the Arduino programming environment and function the same way, creating a smooth transition from ChIDE to the Arduino IDE. This gives students easy experience with different programming environments and the dynamic nature of programming and programming languages. This book assumes that the user has the hardware from the **C-STEM Starter Kit** and **C-STEM Sensor Kit**. These kits are available for purchase from C-STEM Industrial Partners.

These projects and lessons provide a basic knowledge of how microcontrollers function with inputs (such as switches, knobs, temperature and light sensors) and outputs (such as LEDs, servos, motors). As an expansion from previous C-STEM Center courses involving mathematic computing and robotics programming with the Linkbot, this book gives practical applications for programming and explores some of the inner-working of robotics programming. This will eventually lead a user to having the required knowledge to use the Input/Output (I/O) capabilities of a microcontroller to create a self-driving, autonomous, vehicle. While a self-driving car is a classic goal for robotics, the possibilities for what a person can do with the power of a microcontroller is only limited by their creativity. The goal of this book is to give the user enough knowledge to express their creativity with their own projects and functionalities.

1.1 The Board

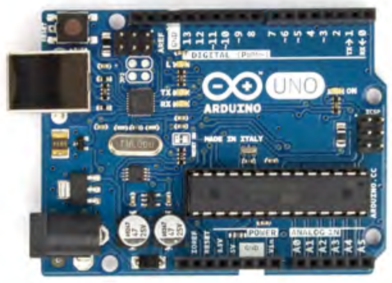


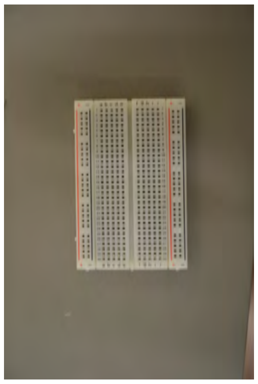
Figure 1: Arduino Uno Microcontroller Board

The centerpiece of the Arduino and other similar project boards is the microcontroller. A microcontroller is a simple computer that is used for specific, simple, tasks that require interfacing with external hardware, like reading information from a sensor or controlling a motor. This is unlike the microprocessor in a typical computer, which can run multiple programs at once and is used for general purposes. Microcontrollers are typically used for what are called embedded systems. Embedded systems are only programmed once for one task and contain only the electronic and mechanical components required for completing its task. A simple example would be the air-conditioning system inside of a house. The microcontroller inside of that little box on the wall takes in information from sensors that tell it what the current temperature is, looks at what buttons the homeowner has pushed to set it, and decides whether or not it needs to turn on the air-conditioning unit.

Aside from the microcontroller, the boards hold supporting hardware that physically allow the microcontroller to communicate with external devices, such as the computer the programmer is using. Most project boards will have a series of sockets, called pins, into which wires can be plugged to connect the board to I/O devices, like a button or LED for example. There are also pins for 5 volts, 3.3 volts, ground, and serial data. There will be both digital and analog pins, and a number of the digital pins will be PWM capable, which will be discussed later in Section 8. The controllers also need to communicate with other computers so that user instructions can be received or programs uploaded.

Ch code currently supports using a variety of Arduino boards, including the Arduino Uno board, which is the board included in the Arduino Starter Kit. Other supported boards include the Arduino Mega, Leonardo, Nano, and others. The differences between the spectrum of Arduino boards are the processor size and power, the board's physical size, and the number of I/O pins. The Uno can be considered the standard board and will work for all of the projects presented in this book. While other boards will work for projects in this book, they are more often used for specific circumstances or special applications.

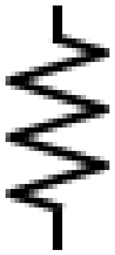
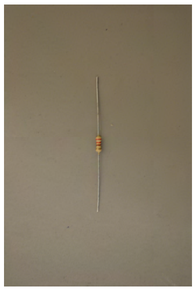
1.2 The External Circuitry



Breadboards

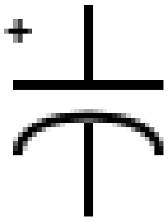
The circuits required for the projects cannot be wired directly to the Arduino. Well, they can but that would be very messy. That is why a breadboard is typically used for building temporary circuits. A breadboard is board with a grid of sockets, like those on the Arduino , that can have wires plugged into them. Some pins in the rows of the breadboard are connected internally so plugging certain into certain pins will connect those wires. Breadboards allow for relatively complex circuits to be built and modified easily for testing and troubleshooting.

Symbol **Hardware**

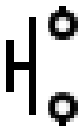
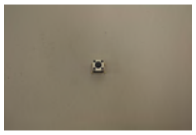
Resistors

The simplest electrical component, resistors do exactly what it sounds like they do, they resist current flow and cause drops in voltage. Resistors are used in filtering electric signals, controlling power input/output, and protecting other components from power overload. Resistors will be used frequently in this book to protect LEDs from too much power. They are color coded to indicate different values of resistance, measured in units called Ohms. Refer to 21.2 in the Appendix for the resistor color-coding system.



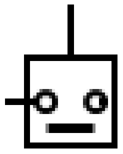
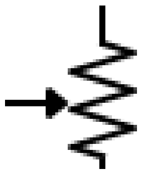
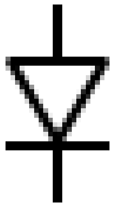
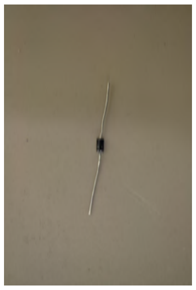
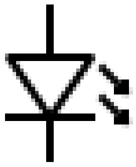
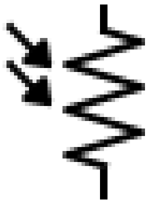
Capacitors

A capacitor is an electrical component that stores electrical energy. Once fully charged, it stops current flow completely and discharges its stored energy. Capacitors are typically used to store energy, like a temporary battery, and also for filtering electric signals. Capacitors will be used later in Sections 10 and 20 to smooth out the voltage changes across a servo.

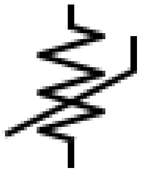



Switches/Buttons

After years of turning lights on and off, modern society has a pretty intuitive understanding of how switches work. A switch or button, when activated by being physically pressed, allows current to flow through a circuit. Switches allow the user/operator to have control and give input to a system. Switches are how the program knows what the user wants it to do.

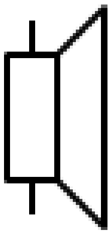
Symbol	Hardware	
		**Tilt Sensor** A tilt sensor is a special type of switch. It contains a little metal ball that, when the sensor is in an upright position, sits on two metal plates which allowing current to flow through the sensor. When the sensor is tilted, the little ball rolls off of the metal plates which stops the current and breaks the circuit.
		**Potentiometers** Another way a user can give input to a system is with a potentiometer, which is more commonly known as a knob. A potentiometer is technically a variable resistor. It can be a wide range of resistances which in turn creates a wide range of voltages that can be detected by the microcontroller. While switches can only be on or off, a potentiometer gives the microcontroller a wide range of feedback values. A classic implementation would be a volume knob on a speaker or stereo.
		**Diodes** A diode simply only allows electrical current to flow in one direction. A diode will be used later in Section 17 to prevent the current generated from a DC motor from damaging the circuit.
		**LEDs** LEDs, or Light Emitting Diodes, have been the primary light producing electrical component for the past few decades. They are a special type of diode. They are used widely as indicator lights on electrical devices.
		**Photo-resistors** A photo-resistor is a special type of resistor that changes its resistance when it is exposed to light. Photo-resistor acts as a variable resistor, similar to a potentiometer, and will create a wide range of voltages depending on how much light it is exposed to. Photo-resistors have many applications as switches where if something blocks a light signal a photo-resistor can detect it.

Symbol **Hardware**



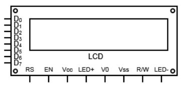
Temperature Sensors

A temperature sensor will change its resistance based on the temperature of the surrounding environment. The changing resistance changes the voltage output from the pins of the component, which the microcontroller can detect. The voltages can be transformed into temperatures with formulas, usually unique to the model of temperature sensor.



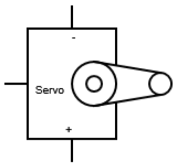
Piezos

A piezo is a buzzer that creates sound with a specific frequency for a specific input voltage. Depending on the voltage supplied it will vibrate, creating sound. It also can be used to detect vibrations. A piezo will be used later in Section 20 for detecting vibrations.



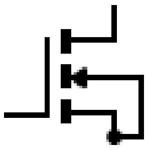
LCD Screens

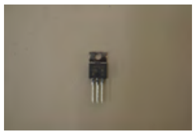
An LCD, or Liquid Crystal Display, is a way for the program to give the user visual feedback from the microcontroller. It can display alpha-numeric characters that can give the user information about data or errors and prompt the user to take some action.



Servo Motors

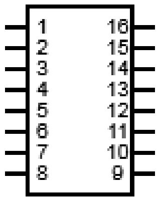
A special type of electric motor that can only rotate in a 180 degree range. The exact position the servo will move to is determined by the voltage or frequency the microcontroller sends to the servo.

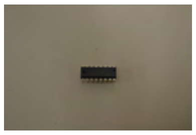




Transistors

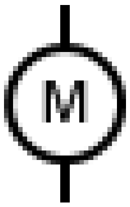
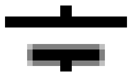
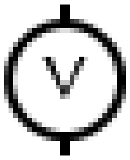
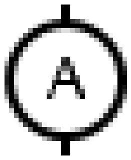
A transistor can be thought of as an electrical switch. When a voltage is applied to specific lead then current can flow through the other two leads. Transistors are used for many, many, applications and are the basis of modern computing.





H-Bridges

An H-bridge is a chip that controls the polarity, positive-to-negative or the negative-to-positive, of the voltage across a component. This is particularly useful in Section 18 to control the direction an electric motor will spin.

Symbol	**Hardware**	
		**DC Motor** A DC motor rotates a shaft when a voltage is applied to it. The direction of the shafts rotation depends upon the polarity of the voltage applied.
		**Battery** A battery provides power to an electric circuit or component. In this book batteries are used to power DC motors
		**Multi-meter** A multi-meter is a tool used to measure current, voltages, and resistances of electrical components or circuits. When measuring current, a multi-meter's symbol is a circle with an 'A' inside, and when measuring voltage the symbol is a circle with a 'V' inside.

1.3 The Breadboard

When looking at the breadboard with the shorter edge as the horizontal, notice that the two left-most and two right-most columns are marked with a '+' and a '-'. These are the power strips and each column is connected internally all the way across the board, so that if one wire is plugged into the far upper socket of a column and another wire is plugged into the far lower socket, then those two wires are connected at the same voltage. The strips marked '+' are typically where the positive power lead is connected and the strips marked '-' are where the ground lead is connected. Plug one end of a wire into one of the columns marked '+' and plug the other end into one of the Arduino pins marked '5V' and do the same for a '-' column and the pin marked 'GND'.

Look in between the two sets of power strips and see that there are two 5 x 30 socket grids with a trench in between them. The five sockets in each row are internally connected to each other so that if two wires are plugged into any of these 5 sockets then those wires are connected at the same voltage. The rows of one grid are not connected to the rows of the other grid, the trench separates them. Thus for any row, pins a, b, c, d, and e are connected internally and pins f, g, h, i, and j are also connected internally but separately from a-e.

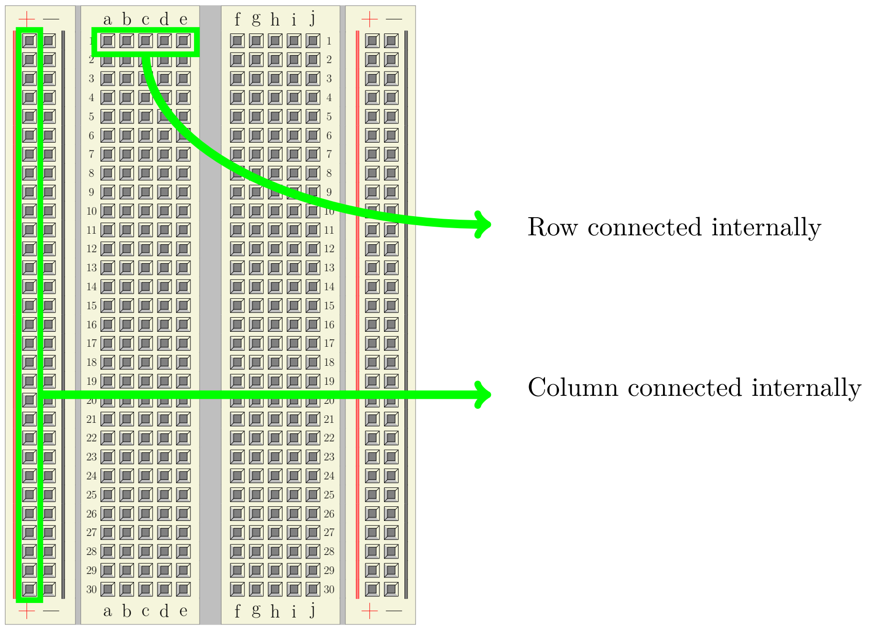


Figure 2: Breadboard Diagram

1.4 The Pins

On either side of the Arduino board is a series of ”pins” where wires can be plugged in. Each of these pins has a dedicated purpose, for example, some are for power, communication, or input/output for external devices. The table below explains the purpose of each of the pins on the Arduino.

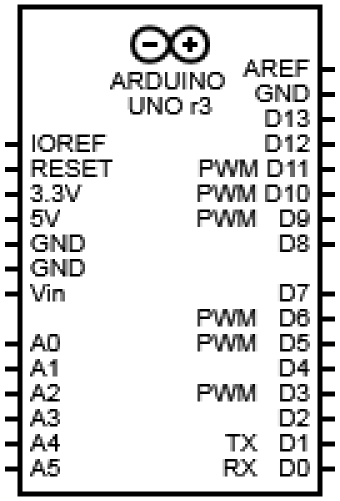


Figure 3: Arduino Circuit Diagram Symbol

Pin Label	Description
IOREF	This pin is at the voltage that the microcontroller normally runs at. The purpose is that any Shield (a smaller, separate, board that plugs into the Arduino for various purposes) can read the voltage on this pin and know what voltage to run at to be compatible with the Arduino. This pin does not have to be worried about for the scope of this book.
RESET	If this pin is made to be low, meaning if it is connected to ground, it will cause the Arduino to reset. This pin does not have to be worried about for the scope of this book.
3.3V	This pin provides 3.3 volts output
5V	This pin provides 5 volts output
GND	This is the ground pin. Ground is the zero-point reference for voltages. Think of it like the endpoint for a circuit. So, if every circuit starts at a high voltage, like 5 or 3.3, it needs to end at ground.
Vin	The Vin pin is where a user can plug in a battery or other power source. It serves the same purpose as the power jack or the usb cable in providing power to the Arduino. This pin does not have to be worried about for the scope of this book.

A0-A5	The are the analog Input/Output pins. They are used to read voltages from analog sensors. Remember, analog means that the voltage can be within a large spectrum. So, these pins can read all kinds of voltages between 0 and 5 volts, or between 0 and the voltage applied to the AREF pin.
AREF	Normally, the analog pins on the Arduino will only read voltages along a spectrum from 0 to 5 volts. The user can set the AREF pin to a certain voltage that will become the new maximum voltage that the analog pins can read. This pin does not have to be worried about for the scope of this book.
D2-D13	These are the digital Input/Output pins. They can read and output digital signals, meaning they can detect a certain level of voltage or output 5 volts. Also, the ones with a tilde, the squiggly line, next to them are PWM, pulse width modulation, capable meaning they can produce any voltage between 0 and 5.
Rx-Tx (D0-D1)	While technically two of the digital I/O pins, these two have a special purpose. These two pins are connected to the same communication line that the usb cable uses, where Rx is the receiving line and Tx is the transmitting line. These can be used for communicating with the Arduino through methods other than the usb cable, say with a bluetooth module.

Table 1: Arduino Pin Description Chart

1.5 Circuit Diagrams

Using words and pictures to describe how circuits are built can only go so far. As circuits become more and more complex it becomes harder and harder to describe them in simple terms, and pictures become an impenetrable nest of wires. Enter the circuit diagram. The standard circuit diagram is designed to layout how a circuit is built using lines as wires and simple symbols to represent the various electrical components. The symbols to the left of the component descriptions in the Section 1.2 are actually standard diagram symbols for each of those components. Figure 4 below is actually the circuit diagram for the second project. The little bits of zig-zag lines represent resistors, the plunger looking symbol represents a push-button switch, and the triangle with the line at the tip and the arrows represents an LED. Each of the lines represents a connection, in this case the connection can either be an actual wire or a row on the breadboard. Notice that sometimes, due to the complexity of the circuits, lines have to cross lines, such as in Figure 4 when the line comes out of the push-button and crosses the three lines connecting the Arduino to the resistors. This does not mean that these lines are physically connected when the circuit is built. Lines are only considered to be connected when there is a dot at the point where they intersect. Circuit diagrams are provided for each circuit for reference only. It can take some time to become used to, and interpret, circuit diagrams, so build the circuits using the descriptions and pictures while comparing these to the circuit diagrams and hopefully by the end of this book, you can become comfortable with how circuit diagrams work.

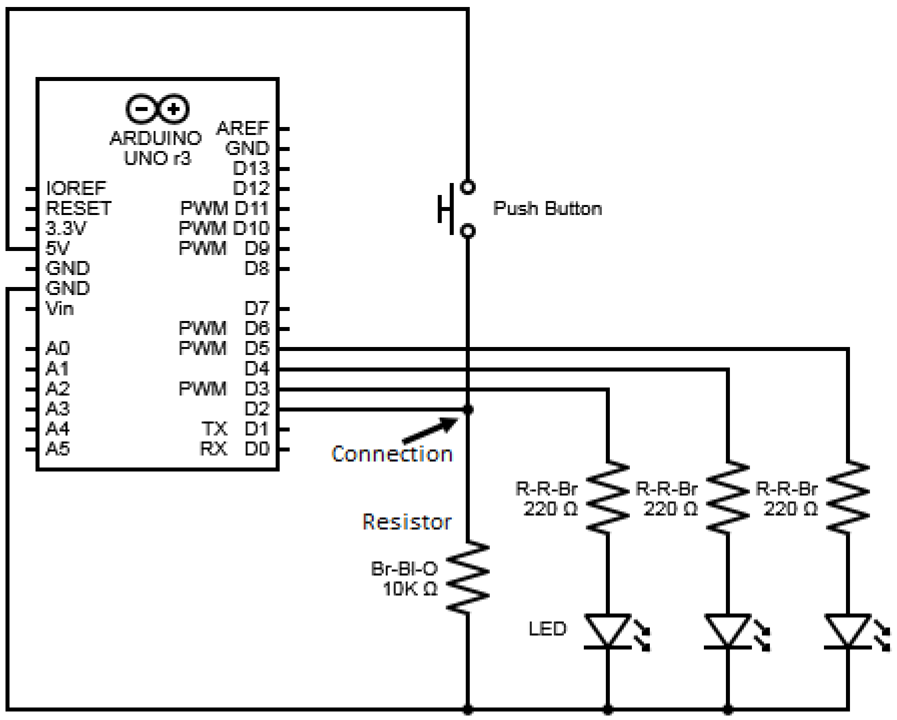


Figure 4: Example Circuit Diagram

2 Getting Started with Programming the Arduino in Ch

2.1 Project 1: Blink

Project Description: The first project will be the standard first step in the world of microcontrollers, making an LED blink. This project will have an LED turn on, blink a couple of times, and turn off.

2.2 New Concepts

This project is an introduction into how to control digital output devices, such as an LED. Digital means that there are only two possible states, on and off, which electrically means there is some level of voltage (on) or no voltage at all (off). For the Arduino, and most similar microcontrollers, the ”on” output voltage is 5 volts and the ”off” is 0 volts. The on state is also called the high state and is often represented in computing as a 1. The off state is also called low and is usually represented by a 0. For digital output devices, the microcontroller can be told to write a 1 to the pin the device is connected to, this will turn the device on. If the microcontroller is told to write a 0 to the device's pin then the device will be turned off.

In addition, this project will introduce the concept of the while-loop and act as an introduction in how to repeat segments of code a certain number of times or indefinitely.

New Functions

- `void pinMode(int pin, int mode)`
- `void digitalWrite(int pin, int value)`
- `void delay(int milliseconds)`

2.3 Required Components and Materials

- Breadboard
- 1 220Ω (R-R-Br) Resistor
- 1 LED

2.4 Building the Circuit

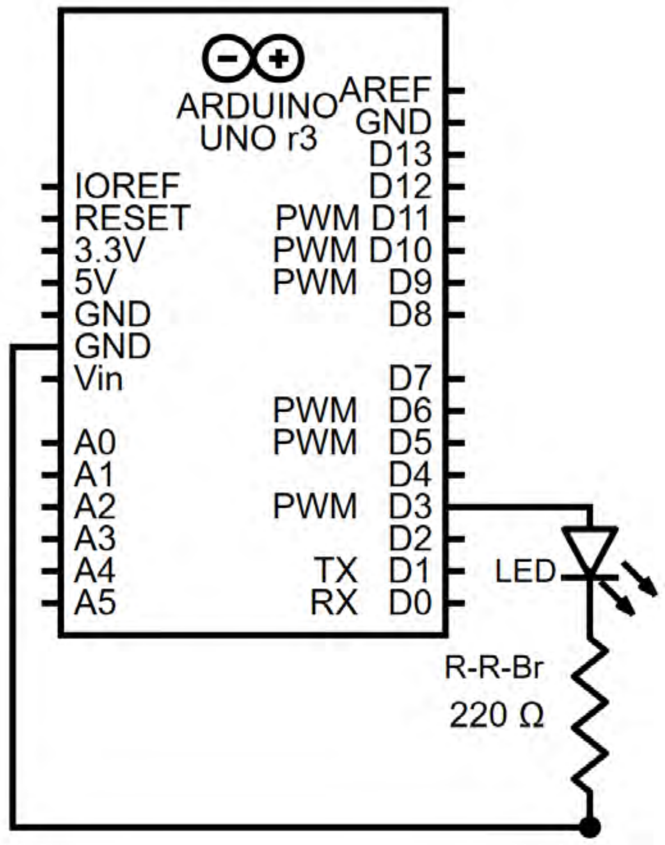
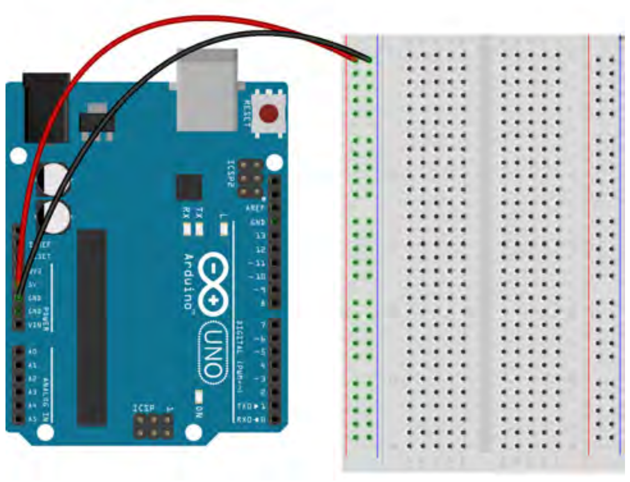


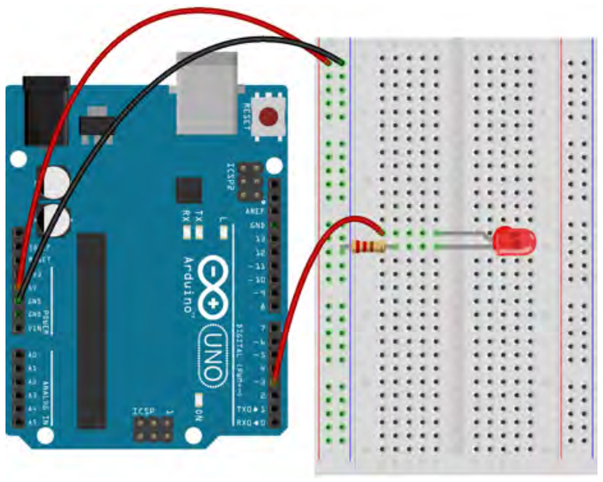
Figure 5: Blink Circuit Diagram

First, plug one end of a wire into one of the column marked '+' and plug the other end into one of the Arduino pins marked '5V' and do the same for a '-' column and the pin marked 'GND'.



Step 1

Now, to place an LED into the breadboard plug one of the wires coming out of the LED, called leads, into one row, and plug the other lead into a different row. Make sure the LEDs leads do not share the same row because then the two leads would be connected and the circuit will not work. Notice that one of the LED leads is longer than the other, this is the positive lead. Remember, LEDs are diodes and can only pass current in one direction so it is important to remember which lead is which. Put a kink in the longer lead so it will be easily recognizable later in this project.



Step 2

Plug one end of one Red-Red-Brown, or 220 Ohm, resistor (see Appendix Section 21.2) into the same row as the negative (shorter) lead of the LED and plug the other end into the same '-' column that 'GND' is plugged into.

Lastly, take a wire and plug one end into the row with the positive lead of the LED, the one that should have a kink in it. Plug the other end of the wire into digital pin 3 on the Arduino.

Pin	**Code**	**Input**	**Output**
D3	3		✓

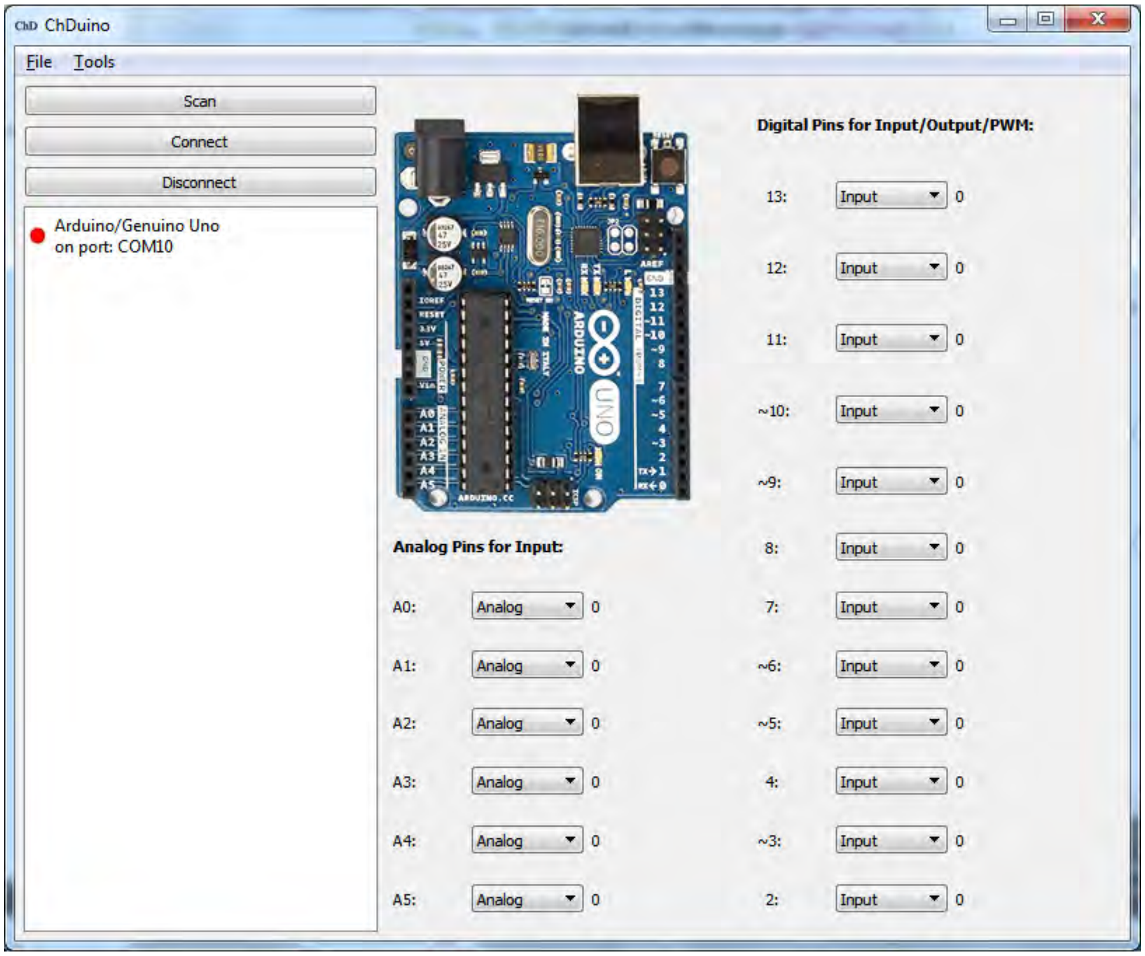
Table 2: Blink Project Pin Assignment Chart

2.5 Using the ChDuino GUI

The ChDuino GUI is a program designed to allow you interact with the Arduino's input and output pins. Using the GUI, you can view the analog values read by the A0-A5 ports, can view digital input values, and can control the digital output values.

2.5.1 Find and Manage Arduino Boards

First, plug the Arduino into your computer and open the "ChDuino" program. Once it opens, you should see a window that looks like this.



● Arduino/Genuino Uno
on port: COM10

			13:	Input	0
			12:	Input	0
			11:	Input	0
			~10:	Input	0
			~9:	Input	0
			8:	Input	0
A0:	Analog	0	7:	Input	0
A1:	Analog	0	~6:	Input	0
A2:	Analog	0	~5:	Input	0
A3:	Analog	0	4:	Input	0
A4:	Analog	0	~3:	Input	0
A5:	Analog	0	2:	Input	0

Figure 6: ChDuino program interface

On the left is a section containing a list of Arduino boards currently connected to the computer. Next to each Arduino, the program will list the COM port it is connected to and will show an option to update the device's firmware.
To connect, click the "scan" button. This tells the program to search for connected boards and refreshes the list to the left. To the left of each board is a dot indicating its connection status. If the dot is green, then the board is currently connected. If it is red, the board is not connected.

2.5.2 Connect and Control an Arduino Board

Click on the desired board, and then click the button that says ”Connect”. In the main window, you should see random values in the analog input readings. This is normal. When testing a project, you should only focus on pins that have components attached.

Analog Pins

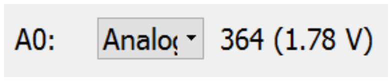


Figure 7: ChDuino Analogpin Value

Digital Pins

The digital pins have 3 modes: input, output, and pulse-width modification. To change between modes, click on the dropdown menu next to each digital pin number.

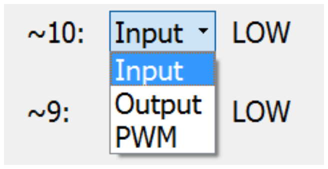
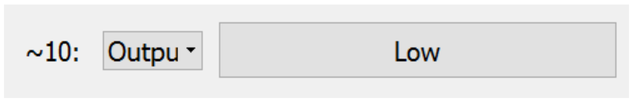
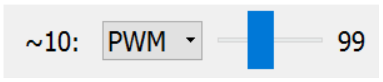


Figure 8: ChDuino Digital Value

”Input” can receive the values ”HIGH” and ”LOW”, corresponding to a digital 1 or 0. Output can also be set to ”HIGH” or ”LOW”, corresponding to an output of either 0V or 5V. Some pins are capable of pulse-width modification, indicated by a ”~” next to the pin number. ”PWM” allows the user to select an output voltage by dragging the slider bar ranging between 0 to 255.



(a) ChDuino Output Setting



(b) ChDuino PWM Value

Figure 9: Digital Pin Mode setting

2.6 ChDuino Basic Test

After building the circuit, open ChDuino and connect your Arduino. This first project will have you turn the LED on and off using the digital output port.

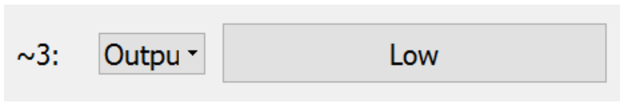
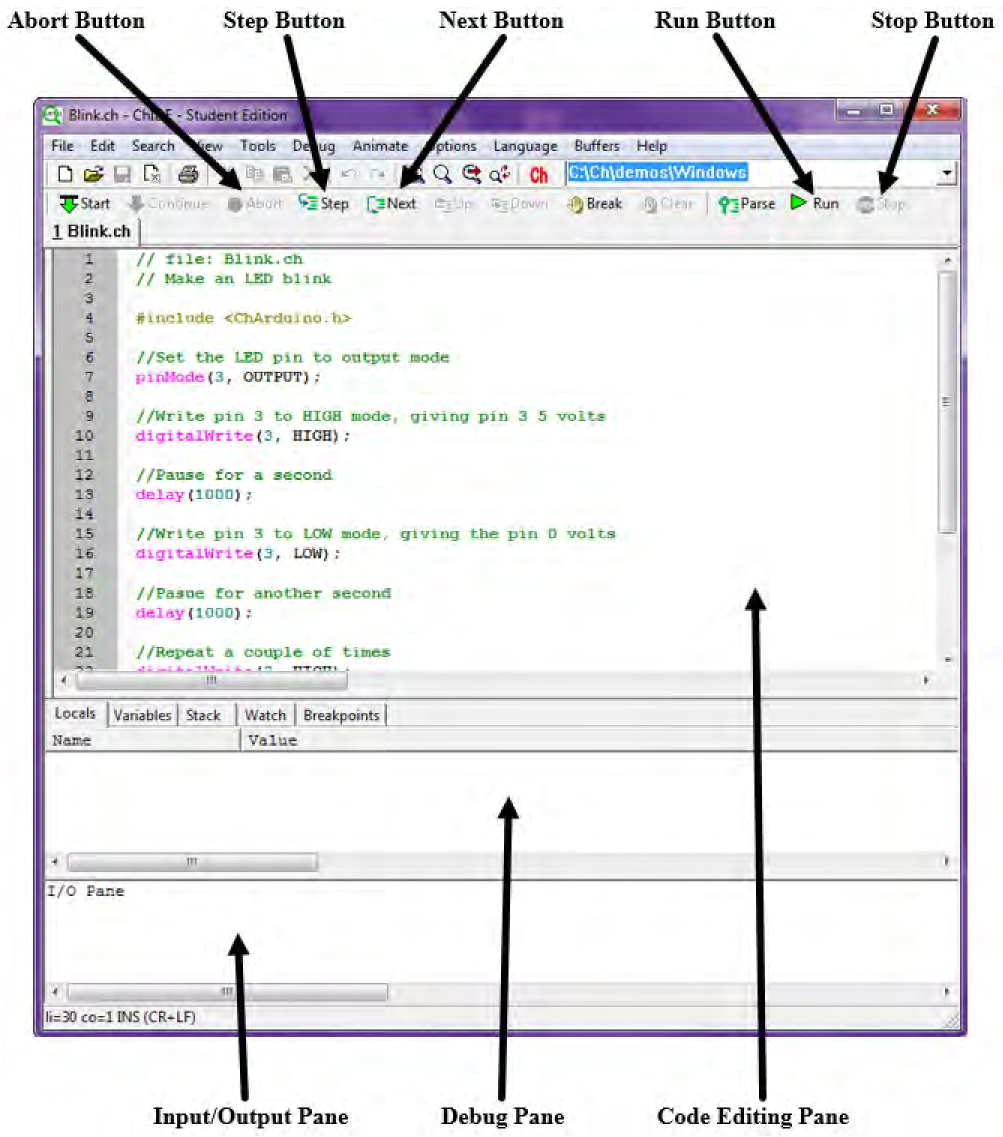


Figure 10: project1

- In ChDuino, set Digital Pin 3 to ”Output”. Practice turning the light on and off by pressing the ”HIGH” and ”LOW” button in the GUI. Clicking ”HIGH” tells the Arduino to send a voltage of 5V to the LED. ”LOW” tells the Arduino to send 0V to the LED.
- Plug the light into different digital pins and practice turning them on and off.

2.7 Using ChIDE to Run Programs

ChIDE is an Integrated Development Environment for the Ch language. It makes it incredibly easy to run, evaluate, and correct Ch programs. For those less familiar with ChIDE, a full users guide is available at `https://www.softintegration.com/docs`. To run a Ch program in ChIDE, write the code inside of the code editing pane and press the run button, shown below, located at the middle of the of the debug tool-bar at the top of the window. In order to stop a program that is already running, simply press the stop button located right next to the run button. If any of the panes shown in Figure 11 are not displayed they can be by clicking on the 'View' tab and selecting the pane that should be displayed. If there is an error, or if there is any information for the program to print out, it will be displayed in the Input/Output Pane shown in Figure 11.


Abort Button
Step Button
Next Button
Run Button
Stop Button
Blink.ch - ChIDE - Student Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\Ch\demos\Windows
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 Blink.ch
Locals Variables Stack Watch Breakpoints
Name Value
I/O Pane
li=30 co=1 INS (CR+LF)
Input/Output Pane
Debug Pane
Code Editing Pane

```
// file: Blink.ch
// Make an LED blink

#include <ChArduino.h>

//Set the LED pin to output mode
pinMode(3, OUTPUT);

//Write pin 3 to HIGH mode, giving pin 3 5 volts
digitalWrite(3, HIGH);

//Pause for a second
delay(1000);

//Write pin 3 to LOW mode, giving the pin 0 volts
digitalWrite(3, LOW);

//Pause for another second
delay(1000);

//Repeat a couple of times
```

Figure 11: ChIDE Window with Labels

The last step before the computer and Arduino are ready to talk to each other is that the 'ChArduino' firmware needs to be uploaded to the Arduino. When a program is run ChIDE should detect whether or not the current version of the firmware is installed and prompt the user to upload the correct firmware. Follow the prompts and the firmware should upload to the Arduino and everything should be ready to run a program. If for any reason the automatic firmware uploading does not function properly, a user can upload the firmware manually by following the directions in Section 21.9 in the Appendix.

2.8 Writing the Code

The following is the code for this project which will be explained step-by-step in this section:

```
// file: blink.ch
// Make an LED blink

#include <arduino.h>

//Set the LED pin to output mode
pinMode(3, OUTPUT);

//Write pin 3 to HIGH mode, giving the pin 5 volts
digitalWrite(3, HIGH);

//Pause for a second
delay(1000);

//Write pin 3 to LOW mode, giving the pin 0 volts
digitalWrite(3, LOW);

//Pasue for another second
delay(1000);

//Repeat a couple of times
digitalWrite(3, HIGH);
delay(1000);
digitalWrite(3, LOW);
delay(1000);
digitalWrite(3, HIGH);
delay(1000);
digitalWrite(3, LOW);
delay(1000);
```

First it should be noted that the green text in the code above are comments, lines of code that don't affect the program, meaning that they are not looked at by the program while it is running, and are just used to write notes. Before getting into the meat of the code, the header files must be included. The first thing that should happen in any program is calling the header files that the program requires. Header files are essentially lists of functions and other important information that would be annoying to put into every program so it is put into its own file that can be included into a program when necessary.

```
#include @<<arduino.h>>@
```

The header file `arduino.h` is included using the `#include @<<file.h>>@` directive. Including the `arduino.h` header file into the program lets the program use all of the `pinMode()`, `digitalWrite()`, and such functions. It also connects the computer to the Arduino and contains a lot of information for the computer that allows it to interact with the Arduino. This is how all of the project codes in this book will start. Now the meat of the code can be written but first it should be noted that the variable declaration and the function call lines end with a semi-colon (;), this is the code's way of marking when a line has ended. See that some lines of code, like including the header file, do not require a semi-colon. Which lines require a semi-colon and which don't are best learned through practice and trial and error.

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

After including the `arduino.h` header file, the program calls the `pinMode()` function. The generic version of this function is shown below. But what exactly are "`OUTPUT`" or "`INPUT`"? These are called macros. A macro is a sort of rule or permanent variable that the user has created and the program has to follow. For example, `OUTPUT` and `INPUT` are macros defined in `arduino.h` as equal to 1 and 0 respectively. The program is able to use them because `arduino.h` was included in the beginning. The macros are similar to variables but more far reaching. `OUTPUT` is automatically equal to 1 in every program that includes `arduino.h` whereas normal variables only apply to the program that they are in. Why use macros? Sometimes it is useful to use a macro to replace a number that is used a lot. In the case above however, the use of macros is for clarification. If the user had to put a 0 or 1 as the mode argument in `pinMode()` it would be very easy to mix up which number tells the function to make an input pin and which number tells it to make an output pin. The macros help the user to remember which is which, and help other people understand the code better.

```
void pinMode(int pin, int mode)
```

This function requires two arguments and returns `void` (meaning that nothing is returned). The `void` before `pinMode` means that the function does not return any value. Arguments are the things inside the parenthesis, separated by commas, that tell the function what to do or what to work on. The first argument is an integer number that is the pin number, letting the function know what pin it is supposed to act on. The second argument, separated from the first by a comma, is an integer representing the pin mode, letting the function know whether to make the pin number specified in the first argument an input or an output. This function is only necessary for digital pins. In this case, pin 3, which is connected to the LED, is set to output mode.

```
digitalWrite(3, HIGH);
```

After setting the mode of pin 3, it is time to turn on the LED connected to pin 3. The program accomplishes this using the `digitalWrite()` function, whose generic form is shown below.

```
void digitalWrite(int pin, int value)
```

This function has two arguments, an integer representing the pin that the microcontroller should write to, and an integer representing whether the pin should be written high or low. Remember that digital devices can only be on or off, otherwise called high or low. The high and low modes are usually specified by the macros, HIGH and LOW, respectively. HIGH is defined as 1 and LOW is defined as 0. In this case high means 5 volts and low means 0 volts. The previous code segment will set digital pin 3 to high, or 5 volts. This will turn the LED on.

```
delay(1000);
```

Now that the LED is turned on, the program moves on to another new function, `delay()`. The generic form is shown below.

```
void delay(int milliseconds);
```

There is only one argument for this function, an integer representing the number of milliseconds. A millisecond is 1/1000th of a second. The `delay()` function will pause the program for the specified number of milliseconds. In the previous segment of code, the argument was 1000 so the `delay()` function will pause the program for 1000 milliseconds, or 1 second. The purpose of this function in the program is to create some time between turning the LED on and turning it off. It determines how long the blink lasts, otherwise the LED would turn on and off so fast that it would be indiscernible to the human eye.

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

Once the delay function is done, the program moves on and calls the `digitalWrite()` function again. This time, the function changes pin 3 to low mode, meaning that there is 0 volts coming from this pin and the LED will be off.

```
delay(1000);
```

Lastly, the program runs another delay function, pausing the program for 1000 milliseconds, or 1 second. The program then goes on to repeat these same function calls two more times. So, if everything is working properly, the LED should blink three times and turn on and off at one second intervals.

2.9 Alternate Method Using the While-loop

It is common in programming where the programmer wants a certain part of the code to repeat itself either a certain number of times or indefinitely. This is most often accomplished using the programming concept of loop, one of which will be introduced in this section, called the while-loop. The general form of a while loop is shown in the code snippet below.

```
while(condition){
...code...
}
```

A while loop works by examining the condition or statement inside the parenthesis after the `while`. If what is in the parenthesis is true then the program will loop back and run the code contained inside the braces, `{..code..}`, again. **WHILE** the condition is true the program will keep repeating the loop. For example, if there was a while-loop of the form `while (i < 5) {.....}` The program will loop through the code inside the while-loop until 'i' is greater than or equal to 5 and the condition becomes false.

In the previous section, an LED was made to blink three times with the code that made it blink written out separately each time. A while-loop can be utilized to accomplish the same task without having to write the same code over and over again, making the code shorter and simpler. Shown below is some code that will accomplish the same task as the code in Section 2.8 using a while-loop.

```
// file: blink2.ch
// Make an LED blink

#include <arduino.h>

//Declare a variable to act as a counter
int i = 0;

//Set the LED pin to output mode
pinMode(3, OUTPUT);

while(i < 3){
    //Write pin 3 to HIGH mode, giving the pin 5 volts
    digitalWrite(3, HIGH);
    //Pause for a second
    delay(1000);

    //Write pin 3 to LOW mode, giving the pin 0 volts
    digitalWrite(3, LOW);

    //Pasue for another second
    delay(1000);

    //Add one to the counter variable
    i = i + 1;
}
```

A few other modifications need to be made so that the while-loop will work. First, the program creates an `int`, or integer variable, called 'i'. This variable will work as a counter and keep track of how many times the program has repeated the while-loop. It is initialized to zero so that the counter starts at zero. Second, a line, `i = i+1`, is added at the very end of the code segment inside of the while-loop. This will increase the value of 'i' by one and then assign it to 'i' as the new value of 'i'. The program will begin and reach the while-loop with the `(i < 3)` condition and, because the current value of 'i', 0, is less than 3, the program will run the code inside of the while-loop. Once the program is at the end of the while-loop, the value of 'i' is increased from 0 to 1 by the `i = i+1` line. This process repeats until 'i' gets increased to 3. Now the condition of the while-loop is no longer true and the program does not run the loop and ends.

But what if the programmer wants the while-loop to repeat forever? For computers, true is the same as 1 and false is the same as 0, this is called boolean algebra. If the condition statement of a while-loop is `(i < 3)` and 'i' is indeed less than 3, then the computer will read the whole statement as a 1, or true. So, if the condition of a while-loop is simply set as 1, so that it looks like `while(1){...code...}`, then the condition is always true, so to speak, and the program will loop through the while-loop forever. This is called an infinite while-loop. The code below is the same code as the previous segment of code, except the condition of the while-loop is changed to be simply 1 and the variable 'i' is removed because a counter is no longer necessary. This code will blink the LED on and off forever, until the programmer stops the program.

```
// file: blink3.ch
// Make an LED blink

#include <arduino.h>

//Set the LED pin to output mode
pinMode(3, OUTPUT);

while(1){
    //Write pin 3 to HIGH mode, giving the pin 5 volts
    digitalWrite(3, HIGH);

    //Pause for a second
    delay(1000);

    //Write pin 3 to LOW mode, giving the pin 0 volts
    digitalWrite(3, LOW);

    //Pasue for another second
    delay(1000);
}
```

2.10 Exercises

1. Modify the code so that the LED will blink one more time
2. Modify the code so that the LED will blink faster. How fast can it blink before you can not tell it is blinking? What happens at that point?

3 Function Definitions and Arduino's Setup and Loop Functions

Inside of every Arduino program are two primary functions, **setup** and **loop()**. These two functions appear in the forms:

```
void setup(){
..code..
}
```

and

```
void loop(){
..code..
}
```

The program will only run the code inside of the **setup()** function once. Thus for Arduino, the code inside the **setup()** function would typically be all of the **pinMode()** functions or anything other functions that only need to happen once at the beginning of the program. The variable declarations are typically not included in the **setup()** function, and instead are placed before the **setup()** function. The program will run the code inside of the **loop()** function over and over again indefinitely, similar to the **while(1)** while-loop used in the previous Ch project. The code that typically is put inside of the **loop()** function for Arduino programs is the **digitalWrite()** functions or any function or code that needs to run over and over again.

The **setup()** and **loop()** functions, as they are shown in the two code snippets above, are actually not functions as have been used previously, but something called a function definition. A function definition is where the code defines a new function's name, the number and types of variable the function will take in and output, and the code that determines what the function does. An example would be if a function was defined like the code snippet below.

```
void setup(){
   pinMode(2, OUTPUT);
   pinMode(3, INPUT);
}
```

This code would tell the program that there is a new function, called **setup()**, that does not take in any arguments and that returns **void** (meaning that nothing is returned). When this new function, **setup()**, is called then two **pinMode()** functions are run, the first setting pin 2 to **OUTPUT** mode and the second setting pin 3 to **INPUT** mode. The Arduino programming environment will automatically call the **setup()** and **loop()** functions when the program is run so that the programmer does not have to call the functions themselves in the code. So, an actual Arduino program, written in the Arduino IDE, would look something like the code snippet below.

```
int pin = 2;

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

void loop(){
        digitalWrite(pin, HIGH);
        delay(500);
        digitalWrite(pin, LOW);
        delay(500);
    }
```

The code just defines the `setup()` function to set the mode of pin 2 to `OUTPUT` mode and defines the `loop()` function to `digitalWrite()` pin 2 to `HIGH` and `LOW`, alternating every 500 milliseconds. In Arduino programs these functions are automatically run when the program is run. This can be similarly accomplished in Ch where the `setup()` and `loop()` functions are defined the same, but the program includes the actual function calls. The while-loop version of the Blink project is shown below using `setup()` and `loop()` functions instead.

```
// file: blink4.ch
// Make an LED blink

#include <arduino.h>

void setup(){
    //Set the LED pin to output mode
    pinMode(3, OUTPUT);
}

void loop(){
    //Write pin 3 to HIGH mode, giving the pin 5 volts
    digitalWrite(3, HIGH);

    //Pause for a second
    delay(1000);

    //Write pin 3 to LOW mode, giving the pin 0 volts
    digitalWrite(3, LOW);

    //Pause for another second
    delay(1000);
}

int main(){
    int i = 0;
    setup();
    while(i < 3) {
        loop();
        i = i+1;
    }
    return 0;
}
```

Notice that the code above is very similar to the alternate methods for the Blink project found in Section 2.9. The `pinMode()` function is placed inside of the `setup()` function definition. The main body of the code is now placed inside of the `loop()` function definition instead of being directly inside of a `while(i < 3)` while-loop. The important part is the few lines of code added onto the end that call the `setup()` and `loop()`

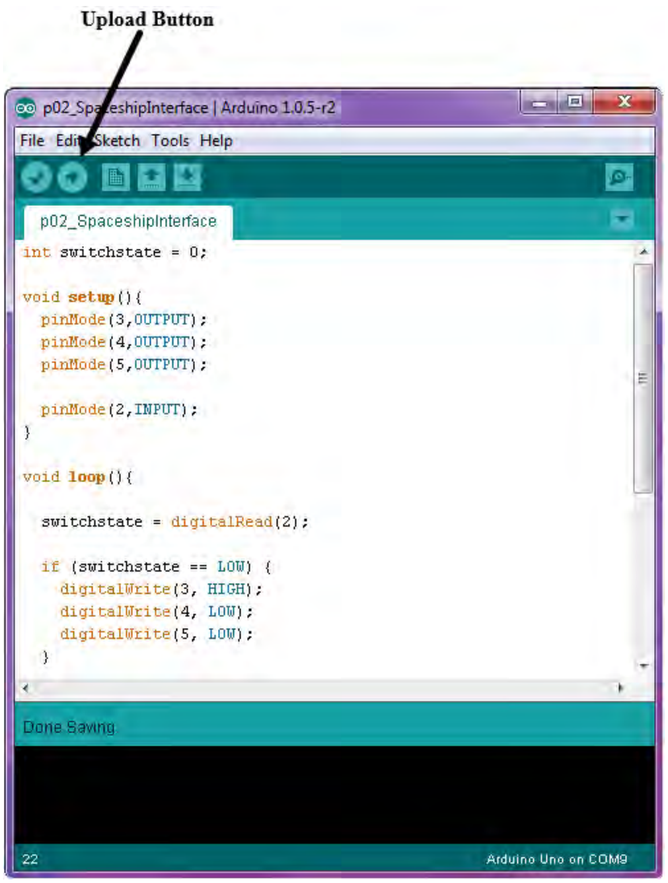
functions. These function calls are placed inside of a special function, called **main()** which is typically in the form of the code snippet below.

```
int main(){
    ...code...
    return 0;
}
```

main() is a special function inside of the C programming language that defines what the actual program is, in other words, the main code. It returns a 0 to let the user know that the program was run successfully. Inside of the **main()** function, an integer variable, called 'i', is declared and initialized to zero. The **setup()** function is then called once. The **loop()** function is placed inside of the while-loop so that it will repeat until the variable 'i' is greater than or equal to 3. An infinite while-loop can also be used with this method, shown in the code below. All of the following Ch projects will be coded using this method.

```
// file: blink5.ch
// Make an LED blink

#include <arduino.h>

void setup(){
    //Set the LED pin to output mode
    pinMode(3, OUTPUT);
}

void loop(){
    //Write pin 3 to HIGH mode, giving the pin 5 volts
    digitalWrite(3, HIGH);

    //Pause for a second
    delay(1000);

    //Write pin 3 to LOW mode, giving the pin 0 volts
    digitalWrite(3, LOW);

    //Pasue for another second
    delay(1000);
}

int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

4 Programming with the Arduino IDE


Upload Button
p02_SpaceshipInterface | Arduino 1.0.5-r2
File Edit Sketch Tools Help
p02_SpaceshipInterface
Done Saving
22
Arduino Uno on COM9

```
int switchstate = 0;

void setup(){
  pinMode(3,OUTPUT);
  pinMode(4,OUTPUT);
  pinMode(5,OUTPUT);

  pinMode(2,INPUT);
}

void loop(){

  switchstate = digitalRead(2);

  if (switchstate == LOW) {
    digitalWrite(3, HIGH);
    digitalWrite(4, LOW);
    digitalWrite(5, LOW);
  }
```

Figure 12: Arduino IDE

Now that some experience has been gained in using the ChIDE to control the Arduino board, it would be beneficial to learn how to program the board using Arduino's IDE. An IDE is an Integrated Development Environment and essentially is a application that makes it easier to write, build, and run code. ChIDE is the IDE for the Ch programming language and Arduino has their own IDE for their programming language shown above in Figure 12. In order to program with the Arduino IDE, write code in the window and press the upload button, the circle with an arrow inside located in the upper left of the window. The Arduino needs to be connected to the computer, via a USB cable, while uploading the code, but after can be disconnected and still run the program. This differs from Ch which needs to maintain a connection with the Arduino in order to control it. The Arduino programming language is slightly different than the Ch language, as can be seen in the code segment below.

```
// file: blink5.ino

void setup(){
    //Set the LED pin to output mode
    pinMode(3, OUTPUT);
}
```

```
void loop(){
    //Write pin 3 to HIGH mode, giving the pin 5 volts
    digitalWrite(3, HIGH);

    //Pause for a second
    delay(1000);

    //Write pin 3 to LOW mode, giving the pin 0 volts
    digitalWrite(3, LOW);

    //Pasue for another second
    delay(1000);
}
```

This is the Arduino code for the last iteration of the Blink project from Section 3 and there are some small but still fundamental differences from the Ch just used. In the Arduino language there is no need to include a header file for simple functions like **pinMode()** or **digitalWrite()**. This is because these functions are automatically included in the Arduino IDE. Variables are declared the same way but Arduino introduces a **setup()** function to contain all for the **pinMode()** functions and other similar preparation for the main part of the code. For the main part of the code, Arduino replaces the **while(1)** that was used in the first project with a function called **loop()** that will loop indefinitely. Notice that the Arduino code in the segment is the exact same as the **loop()** and **setup()** functions definitions in the Ch version of the code in Section 3. For all of the following projects, you can copy the part of the Ch code that contains the variable declarations through the **loop()** and **setup()** functions definitions then paste it into the Arduino IDE and it will function in the same way.

Instead on pressing a Run button to execute code, the Arduino IDE has a Upload button that sends all of the code to the Arduino board at once. This is differs from ChIDE which sends the code to the Arduino one line at a time to make features, like the debug mode, possible. It is important to note that if any code is uploaded to the Arduino board using the Arduino IDE it will erase the firmware on the Arduino that makes it work with ChIDE. Because of this, each time you transition from using the Arduino IDE back to ChIDE you need to reinstall the firmware to the Arduino board. The program will usually detect that the proper firmware is not on the Arduino and prompt the user to upload the most recent firmware, but in case of errors the firmware can be manually uploaded using the instructions found in Section 21.9.

ChIDE	Arduino IDE
• Connection to Arduino must be maintained while a program is running • **setup()** and **loop()** functions must be called inside of the program • Arduino must have special arduino firmware installed in order to work with ChIDE. The firmware will need to be reinstalled after any program is uploaded to the board from the Arduino IDE	• Once program is uploaded, the board can be disconnected from the computer • **setup()** and **loop()** functions are automatically called

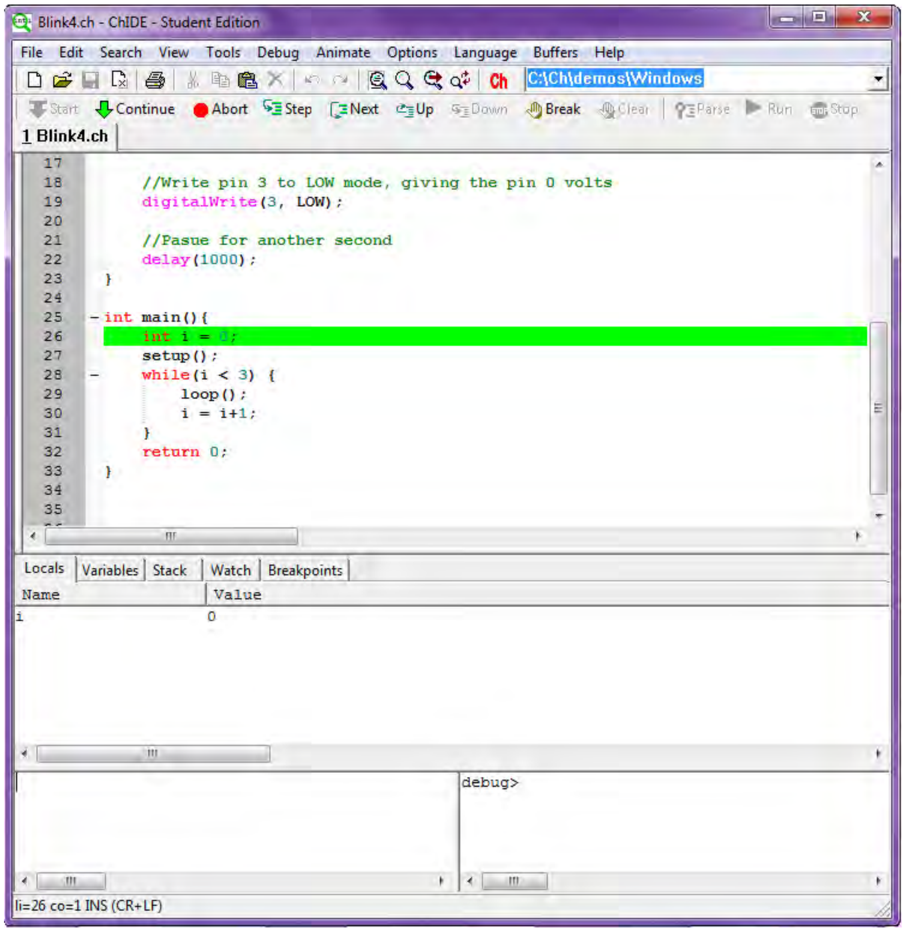
Table 3: A comparison programming the Arduino board from ChIDE and from the Arduino IDE

To make the code written in Ch completely compatible with the Arduino IDE, simply add an `#ifdef @<_CH_>@` statement. The `#ifdef` statement is a preprocessing directive, like the `#include` statements used to add header files. Each `#ifdef` statement need a corresponding `#endif` statement to come after it. It works like an if-statement, discussed later in Section 6, where if the macro `_CH_` is defined then the code between the `#ifdef @<_CH_>@` and the `#endif` will be executed. The macro `_CH_` is only defined when running code in the ChIDE. A code example of how to use this method is shown below. `#ifdef @<_CH_>@` statements are placed around the `#include @<<arduino.h>>@` line and the `main()` function which means when this code is run in ChIDE these lines are included and when the code is run in the Arduino IDE these lines are ignored. Thus, the code below can be run in ChIDE and then copied over to the Arduino IDE, without any changes, and run exactly the same way.

```
// file: blink6.ch
// Make an LED blink

#ifdef _CH_
#include <arduino.h>
#endif

void setup(){
    //Set the LED pin to output mode
    pinMode(3, OUTPUT);
}

void loop(){
    //Write pin 3 to HIGH mode, giving the pin 5 volts
    digitalWrite(3, HIGH);

    //Pause for a second
    delay(1000);

    //Write pin 3 to LOW mode, giving the pin 0 volts
    digitalWrite(3, LOW);

    //Pasue for another second
    delay(1000);
}

#ifdef @<_CH_>@
int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
#endif
```

5 Using Debug Mode to Understand and Troubleshoot Programs

The debug mode within ChIDE can be very useful. It allows a programmer to troubleshoot a program by going through it line-by-line and see why the program is not functioning the way it should. For example, the programmer can look to see how a certain variable is changing or if a loop is acting the way it should. Pressing either the Step or Next buttons will enter the program into debug mode. Starting the debug mode will bring up the debug pane and the Input/Output pane, if it is not already open. This section will go through the fourth version of the Blink project, with the `setup()` and `loop()` functions and `(i < 3)` while-loop condition, in debug mode to illustrate how it can be beneficial. Pressing the Step or Next buttons will enter the program into debug mode and highlight the first line, as shown in Figure 13.


Blink4.ch - ChIDE - Student Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\Ch\demos\Windows
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 Blink4.ch
Locals Variables Stack Watch Breakpoints
li=26 co=1 INS (CR+LF)

```
        //Write pin 3 to LOW mode, giving the pin 0 volts
        digitalWrite(3, LOW);

        //Pasue for another second
        delay(1000);
    }

    int main(){
        int i = 0;
        setup();
        while(i < 3) {
            loop();
            i = i+1;
        }
        return 0;
    }
```

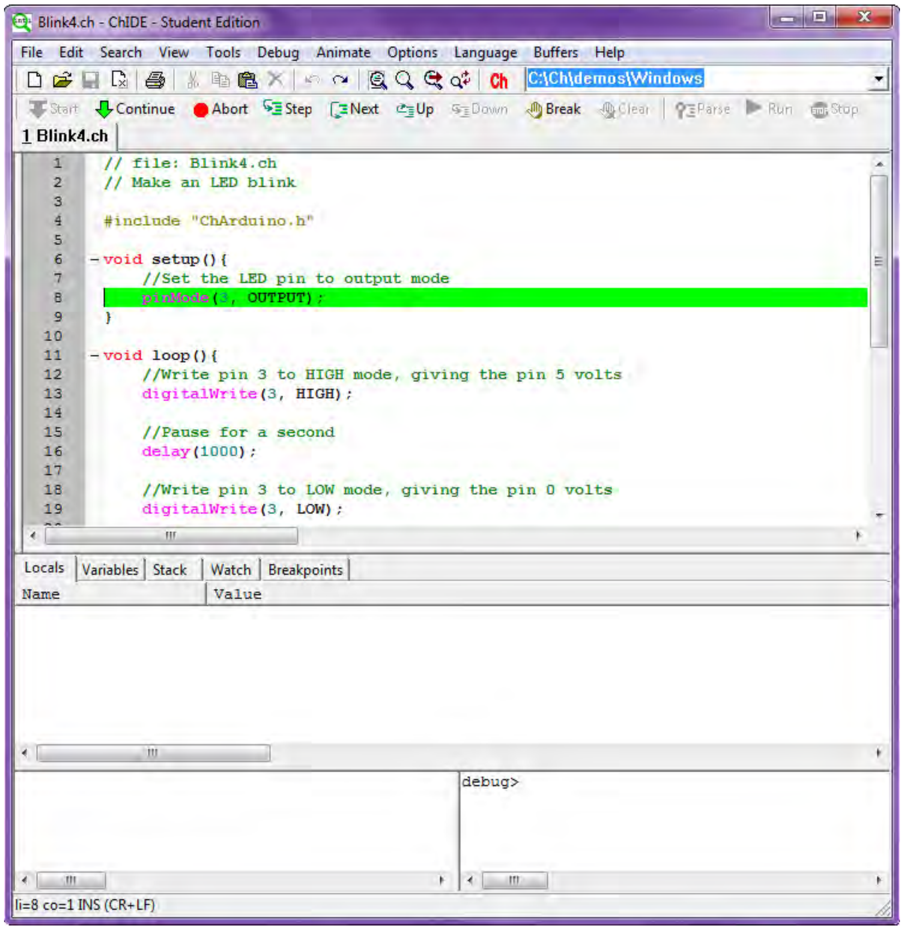
```
Name                Value
i                   0
```

```
debug>
```

Figure 13

Notice that the first line that the program runs is not the literal first line, but the first line inside of the `main()` function. Remember, the `main()` function is what the C programming language uses to define what the actual program is. So, `int i = 0;` is the first line in the actual program. That line will create an integer variable, called 'i', and set it equal to zero. Look into the debug pane and see that there is a variable listed, named 'i', and its current value is 0. The debug pane will list all of the variable and their current values. This is very helpful for troubleshooting because it is easy to see how the variable are changing and pin point where something is going wrong. However, the debug pane will only list the variable for the current scope. The scope is an important programming concept where variables are only relevant to a certain function or part of the code, depending on where that variable is declared. So, because the variable 'i' is defined inside of the **`main()`** function it cannot be used inside of the **`setup()`** or **`loop()`** functions and visa versa. In the next few steps you will see that when the program enters the **`setup()`** or **`loop()`** functions the 'i' variable will no longer be listed in the debug pane.

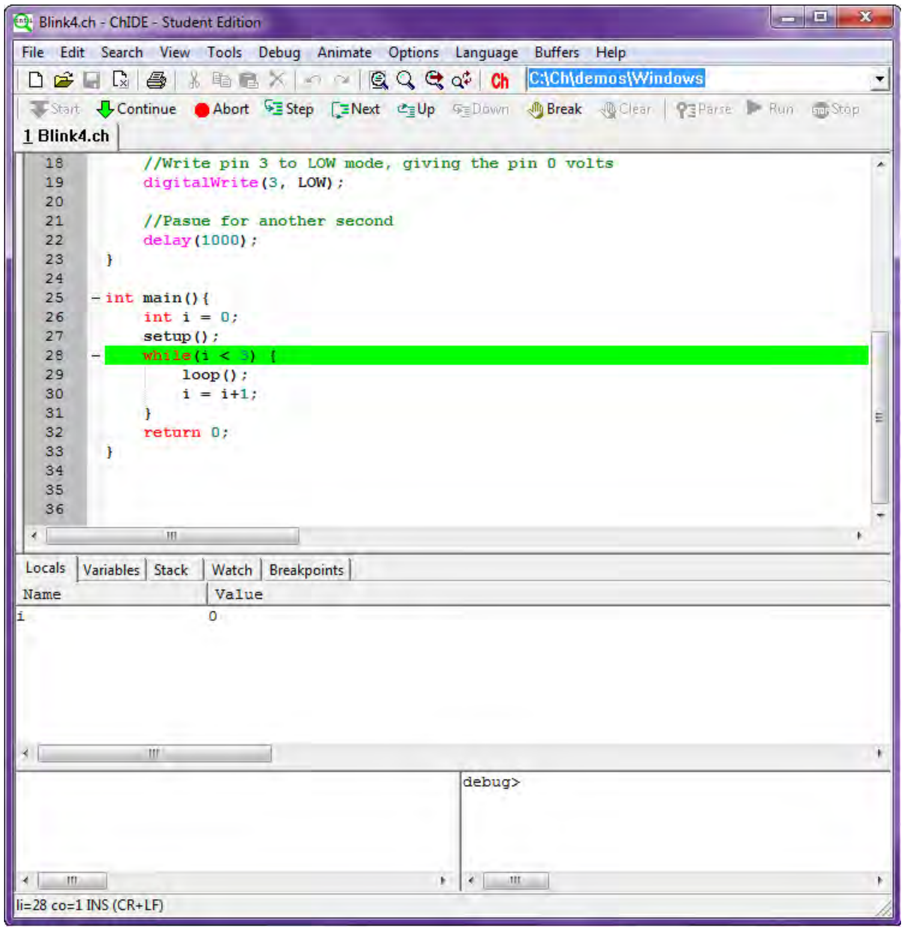
Press the Next button again and the program should move on to the next line, **`setup();`**, which should now be highlighted. Now press the Step function to enter the **`setup()`** function. Pressing the Step button will enter a function and let you go through every line in the function, as opposed to the Next button which will just run the function and move on. So, after pressing the Step button the ChIDE window should look like Figure 14.


Blink4.ch - ChIDE - Student Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\Ch\demos\Windows
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 Blink4.ch
Locals Variables Stack Watch Breakpoints
Name Value
debug>
li=8 co=1 INS (CR+LF)

```
// file: Blink4.ch
// Make an LED blink

#include "ChArduino.h"

void setup(){
    //Set the LED pin to output mode
    pinMode(3, OUTPUT);
}

void loop(){
    //Write pin 3 to HIGH mode, giving the pin 5 volts
    digitalWrite(3, HIGH);

    //Pause for a second
    delay(1000);

    //Write pin 3 to LOW mode, giving the pin 0 volts
    digitalWrite(3, LOW);
```

Figure 14

The `pinMode()` function inside of the **`setup()`** function should now be highlighted. Press the Next button and the `pinMode()` function will be run. Always press the Next button, as opposed to Step, on functions like `pinMode()` or `digitalWrite()` because then the debug mode will enter the code inside of these functions and that is beyond the scope of this book. After pressing Next and executing the `pinMode()` function, the function should return back to the **`main()`** function and the ChIDE window should look like Figure 15.



Blink4.ch - ChIDE - Student Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\Ch\demos\Windows
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 Blink4.ch
Locals Variables Stack Watch Breakpoints
li=28 co=1 INS (CR+LF)

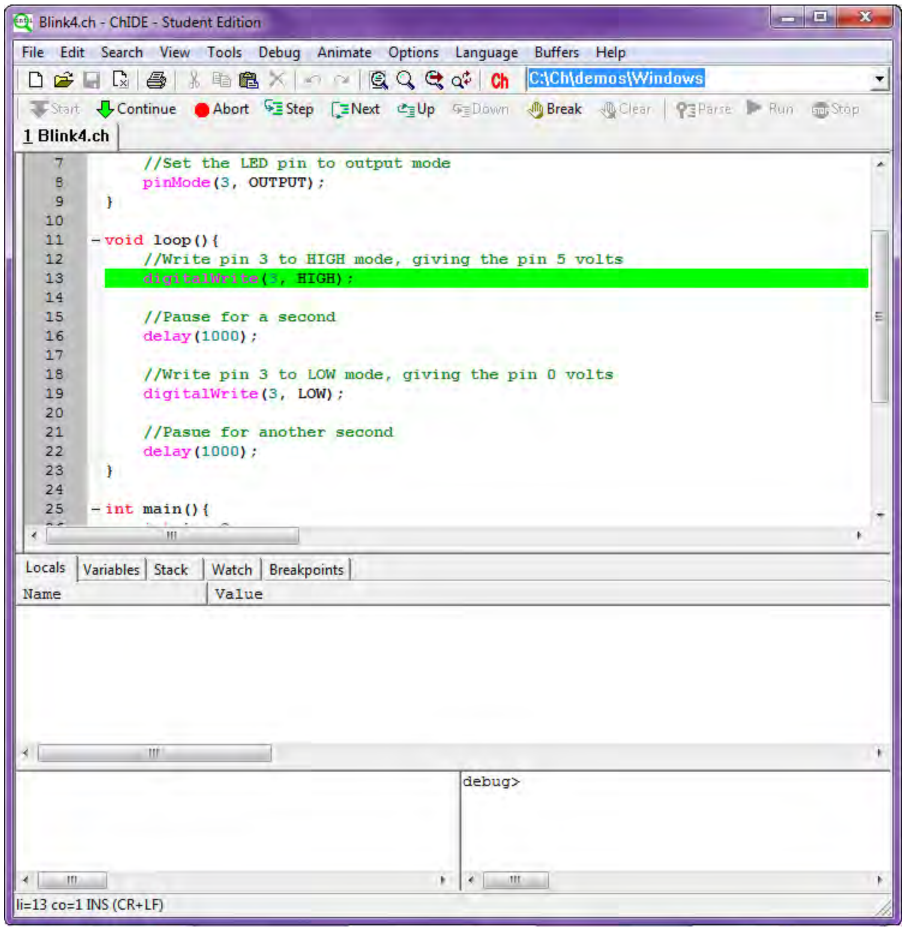
```
    //Write pin 3 to LOW mode, giving the pin 0 volts
    digitalWrite(3, LOW);

    //Pasue for another second
    delay(1000);
}

int main(){
    int i = 0;
    setup();
    while(i < 3) {
        loop();
        i = i+1;
    }
    return 0;
}
```

```
Name          Value
i             0
```

```
debug>
```

Figure 15

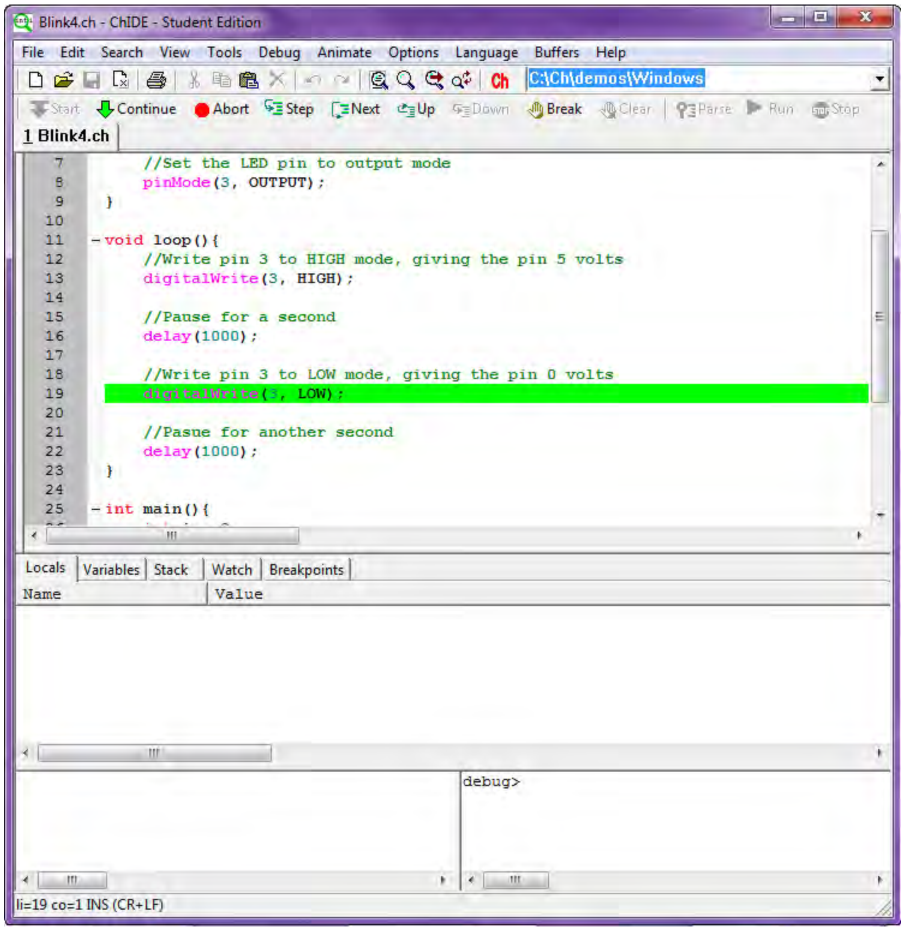
The while-loop should now be highlighted. Pressing Next again will enter the while-loop and highlight the `loop()` function. Press the Step button to enter the `loop()` function. The first `digitalWrite()` function should be highlighted and the window should look like Figure 16.


Blink4.ch - ChIDE - Student Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\Ch\demos\Windows
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 Blink4.ch
Locals Variables Stack Watch Breakpoints
Name Value
debug>
li=13 co=1 INS (CR+LF)

```
    //Set the LED pin to output mode
    pinMode(3, OUTPUT);
}

void loop(){
    //Write pin 3 to HIGH mode, giving the pin 5 volts
    digitalWrite(3, HIGH);

    //Pause for a second
    delay(1000);

    //Write pin 3 to LOW mode, giving the pin 0 volts
    digitalWrite(3, LOW);

    //Pasue for another second
    delay(1000);
}

int main(){
```

Figure 16

Press the Next button and the code will run the highlighted `digitalWrite()` function and the LED should light up. Now the first `delay()` function should be highlighted. Press the Next button and the program will run this `delay()` function, pausing the program for one second, and moving on to where the second `digitalWrite()` function is highlighted. The window should now look like Figure 17.


Blink4.ch - ChIDE - Student Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\Ch\demos\Windows
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 Blink4.ch
Locals Variables Stack Watch Breakpoints
Name Value

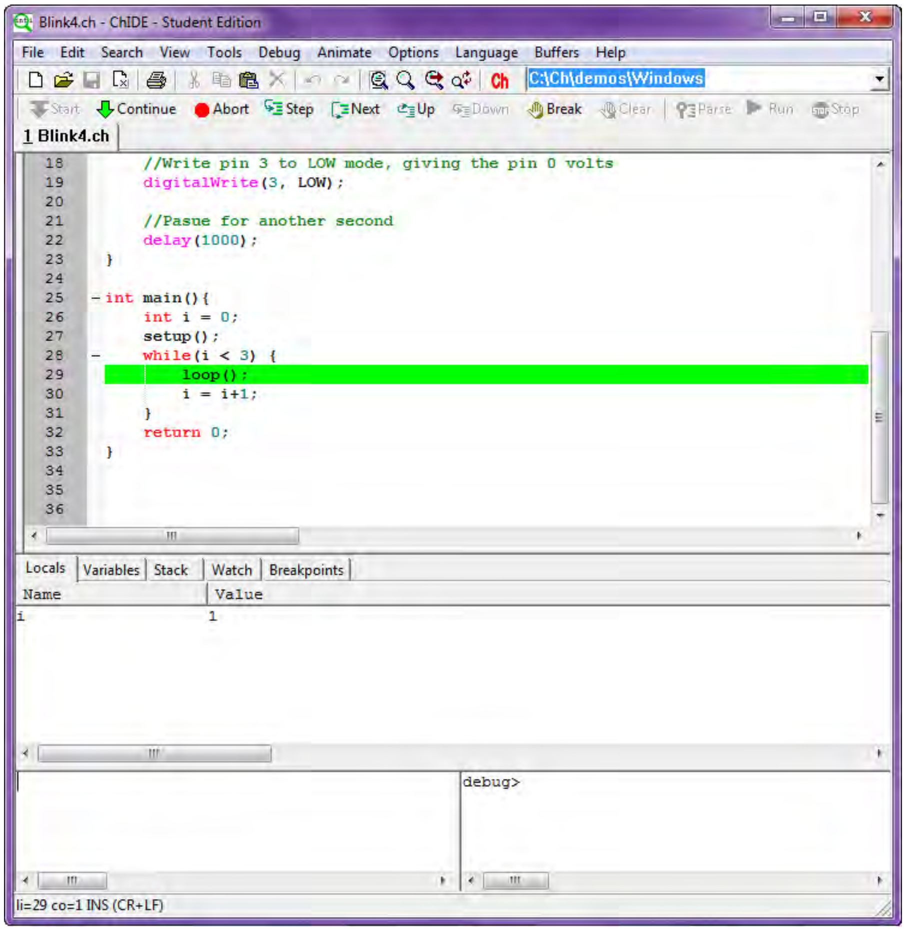
```
    //Set the LED pin to output mode
    pinMode(3, OUTPUT);
}

void loop(){
    //Write pin 3 to HIGH mode, giving the pin 5 volts
    digitalWrite(3, HIGH);

    //Pause for a second
    delay(1000);

    //Write pin 3 to LOW mode, giving the pin 0 volts
    digitalWrite(3, LOW);

    //Pasue for another second
    delay(1000);
}

int main(){
```

```
debug>
```

```
li=19 co=1 INS (CR+LF)
```

Figure 17

If you press the Next button the program will run the highlighted `digitalWrite()` function and the LED will turn off. The second `delay()` function should now be highlighted. Press the Next button and the program will run the `delay()` function, pausing the program for one second and completing this iteration of the `loop()` function. This will bring the program out of the `loop()` function and back to the `main()` function where the `i = i+1;` line should be highlighted. Run this line by pressing the Next button and the window should look like Figure 18.


Blink4.ch - ChIDE - Student Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\Ch\demos\Windows
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 Blink4.ch

```
    //Write pin 3 to LOW mode, giving the pin 0 volts
    digitalWrite(3, LOW);

    //Pasue for another second
    delay(1000);
}

int main(){
    int i = 0;
    setup();
    while(i < 3) {
        loop();
        i = i+1;
    }
    return 0;
}
```

Locals Variables Stack Watch Breakpoints

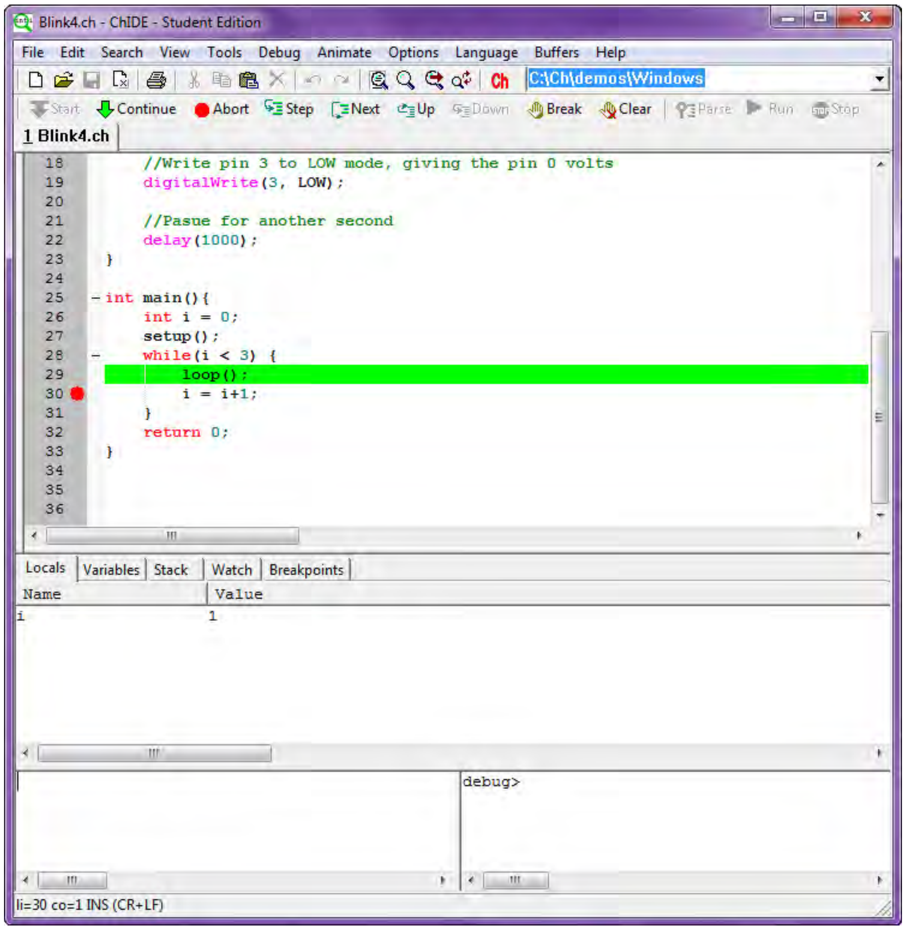
Name	Value
i	1

```
debug>
```

```
li=29 co=1 INS (CR+LF)
```

Figure 18

Notice that after running the **i = i+1** line the value of the 'i' variable in the debug pane has increased by 1, changing from 0 to 1. Now, lets say we want to see how the value of 'i' changes as the while-loop repeats itself but we don't want to keep going through the **loop()** function or pressing Next over and over again. This can be accomplished using a break-point and the Continue button, found next to the Abort button. The Continue button will run the program like normal until it hits a break-point and then it stops. To insert a break-point, select the line where you want the program to stop then bring the mouse over to the line number for that line. Now, bring the mouse slightly to the right of the line number, still in the same shaded column, and click the mouse. A small red dot should appear, indicating that there is a break point on that line. Insert a break-point on the line with **i = i+1** and the window should look like Figure 19.


Blink4.ch - ChIDE - Student Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\Ch\demos\Windows
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 Blink4.ch
Locals Variables Stack Watch Breakpoints
li=30 co=1 INS (CR+LF)

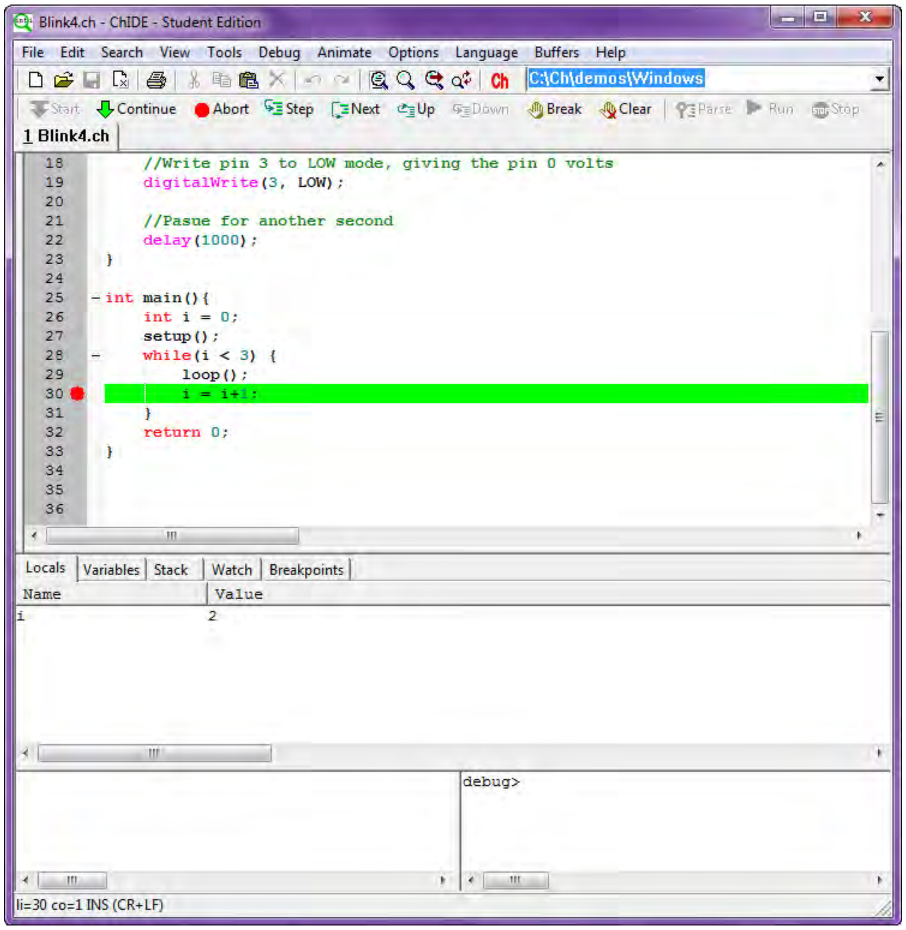
```
      //Write pin 3 to LOW mode, giving the pin 0 volts
      digitalWrite(3, LOW);

      //Pasue for another second
      delay(1000);
  }

- int main(){
      int i = 0;
      setup();
-     while(i < 3) {
          loop();
          i = i+1;
      }
      return 0;
  }
```

```
Name                 Value
i                    1
```

```
debug>
```

Figure 19

Now hit the Continue button and the program should run until it hits the break-point. Because the program was left highlighted on the line with the `loop()` function, it will only run the one line before hitting the break-point at the `i = i+1` line. Hit the Continue button again and the program will run all the way through the while-loop before getting back to the break-point. The window should look like Figure 20.


Blink4.ch - ChIDE - Student Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\Ch\demos\Windows
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 Blink4.ch
Locals Variables Stack Watch Breakpoints

```
        //Write pin 3 to LOW mode, giving the pin 0 volts
        digitalWrite(3, LOW);

        //Pasue for another second
        delay(1000);
    }

-   int main(){
        int i = 0;
        setup();
-       while(i < 3) {
            loop();
            i = i+1;
        }
        return 0;
    }
```

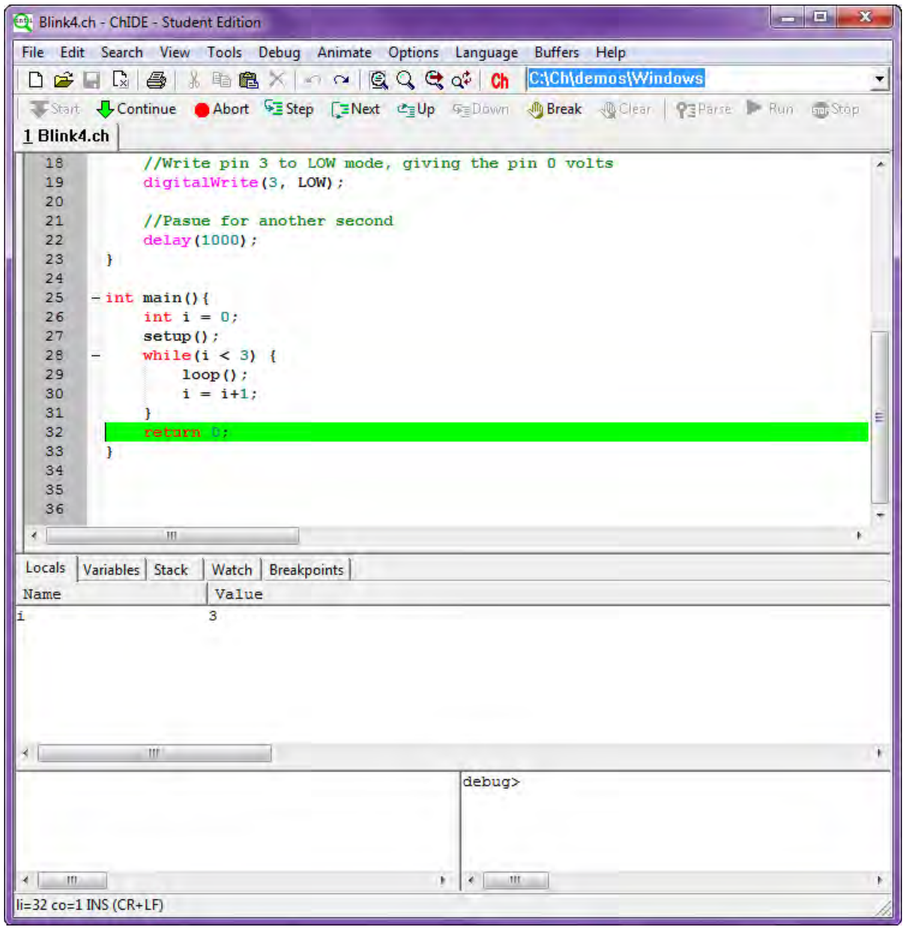
Name	Value
i	2

```
debug>
```

```
li=30 co=1 INS (CR+LF)
```

Figure 20

Notice that because the program has gone through the while-loop another time, the value of the 'i' variable is now 2. Now, press the Next button and see what happens. The window should now look like Figure 21.


Blink4.ch - ChIDE - Student Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\Ch\demos\Windows
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 Blink4.ch
Locals Variables Stack Watch Breakpoints

```
        //Write pin 3 to LOW mode, giving the pin 0 volts
        digitalWrite(3, LOW);

        //Pasue for another second
        delay(1000);
    }

int main(){
        int i = 0;
        setup();
        while(i < 3) {
            loop();
            i = i+1;
        }
    return 0;
}
```

```
Name                Value
i                   3
```

```
debug>
```

```
li=32 co=1 INS (CR+LF)
```

Figure 21

Because the last pressing of the Next button ran the `i = i+1` line when the value of 'i' was 2, the value was increased to 3. See in the debug pane that the value of 'i' is now 3. A value of 3 for 'i' no longer satisfies the while-loop condition that `(i < 3)`. So, the while-loop finishes and the program moves on to the next line after the while-loop, `return 0;`, which in this case is the last line in the program. Press the Next button again and the program should end.

The debug mode in ChIDE is a wonderful way to track the way a program functions, line-by-line, making it easy to intuitively understand how fundamental programming concepts, like a while-loop, work.

6 Interfacing with a Push-Button

6.1 Project 2: Spaceship Interface

Project Description: For the second project, three LEDs are going to be controlled based upon the users input, in the form of pressing a button. Specifically, a green LED will be turned on during the time when a button is not pressed, and when the button is pressed, two red LEDs will turn on and off consecutively.

6.2 New Concepts

This project will serve as an introduction to receiving digital inputs. Remember, digital means that there are only two possible states, on and off. Typically the "on" voltage for digital reading is around 2 volts, where below 2 volts is off and above 2 volts is on. So when reading data from the pin connected to a digital input device, like a button, that isn't activated the microcontroller will read a 0 and when the device is activated it will read a 1. To accomplish these tasks, this project will introduce if-else statements and a function to read digital inputs.

New Functions

- `int digitalRead(int pin)`

6.3 Required Components and Materials

- Breadboard
- 1 Push-Button Switch
- 1 10kΩ (Br-Bl-O) Resistor
- 3 220Ω (R-R-Br) Resistor
- 3 LED

6.4 Building the Circuit

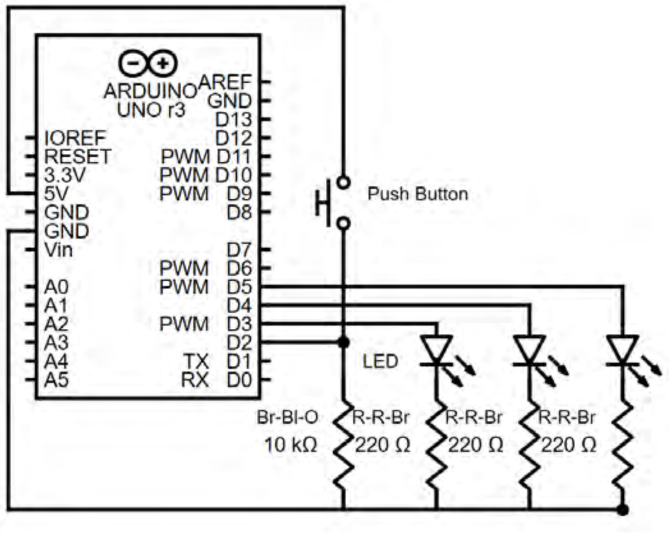
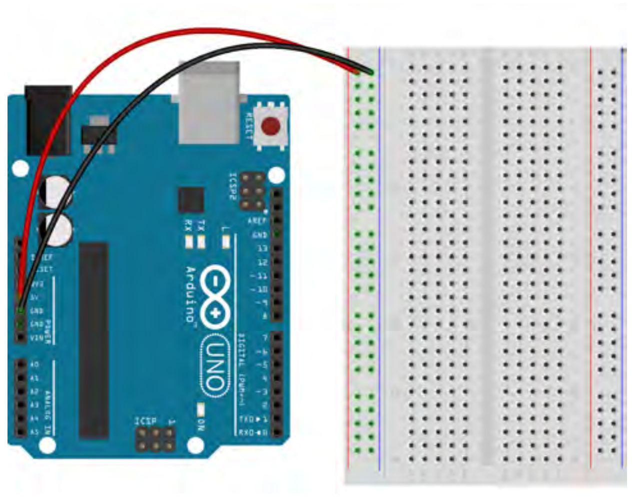


Figure 22: Spaceship Interface Circuit Diagram

Like the last project, the first step here is two connect two wires, one from the '5v' pin on the Arduino to the '+' column on the breadboard, and the other from the Arduino's 'GND' pin to the breadboard's '-' column.

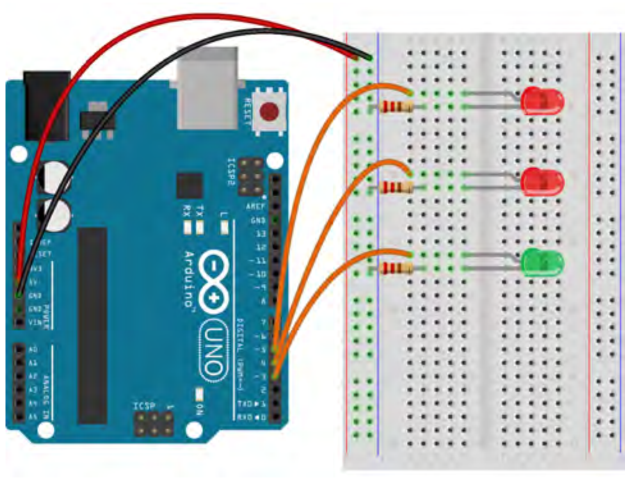


Step 1

Now it is time to plug the LEDs into the breadboard. Plug two red LEDs followed by a green LED into the breadboard so that none of the leads are sharing a row.

Plug one end of one Red-Red-Brown resistor into the same row as each of the negative (shorter) leads of the LEDs and plug the other end into the same '-' column that you have 'GND' plugged into.

Take three wires and plug one end of each into each row with an LED positive lead (the longer one that had a kink put in it). Plug the other end of the wire connected to the green LED's positive lead into the '3' pin on the right side (the side marked 'digital') of the Arduino. Plug the ends of the wires connected to the red LEDs into the '4' and '5' pins on the same side of the board.



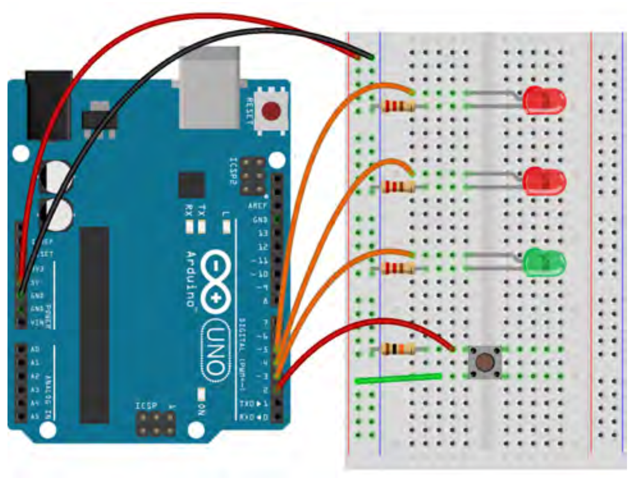
Step 2

Next, place the push-button switch so that it spans the trench in between the two grids. Two of the button's leads should be connected to rows in one grid and the other two lead should be connected to rows in the second grid. Make sure none of the push-button leads share a row with an LED lead. Only one side of the push-button is going to be used in this project, the side that is in the same grid as the LEDs, and the other half of the push-button can be ignored.

Take a Brown-Black-Orange resistor and plug one lead into the same '-' column and plug the other end into the same row as one of the leads of the push-button.

Take a wire and plug one end into the other push-button lead (the one that didn't just get a resistor plugged into it) and plug the other end into the same '+' column that is attached to the '5V' pin on the board.

Lastly, take a wire and plug one end into the same row where the push-button and the Brown-Black-Orange resistor are connected. Plug the other end into the '2' pin on the digital side of the Arduino.



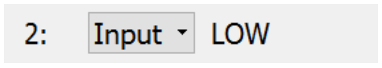
Step 3

Pin	Code	Input	Output
D2	2	✓	
D3	3		✓
D4	4		✓
D5	5		✓

Table 4: Spaceship Interface Project Pin Assignment Chart

6.5 ChDuino Basic Test

This project introduces digital inputs. These can be useful for any task requiring you to determine whether or not an action has occurred, such as pressing on a push button.



2: Input LOW

Figure 23: project 2

- Set the digital pin (D2) connected to the button to ”Input”. Press the push button and watch how the push-button pin reading (D2) changes from ”LOW” to ”HIGH”.
- Set the digital pins (D3-5) connected to the LED lights to ”Output” mode. Just like in Project 1, you can turn the LED lights on and off using the ”HIGH” and ”LOW” buttons.

6.6 Writing the Code

The following is the code for this project which will be explained step-by-step in this section:

```
// file: spaceshipInterface.ch
// Control LEDs with a push-button switch

#include <arduino.h>

//Declare a variable to keep track of the state of the push-button switch
int switchState = 0;

void setup() {
    //Set the LED pins to output mode
    pinMode(3, OUTPUT);
    pinMode(4, OUTPUT);
    pinMode(5, OUTPUT);
    //Set the switch pin to input mode
    pinMode(2, INPUT);
}

void loop() {
    //Read the state of the switch
    switchState = digitalRead(2);

    //If the button is not pushed turn on the green LED and turn off the red LEDs
    if (switchState == LOW) {
        digitalWrite(3, HIGH);
        digitalWrite(4, LOW);
        digitalWrite(5, LOW);
    }

    //If the button is pressed turn the green LED off and turn of the second red LED,
    //wait a quater second, turn the second red LED off and turn the first red LED on
    else {
        digitalWrite(3, LOW);
        digitalWrite(4, LOW);
        digitalWrite(5, HIGH);
        delay(250);

        digitalWrite(4, HIGH);
        digitalWrite(5, LOW);
        delay(250);
    }
}

int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

Like the last project, the code for this project starts by including the `arduino.h` header file.

```
int switchState = 0;
```

Next, the program creates an int variable called 'switchState'. This is shown in the code segment above. This variable will keep track of whether or not the push-button is pressed. Why is it an integer? Remember that the push-button is a digital input so 'switchState' will equal 1 when the button is pressed and will equal 0 when not pressed. If the variable is only going to be 0 or 1 then it is only required to be an integer. int variables take up less of the computer's memory than decimal numbers so having an integer variable instead of a decimal variable makes the program smaller and run faster.

```
void setup(){
   pinMode(3, OUTPUT);
   pinMode(4, OUTPUT);
   pinMode(5, OUTPUT);

   pinMode(2, INPUT);
}
```

For this project, pins 3-5, the pins connected to the LEDs, are output pins because the microcontroller has to send a signal to the LEDs to turn them on. Pin 2, the one connected to the push-button, is an input pin because the action the program takes depends on the signal created by pressing the button. The microcontroller needs to know which pins are inputs, meaning it needs to read a value from the pin, and which are outputs, meaning it needs to write a value to a pin. The code segment above assigns each pin as an input or output using the `pinMode()` function. Looking at the segment of code above, the first three lines set the LED pins, pins 3-5, to output mode and the last line sets the push-button pin, pin 2, to input mode.

```
void loop(){
```

Now that the pin modes are set, the program starts the definition for the `loop()` function so that the code inside of the `loop()` function will run continuously forever.

```
   switchState = digitalRead(2);
```

Now to start the code inside the `loop()`, meaning inside of the braces `{....}`. First, check the button to see if it is pressed. This is done in the code segment above by using the function `digitalRead()`.

```
int digitalRead(int pin)
```

The only argument of this function is an integer representing the number of the pin that is to be read. In this case the function is reading the pin connected to the push-button, pin 2. The function will return 1, or true, if the pin has a voltage of 2 volts or more and will return a 0, or false, if the pin has less than 2 volts. So, if the button is pressed, the current is allowed to flow through the button, creating a voltage difference that the microcontroller can read. The `digitalRead()` function returns a 1 if the voltage is high enough. The integer variable 'switchState' is set equal to the function so 'switchState' is set equal to 1 when the function returns 1. Thus when the button is pressed 'switchState' becomes equal to 1, and when the button is not pressed 'switchState' becomes equal to 0. 'switchState' is what is commonly referred to as a flag, or a variable that indicates whether something is true or not, whether an event has happened or not, or whether something is in one state or another.

```
   if (switchState == LOW) {
        digitalWrite(3, HIGH);
        digitalWrite(4, LOW);
        digitalWrite(5, LOW);
    }
```

The program now needs to tell the microcontroller what to do if the button is pressed or not. This is where the 'switchState' flag comes in handy. The program makes use of what is called an if-else statement. The code above shows the first part of this statement, the 'if' part. The if-statement works similar to the while-loop. It examines the condition or statement inside the parenthesis and **IF** it is true then the program will go through the code inside of the braces only one time, unlike the while-loop which can loop indefinitely. In this case the condition of the if-statement is `(switchState == LOW)`. Remember that the 'LOW' macro is defined as 0. The double equal sign is an operator that checks for equivalence. If there was only one equal sign then 'switchState' would actually be set to 0 rather than checking to see if it was equal to 0. So, the program will run the lines inside the braces if 'switchState' is equal to 0, or in other words it will run this code if the button has not been pressed. Macros are used again to clarify to the user what they are telling the function to do, where `HIGH` is defined to be equal to 1 and `LOW` is similarly defined to be 0. `HIGH` will set the pin to 5 volts and `LOW` will set the pin to 0 volts. So, the three lines inside of the if-statement above will turn pin 3 high and turn pins 4 and 5 low, meaning that the green LED will be turned on and the red LEDs will be turned off. Again this will only happen if the button is not pressed, making the 'switchState' flag equal to 0.

```
    else {
        digitalWrite(3, LOW);
        digitalWrite(4, LOW);
        digitalWrite(5, HIGH);
        delay(250);

        digitalWrite(4, HIGH);
        digitalWrite(5, LOW);
        delay(250);
    }
}
```

The segment of code above is the second half of the if-else statement started in the last segment. An if-else statement is a combination of an if-statement and an else-statement. An if-statement does not require an else-statement but an else-statement requires, and must follow, an if-statement. The program will run the else-statement when the condition of the if-statement is anything **ELSE** besides true. If the if-statement condition is true then the program will run the if-statement and not run the else-statement. So, in this case when the button is not pressed, when 'switchState' is equal to 0, the program runs the if-statement. If 'swtichState' is anything beside 0, like how it is 1 when the button is pressed, then the program will run the else-statement. Thus when the button is pressed, the program will run the code above inside the else-statement.

Like the if-statement earlier, the else-statement uses the `digitalWrite()` function to turn pins on or off. The first three lines inside of the else-statement turn off the green LED and the first red LED while turning on the second red LED. The program then uses the `delay()` function to pause the program for a quarter of a second before moving on. The lines after the first delay again make use of the `digitalWrite()` function to turn on the first red LED and turn off the second red LED. The last line inside of the else-statement pauses the program for another quarter of a second. Finally, the last brace closes the `loop()` function that was created way back.

```
int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

The program ends by calling the **`setup()`** function once and repeatedly calling the **`loop()`** function an infinite number of times using an infinite while-loop.

When the program is run, the green LED should light up until the button is pressed, at which point the green LED turns off, the first red LED turns on, and then after a small delay the second red LED will turn on and the first red LED will turn off. If errors are encountered or the program doesn't run exactly as desired do not worry, debugging, or the act of troubleshooting code, is part of the life of anyone who programs. Debugging is actually a beautiful thing, it forces the programmer to get into the nitty gritty of the code in order to understand how it works so they can then find out what went wrong. Most often errors are small, almost silly, mistakes, like forgetting a semi-colon at the end of a line or misspelling a function, but this quickly teaches the programmer all the little syntax quirks of a programming language. The more someone debugs, the better programmer they become.

6.7 Exercises

1. As it is, the code blinks the red LEDs only once. Modify the code so that the red LEDs blink multiple times, at least twice. Bonus points if this is accomplished using a loop.
2. Modify the code to make the red LEDs blink faster, say every 125 milliseconds.
3. Modify the code so that the roles of the different colored LEDs are reversed. Make the red LEDs the ones that are on when the button is not pressed and the green LED the one that turns on and blinks when the button is pressed.
4. Try using different colored resistors. What effect does this have on the LEDs? Why might this be? Use the resistor color chart in Section 21.2 and the explanation of resistors in Section 1.2.
5. Modify the circuit so that the button turns on an LED when pressed without using any code.

7 Reading an Analog Temperature Sensor

7.1 Project 3: Love-O-Meter

Project Description: In this second project, instead of controlling a series of LEDs with a button they will be controlled with a temperature sensor. There will be three LEDs, just like the last project, expect they will be turned on in sequence as the temperature increases, creating a thermometer of sorts.

7.2 New Concepts

Temperature sensors, like most types of sensors, are analog, not digital. Remember that, where digital means there can only be two options, on and off, analog means that the input or output can be any voltage within certain range. The temperature sensor works because its electrical resistance, and its voltage, changes with changing temperature. So, the temperature sensor will have a voltage within a known range and based on that voltage, the temperature can be calculated. This program will also introduce a new type of loop, called a for-loop as well a the '`<=`', '`>=`', and '`&&`' operators.

New Functions

- `int analogRead(int pin)`
- `void Serial.begin(unsigned long baud)`
- `void Serial.print(...)`
- `void Serial.println(...)`

7.3 Required Components and Materials

- Breadboard
- 1 TMP36 Temperature Sensor
- 3 220Ω (R-R-Br) Resistor
- 3 LED

7.4 Building the Circuit

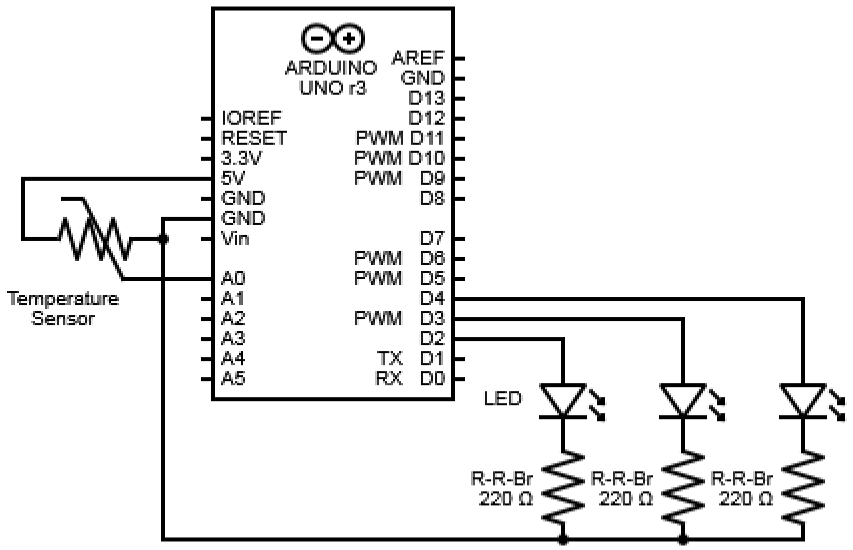
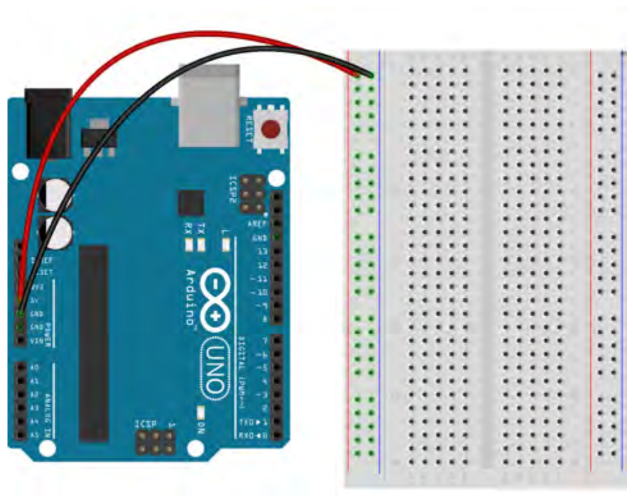


Figure 24: Love-O-Meter Circuit Diagram

First, take a wire and connect the '5V' pin on the Arduino to the column marked '+' on the breadboard. Do the same for the 'GND' pin and the '-' column.

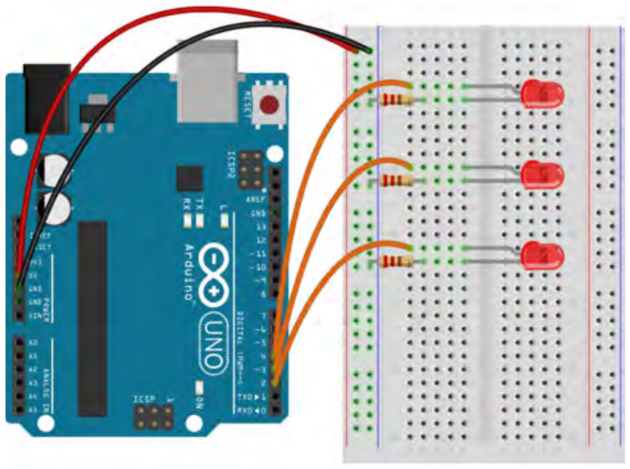


Step 1

The LEDs in this circuit are going to be wired almost exactly the same as in the last project. Plug the LEDs into the breadboard so that each LED's leads are in different rows of the breadboard and none of the leads of any LED share a row. Remember that the positive lead of the LED is the one that is slightly longer than the other. It is important to keep track of which lead is which so, if not done already, add a kink in the positive lead to mark which one is which.

For each LED, take a Red-Red-Brown resistor and plug one of the resistor's leads into the same row as the negative lead of the LED. Take the other end of the resistor and plug in into the column marked with a '-', or the ground strip.

To connect the LEDs to the Arduino take a wire for each LED, plug one end into the same breadboard row as the LED's positive lead and plug the other end of the wires into digital pins 2-4 on the Arduino. Make sure that the order the LED are placed on the breadboard matches the order they are plugged into pins 2-4 (this is not necessary but makes things easier while coding).

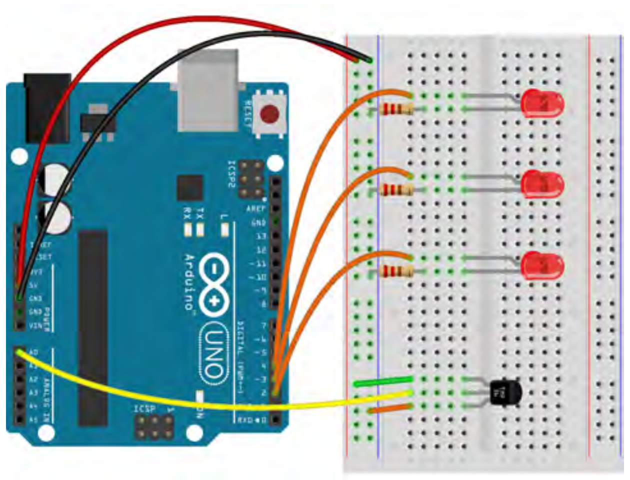


Step 2

The next step is to set up the temperature sensor. The temperature sensor looks like a little, black, cylinder that has been sliced off-center with three leads sticking out of one end. On the flat side, in very small lettering, it should say "TMP". If you look at the sensor so that the leads are pointing towards the ground and the flat surface is facing you then the left lead is the positive power lead, the middle lead is the lead that changes voltages and outputs the data that the program will use, and the right lead is the negative power lead. Plug the temperature sensor into the breadboard so that the three leads are in different rows. Make sure that the temperature sensor leads do not share any rows with LED leads. Make sure that you have the temperature sensor leads connected in the right orientation because reversing the leads can cause the sensor to heat up and get damaged.

Take a wire and plug one end into the same row as the positive power lead of the temperature sensor, as described above, and plug the other end into the column marked '+', or the positive power strip. Do the same for the negative power lead of the temperature sensor expect plug the other end of the wire into the ground column of the breadboard.

Lastly, take a wire and plug one end into the data lead, the remaining lead, of the temperature sensor. Plug the other end into the Arduino pin marked 'A0' in the section marked 'Analog In'. The circuit is now ready to be programmed.



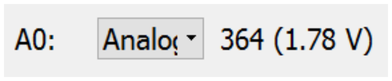
Step 3

Pin	Code	Input	Output
A0	0	✓	
D2	2		✓
D3	3		✓
D4	4		✓

Table 5: Love-O-Meter Project Pin Assignment Chart

7.5 ChDuino Basic Test

The last project introduced digital input. This project will introduce the concept of Analog input. Analog values read voltages and convert that value to an integer between 0 and 1023. In the GUI, you can see both numbers. The analog values from the GUI will always be integers between 0 and 1023. Using the code in the next section, the Arduino can convert these analog values to real-world temperatures.



A0: Analo 364 (1.78 V)

Figure 25: project 3

7.6 Writing the Code

The following is the code for this project:

```
// file: loveOMeter.ch
// turn on LEDs based on the temperature measured by a temperature sensor

#include <arduino.h>
```

```
//Declare pin assignment, sensor, and baseline variables
const int sensorPin = A0;

int sensorVal,
    pinNumber;

const double baselineTemp = 20.0;

double voltage,
    temperature;

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

    //Set pins 2-4 as outputs and set them to 0 volts
    for (pinNumber=2; pinNumber<5; pinNumber++) {
        pinMode(pinNumber, OUTPUT);
        digitalWrite(pinNumber, LOW);
    }
}

void loop(){
    //Read the value from the sensor pin then transform it into the appropriate voltage
    //between 0 and 5 volts and transform this to the corresponding temperature for
    //this sensor
    sensorVal = analogRead(sensorPin);

    Serial.print("Sensor Value: ");
    Serial.print(sensorVal);

    voltage = (sensorVal/1024.0)*5.0;

    Serial.print(", Voltage: ");
    Serial.print(voltage);

    temperature = (voltage - 0.5)*100;

    Serial.print(", degrees C: ");
    Serial.println(temperature);

    //If the new temperature is less than the user defined minimum temperature then all
    //the LEDs are off
    if (temperature < baselineTemp){
        digitalWrite(2, LOW);
        digitalWrite(3, LOW);
        digitalWrite(4, LOW);
    }

    //If the new temperature is within 2 degrees above the minimum temperature then only
    //the first LED is on
    else if (temperature >= baselineTemp && temperature < (baselineTemp+2)){
        digitalWrite(2, HIGH);
        digitalWrite(3, LOW);
        digitalWrite(4, LOW);
    }

    //If the new temperature is within 2 to 4 degrees above the minimum temperature then
    //the first two LEDs are on
    else if (temperature >= (baselineTemp+2) && temperature < (baselineTemp+4)){
        digitalWrite(2, HIGH);
        digitalWrite(3, HIGH);
        digitalWrite(4, LOW);
    }

    //If the new temperature is greater than 4 degrees above the minimum temperature
    //then all of the LEDs are on
    else {
        digitalWrite(2, HIGH);
        digitalWrite(3, HIGH);
        digitalWrite(4, HIGH);
    }
    delay(1000);
}

int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

This program begins the same way as the last two programs.

```
const int sensorPin = A0;

int sensorVal,
    pinNumber;

const double baselineTemp = 20.0;

double voltage,
    temperature;
```

After the header files are included and the Arduino is connected, it is time to declare the variables that will be used in this program. First is a variable called 'sensorPin' to remember the number of the pin that is connected to the temperature sensor. It is a variable of data type `const int`. The `int`, as seen before, makes 'sensorPin' an integer and the `const` is an additional modifier that makes the variable's value constant throughout the program. `const` protects the variable from being changed for whatever reason and messing up the program. While it is not absolutely necessary in this instance because the program does not get close to changing the value of 'sensorPin', it is always wise to put a `const` modifier on any variable that is not supposed to change, just in case.

In the next two lines two integers are declared. 'sensorVal' will record the raw value read from the temperature sensor and ’pinNumber’ will be used soon to set the pin mode of the LED pins. Notice that ’pinNumber’ is part of a different line than ’sensorVal’ and doesn’t have an `int` before it. This is because of the comma at the end of the line creating ’sensorVal’. Each variable does not need its own `int` or `double` to be declared. They can all be declared in the same line with the same `int` as long as they are separated by a comma. Remember that the semi-colon is the programs way of marking the end of a line. So, even though the two lines declaring ’sensorVal’ and ’pinNumber’ are visually two separate lines, because there is no semi-colon at the end of the first line then program will see the two lines as the same line and thus ’sensorVal’ and ’pinNumber’ are both declared by the single `int`. These two lines are equivalent to the single line `int sensorVal, pinNumber;`. Moving the ’pinNumber’ declaration to the next line simply make the program visually easier to read.

Now the program declares a variable called ’baselineTemp’ in order to set a minimum temperature at which the LEDs will start to light up. This book’s program has the baseline value at 20 (degrees Celsius) but this value can be changed depending upon the environment the user is working in. This value should not change throughout the program, hence the `const` modifier described earlier. This variable is a new type called a `double` that is part of a group of variable types called floating-point types. Floating-point types are how the computer deals with decimal numbers. So, the `double` type is a way of declaring a decimal number, note that the initial value of ’baselineTemp’ is `20.0` instead of `20`. The `double` type takes up twice as much memory space as an `int` type and can hold up to fifteen significant digits. Finally, the code segment creates two more variables of the `double` type. ’voltage’ and ’temperature’ will hold the voltage and temperature values calculated from ’sensorVal’, the raw values read from the temperature sensor.

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

The `setup()` function begin with something new, a function, called `begin()`. `begin()` is a member function of the `Serial` class. Classes are an important part of programming. They are special structures that contain special variables or functions, called member functions. So, `begin()` is a member function of the `Serial` class. Normally, the programmer would create an instance of a class like they would any other variable, which will be discussed in later sections, but the `Serial` class is pre-created inside of `@<arduino.h>@` so creating one is not necessary. The syntax to call a member function of a class is to type the name of the class, followed by a period, then the function. Thus the general form of the `begin()` function is shown below.

```
void Serial.begin(unsigned long baud)
```

The only argument is called ’baud’ and is an `unsigned long` type variable. A `long` variable is similar to an `int` but has more memory space and thus has the room to represent much bigger numbers. The `unsigned` modifier simply means that the variable cannot be negative. What the argument, ’baud’, represents is the Baud Rate which is how fast the information is transmitted to and from the Arduino and computer. 9600 is a standard value for a Baud Rate and is the default for arduino. This function has different purposes in the Ch and Arduino languages. In the Arduino IDE, `begin()` will tell the Arduino to start to communicate with the computer at the given Baud Rate. In Ch, this is accomplished automatically within the firmware and `@<arduino.h>@` header file, so the `begin()` function simply checks to see if the connection is working and the Baud Rate is correct.

```
    for (pinNumber=2; pinNumber<5; pinNumber++) {
        pinMode(pinNumber, OUTPUT);
        digitalWrite(pinNumber, LOW);
    }
}
```

The second part of the **setup()** function above sets the pin mode of all the LED pins to output and turns them all off initially. It does this by using what is called a for-loop. For-loops are the most visually confusing of all the loops. There are three arguments inside of the parenthesis, separated by semi-colons. The first is the initial condition, in this case it is **pinNumber=2**. The first time the program enters the for-loop it will set 'pinNumber' equal to 2. The second number is the exit condition, in this case it is **pinNumber<5**. Similar to a while-loop, the program will repeat the code inside of the for-loop until this condition is met. The third argument is the action the program is to take each time it goes through the for-loop, in this case it is **pinNumber++**. The '++' at the end of 'pinNumber' is a special operator that increases the value of the variable it is attached to by one. So, every time the program finished going through the for-loop it will add one to the value of 'pinNumber'. Thus, when the for-loop is of the form `for (pinNumber=2; pinNumber<5; pinNumber++)` the program will set the value of 'pinNumber' equal to 2 before the first time it enters the loop, then each time it goes through the loop it will add one to 'pinNumber' until 'pinNumber' is no longer less than 5 and it exits the loop. In the segment above, the first time the program goes through the for-loop it sets the pin mode and turns off pin 2 because 'pinNumber' is equal to 2. At the end of the loop the program increases the value of 'pinNumber' by one, making it 3, however, 'pinNumber' is less than 5 so the loop is run again. The next time the program goes through the loop it will set the mode and turn off pin 3 then increase 'pinNumber' to 4. Again, 'pinNumber' is still less than 5 so the loop is run again, but after running the loop again 'pinNumber' will be equal to 5, not less than 5, and the program will exit the for-loop. A for-loop is extremely useful when an action or segment of code needs to be run a specific number of times. For example, the code above allows the programmer to configure the LED pins without having to type the same code three times over. The programmer does not need to set the pin mode for the analog pin connected to the temperature sensor because all of the analog pins are assumed to be inputs only.

```
void loop(){
    sensorVal = analogRead(sensorPin);

    Serial.print("Sensor Value: ");
    Serial.print(sensorVal);
```

Now that everything is set up, the main body of the program can be written. Start by initiating the **loop()** function as discussed in the last project. The first thing to do once inside of the **loop()** function is read the data from the temperature sensor. The program does this by setting 'sensorVal' equal to the **analogRead()** function. The generic form of this function is shown below.

```
int analogRead(int pin)
```

This function has only one argument which is the pin number that the function is to read data from. This function will only read data from the analog pins on the Arduino and it will return a number between 0 and 1023.

During the process of testing and troubleshooting the project it would be helpful to have some visual feedback to know that the sensor and calculations are working. Enter the **print()** member function of the **Serial** class, shown below.

```
int Serial.print(...)
```

This function tells the program to display something, usually words or data, to the Input/Output Pane in ChIDE. This function is different in that its argument is ... meaning that it can take in, and print out, all different kinds of types of variables. Any strings need to be surrounded by quotation marks. So, the code segment above will print out the string, `"Sensor Value: "`, then print out the integer, 'sensorVal'. The **print()** function can print out many types of variables, but can only print them out one at a time. The print() function will return the number of bytes printed out.

```
    voltage = (sensorVal/1024.0)*5.0;

    Serial.print(", Voltage: ");
    Serial.print(voltage);

    temperature = (voltage - 0.5)*100;

    Serial.print(", degrees C: ");
    Serial.println(temperature);
```

The first line in the segment above transforms this raw value into the appropriate voltage and sets it equal to 'voltage'. It does this by dividing 'sensorVal' by `1024.0`, which is how large the range of possible values is, and this gives the ratio of how close 'sensorVal' is to its maximum possible value. The line then multiplies this ratio by `5.0`, the maximum possible voltage, to give the voltage that is the same percent of its maximum as 'sensorVal'. Remember 'voltage' is a `double`, which is a decimal number, so the right side of the equal sign should create a decimal number. However, in most programming languages, including Ch, if an integer is divided by an integer it creates an integer and any decimal part of the answer is ignored. This is why the denominator of the division is `1024.0` instead of `1024`. By having the decimal number `1024.0` in the denominator it ensures that the division will create another decimal number instead of an integer. The next line takes the voltage just calculated and uses it to calculate the temperature using an equation that is specific to the particular temperature sensor. Notice that the last printing statement is not `print()`, but `println()`. The `println()` function is the exact same as the `print()` function except that `println()` moves to a new line after it is done printing.

```
    if (temperature < baselineTemp){
      digitalWrite(2, LOW);
      digitalWrite(3, LOW);
      digitalWrite(4, LOW);
    }

    else if (temperature >= baselineTemp && temperature < (baselineTemp+2)){
      digitalWrite(2, HIGH);
      digitalWrite(3, LOW);
      digitalWrite(4, LOW);
    }

    else if (temperature >= (baselineTemp+2) && temperature < (baselineTemp+4)){
      digitalWrite(2, HIGH);
      digitalWrite(3, HIGH);
      digitalWrite(4, LOW);
    }

    else {
      digitalWrite(2, HIGH);
      digitalWrite(3, HIGH);
      digitalWrite(4, HIGH);
    }
    delay(1000);
  }
```

This is the final segment of this projects code and it is the part that gives different commands to the LED pins depending on the temperature value calculated earlier. It does this using an if-elseif-else statement, which is an extended version of the if-else statement discussed in the previous project. The first five lines form the initial if-statement. The condition of this if-statement is **`(temperature < baselineTemp)`** meaning that if 'temperature', calculated earlier, is less than "baselineTemp", defined by the user in the beginning, then the program will execute the three lines inside of the if-statement turning off all of the LEDs. However, if 'temperature' is greater or equal to 'baselineTemp' then the program will skip the if-statement and move on to the first elseif-statement. Elseif-statements are a way of extending if-else statements by adding more options and routes that the program could take rather than just the two, if and else.

The condition of the elseif-statement contains some new operators, the '>=' and the '&&'. '>=' is simply the greater than or equal to operator, similarly '<=' is the less than or equal to operator. The '&&' operator is called the 'logical AND'. For example, the first elseif-statement has the condition that 'temperature' is greater than or equal to 'baselineTemp' **AND** 'temperature' is less than 2 added to 'baselineTemp'. Both statements have to be true, meaning that the temperature has to be within two degrees above the baseline, in order for the program to run the statement, which turns on the first LED but leave the other two off. The second elseif-statement is nearly identical to the first one expect this time the condition is that the temperature be greater or equal to 2 degrees and less than 4 degrees above the baseline. The second elseif-statement, if it's condition is true, will turn on the first two LEDs and leave the third off. The else-statement finishes of the if-elseif-else statement. If 'temperature' is anything else, meaning if it is above 4 degrees from the baseline, then the program will run the statement, turning all of the LEDs on.

The second to last line runs the **`delay()`** function with an argument of 1000. Remember that the argument of the **`delay()`** function is in milliseconds, so that line will pause the program for 1000 milliseconds, or 1 second, every time the program goes through the **`loop()`** function. This delay allows time for the LEDs to turn on and off otherwise it would be happening so fast the human eye couldn't distinguish them changing. Finally, the last line closes the braces for the **`loop()`** function started earlier and the program is finished.

```
int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

The program ends by calling the **`setup()`** function once and repeatedly calling the **`loop()`** function an infinite number of times using an infinite while-loop.

If everything is hooked up and programmed correctly the LEDs should be off when the temperature is less than 20 degrees Celsius, the first LED should turn on when the temperature hits 20 degrees, the second LED should turn on when the temperature reaches 22 degrees, and after it reaches 24 degrees all of the LEDs should be turned on. It is easy to verify this because the new current temperature is printed out every second by the **`print()`** function.

7.7 Exercises

1. Change the baseline temperature and the temperature ranges inside of the else-if statements so that the code detects (breath?, ice cube?, AC?).

2. Go back to the previous project and use a for-loop to set the pin mode of the LEDs

3. Create another elseif-statement that blinks all of the LEDs when the temperature gets past 8 degrees above the baseline

8 Using a Potentiometer to Dim an LED

8.1 Project 4: Dimmer

Project Description: For the fourth project, a LED's brightness will be controlled by turning a potentiometer

8.2 New Concepts

Potentiometers are what people commonly refer to as knobs. They are variable resistors that become more or less resistant as they are turned. So, instead of resistance changing due to temperature or light, potentiometers change their resistance based on user input by turning. Changing an LEDs brightness means sending it different levels of voltage, or analog voltage. The Arduino doesn't have any pins dedicated for analog output, so it uses a method called PWM (Pulse Width Modulation) in order to have a digital pin output a range of voltages, thus changing the brightness of a connected LED.

New Functions

- `void analogWrite(int pin, int value)`

8.3 Required Components and Materials

- Breadboard
- 1 Potentiometer
- 1 220Ω (R-R-Br) Resistor
- 1 LED

8.4 Building the Circuit

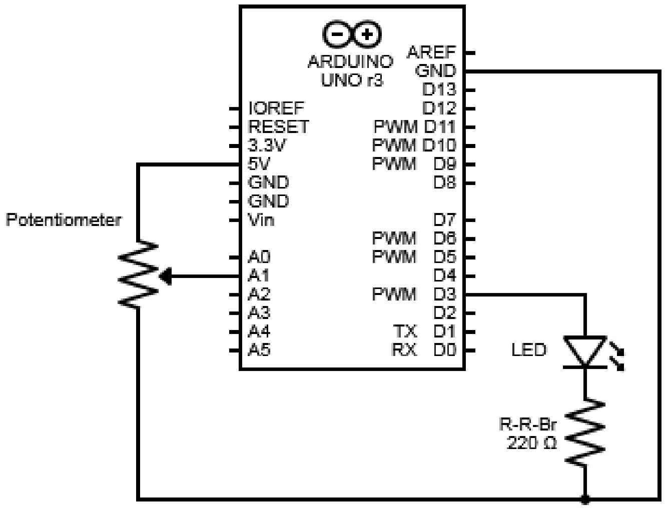
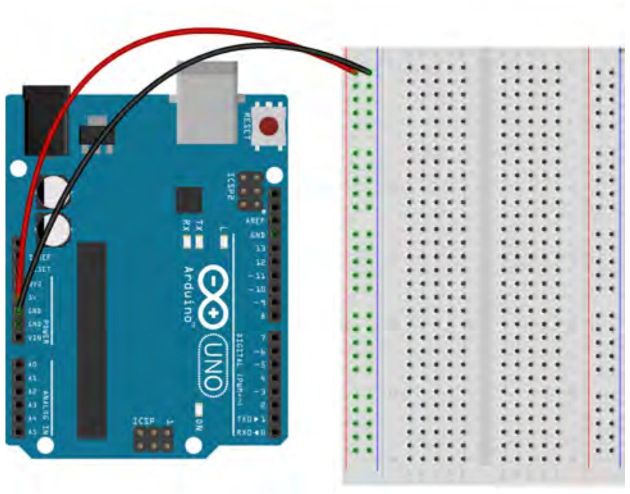


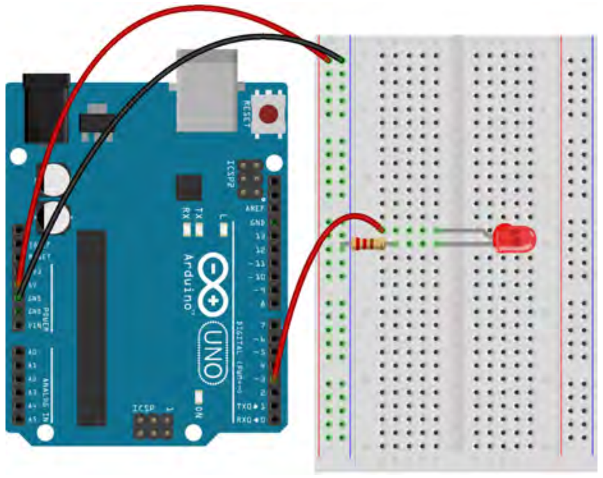
Figure 26: Dimmer Circuit Diagram

First, connect wires from the '+' column on the breadboard to the '5V' pin on the Arduino and from the '-' column to the 'GND' pin.



Step 1

Next, plug an LED into the breadboard so that the two leads are plugged into different rows. Take a wire and connect one end to the same row as the positive, longer, lead of the LED. Connect the other end of this wire to Digital Pin 3 of the Arduino. Grab a Red-Red-Brown resistor and plug one of the leads into the same row as the LED's negative, shorter, lead. The other end of the resistor should be plugged into the 'GND' column.

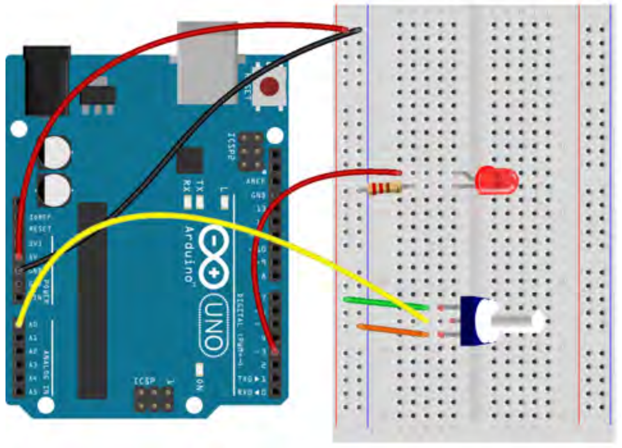


Step 2

The next step is to setup the potentiometer. There are two leads coming out of one side of the potentiometer and one lead coming out of the other. Place the potentiometer in such a way so that the leads are spanning the trench in the middle meaning that the side with two leads is on one side of the trench and the side with one lead is on the other.

On the side of the potentiometer with two leads, one of these leads needs to be connected to power and the other needs to be connected to ground, it does not matter which. Take a wire and connect one end to a row with one of the two lead and connect the other end to the '+' column. Take another wire and connect one end to the row with the other of the two potentiometer leads and plug the other end into the '-' column.

Lastly, for the single lead on the opposite side of the potentiometer, connect a wire between the row with this lead and the 'A0' pin on the Arduino.



Step 3

Pin	Code	Input	Output
A0	A0	✓	
D3	3		✓

Table 6: Dimmer Project Pin Assignment Chart

8.5 ChDuino Basic Test

In addition to having "HIGH" and "LOW" digital outputs, the Arduino can also produce a variable output from its digital pins using pulse-width modification (PWM). Pins that are capable of PWM will have a '~' symbol next to them.

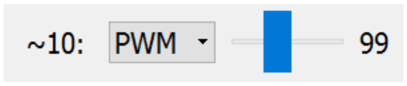


Figure 27: project 4

- To use PWM, set the digital pin connected to the LED (D3) to "PWM". By dragging the slider that appears, you can change the brightness of the LED.
- Similar to what you did in the last section, you can observe how twisting the potentiometer on your circuit outputs different analog values. 0 means that no current passes through the potentiometer. 1023 means that the potentiometer does not limit any current passing through. To familiarize yourself with the potentiometer, try picking a random number between 0 and 1023 and see if you can produce that number in the analog input reading.

8.6 Writing the Code

The following is the code for this project:

```
// file: dimmer.ch
// Dim the brightness of an LED using a potentiometer

#include <arduino.h>

// Declare pin assignment variables
const int LEDpin = 3,
          potPin = A0;

// Declare variables to hold values from the potemtiometer
// and hold values to be written to the LED
int potVal = 0,
    LEDval = 0;

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

    // Set the pin mode of the LED pin to output mode
    pinMode(LEDpin, OUTPUT);
}

void loop(){
    // Get the current voltage across the potentiometer
    potVal = analogRead(potPin);

    // Calculate the output value using the ratio of the
    // Input value to its maximum and print it out
    LEDval = potVal/1024.0*256;
    Serial.println(LEDval);

    // Write the output value to the LED pin
    analogWrite(LEDpin, LEDval);
}

int main(){
    setup();
    while(1) {
        loop();
    }

    return 0;
}
```

Once again, the program starts by including the header file.

```
const int LEDpin = 3,
        potPin = A0;

int potVal = 0,
    LEDval = 0;
```

The program begins declaring variable by creating two integers with the **const** modifier to hold the pin numbers connected to the LED and the potentiometer. Next, the program creates two regular integers. One is called 'potVal' and will hold the value that will be read from the potentiometer. The other is called 'LEDval' and will hold the value that will be written to the LED.

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

    pinMode(LEDpin, OUTPUT);
}
```

The `setup()` function is very simple, simply using the `pinMode()` function to set the mode of the LED pin to output mode.

```
void loop(){
    potVal = analogRead(potPin);

    LEDval = potVal/1024.0*256;
    Serial.println(LEDval);
    analogWrite(LEDpin, LEDval);
}
```

The `loop()` function begins by reading the current value of the potentiometer using the `analogRead()` function introduced in the last project. The next three lines have to do with writing an analog, meaning variable, voltage to the LED using one of the Arduino's digital pins even though the pins are digital and should only be either on or off. It is able to do this because of a process called Pulse Width Modulation (PWM). This is where the microcontroller turns the pin on and off very fast and the average voltage is what is seen by anything connected to the pin. So, if the program wants to send 2.5V out of a digital pin the microcontroller will turn the pin on, to 5V, half of the time and turn it off, to 0V, the other half of the time so that the average voltage is 2.5V. Before using PWM, the value read from the potentiometer needs to be transformed into the appropriate scale to be written. The 'potVal' variable is first divided by 1024, the size of the range of values that can be possibly read, giving a ratio. This ratio is then multiplied by 256, the size of the range of values that can be written, effectively transforming the read value from a 0-to-1023 scale to a 0-to-255 scale for writing. To use PWN, the `analogWrite()` function is utilized. The generic form is shown below.

```
void analogWrite(int pin, int value)
```

The first argument is an integer telling the function what pin to write to, and the second argument is the value that the function should write. The function will write values from 0 to 255 which is why the raw sensor value had to be transformed earlier. Note, the 0 to 255 range is just a scale and does represent actual voltage. The voltage created by the function will be the same percent of 5V as the value the user inputs is to 255.

```
int main(){
   setup();
    while(1) {
        loop();
    }

    return 0;
}
```

The program ends by calling the `setup()` function once and repeatedly calling the `loop()` function an infinite number of times using an infinite while-loop.

If everything if working as it should then the LED should become dimmer or brighter as you turn the potentiometer.

8.7 Exercises

1. Use the temperature sensor from the last project to make a hot/cold warning light.

9 Using Photo-Resistors to Change the Brightness and Color of an RGB LED

9.1 Project 5: Color Mixing Lamp

Project Description: For the fifth project, a RGB LED will be controlled by three photo-resistors that are set up to detect red, green, and blue light. The inputs of the RGB LED will each individually be controlled by a photo-resistor. The color of the RGB LED will be determined by the how much of each color of light hits the photo-resistors.

9.2 New Concepts

An RGB LED is like having three LEDs (red, green, and blue) inside of one LED. It has four leads, 3 inputs and a ground pin. Each of the three inputs controls one of the colors the LED creates. The amount of voltage on each pin controls the brightness that the RGB LED displays that particular color. A photo-resistor is a sensor that, like the temperature sensor, changes its resistance, except that a photo-resistor changes its resistance depending on the intensity of light that it is exposed to rather than temperature.

9.3 Required Components and Materials

- Breadboard
- 3 Photo-Resistor
- 3 220Ω (R-R-Br) Resistor
- 3 10kΩ (Br-Bl-O) Resistor
- 1 RGB LED

9.4 Building the Circuit

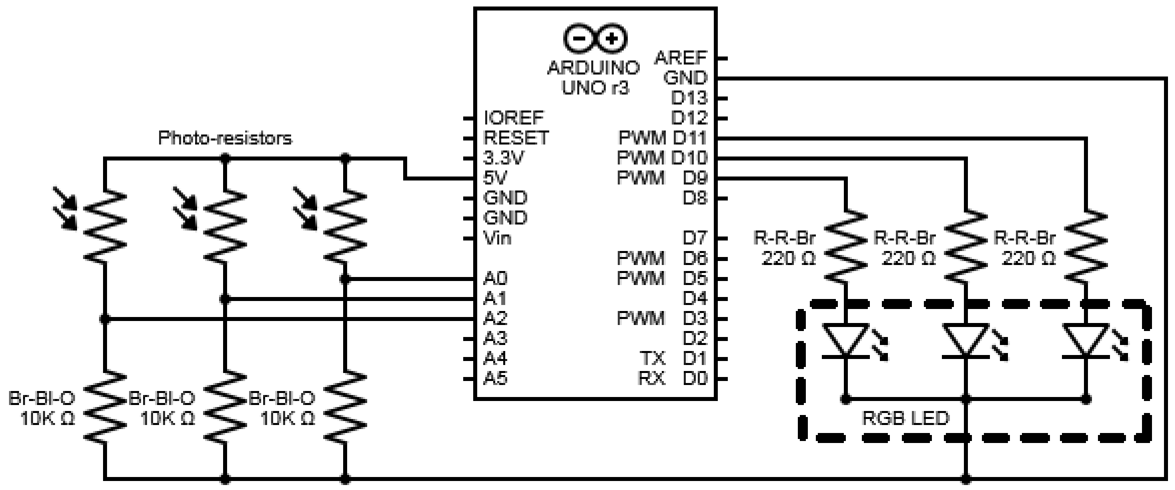
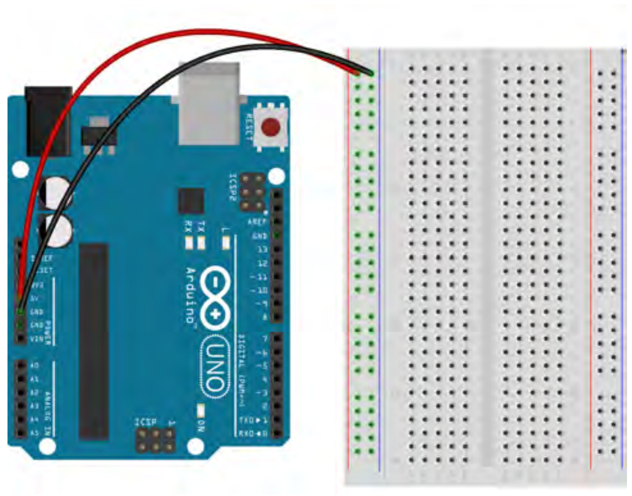


Figure 28: Color Mixing Lamp Circuit Diagram

First things first, connect wires from the '+' column on the breadboard to the '5V' pin on the Arduino and from the '-' column to the 'GND' pin.



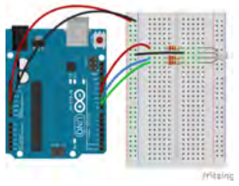
Step 1

Now plug in the RGB LED so that each of the four leads is plugged into its own row on the breadboard.

The longest lead on the RGB LED is the ground, or negative power, lead. Take a wire and plug one end into the row with the ground lead and plug the other end into the ground column, the one marked '-'.

Take three Red-Red-Brown resistors and plug in end of each into the separate rows containing the other three leads of the RGB LED. Plug the other end of the resistor into a row on the opposite side of the trench in the middle of the breadboard.

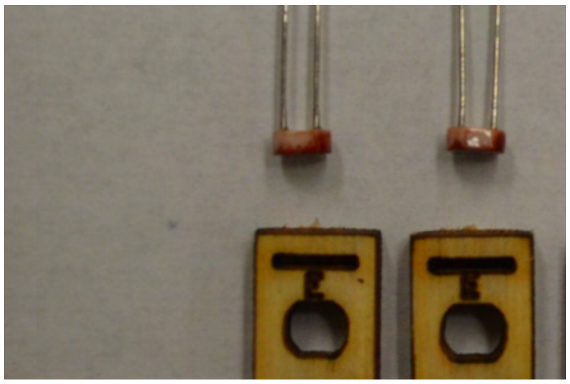
The leads on the RGB LED will be in a line so on one side of the ground lead there will be only one lead and on the other side of the ground lead should be two leads. The single lead is the pin that controls how much red the LED produces. The other two pins are the blue and green pins, going away from the ground pin. Take three wires and plug one end of each into the same rows as the leads of the three resistors that are away from the LED. The other ends of the wires should be plugged into the Arduino's digital pins so that the wire connected to the red lead of the LED is connected to digital pin 11, the wire connected to the blue lead is connected to digital pin 10, and the wire connected to the green lead is connected to digital pin 9. Notice that these three pins all have a '~' next to their pin number. This marks them as pins capable of PWM.

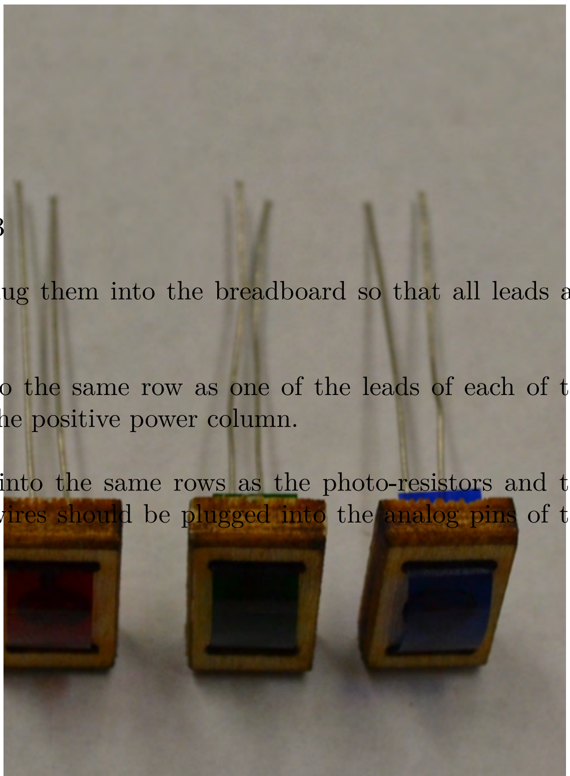


Step 2

The last part of the build is to set up the three photo-resistors. The photo-resistors look like a flat disk with leads coming out of one side and a squiggle on the other. When using the **Arduino Starter Kit** there should be three small wooden boxes with slots in the middle that match the shape of the photo-resistor.

Insert each photo-resistor into on of these slots and place one of the colored gel strips over the squiggle part of the photo-resistor. There should be three of those gels, red, green, and blue. The gels filter out all of the light except the light that is the color of the gel. So, the photo-resistor covered by the blue gel will only detect blue light.





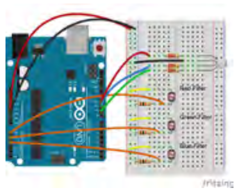
Step 3

Take the three, now covered, photo-resistors and plug them into the breadboard so that all leads are plugged into their own row.

Take three wires and plug one end of each into into the same row as one of the leads of each of the photo-resistors. Plug the other end of these wires into the positive power column.

Lastly, take three wires and plug one end of each into the same rows as the photo-resistors and the Brown-Black-Orange resistors. The other ends of the wires should be plugged into the analog pins of the

Arduino so that the red photo-resistor wire is connected to analog pin A2, the blue wire is connected to analog pin A1, and the green wire is connected to analog pin A0.



Step 4

Pin	**Code**	**Input**	**Output**
A0	A0	✓	
A1	A1	✓	
A2	A2	✓	
D9	9		✓
D10	10		✓
D11	11		✓

Table 7: Color Mixing Lamp Project Pin Assignment Chart

9.5 ChDuino Basic Test

This project is another variation on analog input and digital output, but with different components.

9.6 Writing the Code

The following is the code for this project:

```
// file: colorMixingLamp.ch
// control the brightness of the three colors of an RGB LED based on three
// photoresistors detecting different colors

#include <arduino.h>

//Declare the pin assignment variables
const int greenLEDpin = 9,
          blueLEDpin = 10,
          redLEDpin = 11,
          greenSensorPin = A0,
          blueSensorPin = A1,
          redSensorPin = A2;

//Declare variables to hold the raw sensor and transformed values for each LED
```

```
int greenVal,
    greenSensorVal,
    redVal,
    redSensorVal,
    blueVal,
    blueSensorVal;

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

    //Set the LED pins to output mode
    pinMode(greenLEDpin, OUTPUT);
    pinMode(redLEDpin, OUTPUT);
    pinMode(blueLEDpin, OUTPUT);
}

void loop(){
    //Read the input values from each sensor and print them out
    greenSensorVal = analogRead(greenSensorPin);
    delay(5);
    redSensorVal = analogRead(redSensorPin);
    delay(5);
    blueSensorVal = analogRead(blueSensorPin);

    Serial.print("Raw Sensor Values \t Red:");
    Serial.print(redSensorVal);
    Serial.print("\t Green:");
    Serial.print(greenSensorVal);
    Serial.print("\t Blue:");
    Serial.println(blueSensorVal);

    //Transform the sensor value from the 0-1023 scale of the inputs to the 0-255 scale
    //for output to the LEDs and print them out
    greenVal = greenSensorVal/4;
    redVal = redSensorVal/4;
    blueVal = blueSensorVal/4;

    Serial.print("Mapped Sensor Values \t Red:");
    Serial.print(redVal);
    Serial.print("\t Green:");
    Serial.print(greenVal);
    Serial.print("\t Blue:");
    Serial.println(blueVal);

    //Write the appropriate voltage based on the sensor input to each LED
    analogWrite(greenLEDpin, greenVal);
    analogWrite(redLEDpin, redVal);
    analogWrite(blueLEDpin, blueVal);
}

int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

Like the other programs, this one starts out by including the header files.

```
const int greenLEDpin = 9,
          blueLEDpin = 10,
          redLEDpin = 11,
          greenSensorPin = A0,
          blueSensorPin = A1,
          redSensorPin = A2;

int greenVal,
    greenSensorVal,
    redVal,
    redSensorVal,
    blueVal,
    blueSensorVal;
```

The next step is to declare all of the variables that will be needed in this program. First, the variables holding the pin numbers are declared. The pin numbers should reflect and match the way that the circuit was built. These variable are of type `const int` meaning that they are integers and the actual number cannot be changed for any reason by the program. Remember, the `const` modifier is not entirely necessary to make the program work but it is wise to have it as a precaution. All the other required variables are then created. For each of the three colors there is a '..Val' and a '..SensorVal' variable. The '..SensorVal' variables will hold the raw values read from each of the photo-resistors. The '..Val' variables will hold the result of transforming the '..SensorVal' variables that will be sent to the LEDs leads.

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

    pinMode(greenLEDpin, OUTPUT);
    pinMode(redLEDpin, OUTPUT);
    pinMode(blueLEDpin, OUTPUT);
}
```

The next step is to set the mode for the pins. The lines above use the `pinMode()` function to set each of the pins connected to the RGB LED to output mode. Notice the use of the pin number variables that were just created. For the programmer these kinds of variable make keeping track of the pins and pin numbers much easier.

```
void loop(){
    greenSensorVal = analogRead(greenSensorPin);
    delay(5);
    redSensorVal = analogRead(redSensorPin);
    delay(5);
    blueSensorVal = analogRead(blueSensorPin);
```

```
      Serial.print("Raw Sensor Values \t Red:");
      Serial.print(redSensorVal);
      Serial.print("\t Green:");
      Serial.print(greenSensorVal);
      Serial.print("\t Blue:");
      Serial.println(blueSensorVal);
```

The previous segment of code was the last bit of set-up for this program and now the body of the program can be written. Like the previous projects, the body of the code starts with the `loop()` function. Like the last project, this segment of code uses the `analogRead()` function to read data from analog sensors, like the temperature sensor but in this case a photo-resistor. The values read from the sensor are stored in the three '..SensorVal' variables created earlier. In between reading the photo-resistors, the program pauses the program for five milliseconds using the `delay()` function. This is because it takes a little time for the microcontroller to read the sensors and not having the delay could cause the functions to interfere with each other and return bad readings. Finally, the `print()` and `println()` functions is used to print out the data just read so the user can verify that the data makes sense and that nothing is wrong with the sensors. Notice that in the there is a `\t` inside the third and fifth print statements. This is a special character that tells the `print()` function to insert a tab space.

```
      greenVal = greenSensorVal/4;
      redVal = redSensorVal/4;
      blueVal = blueSensorVal/4;

      Serial.print("Mapped Sensor Values \t Red:");
      Serial.print(redVal);
      Serial.print("\t Green:");
      Serial.print(greenVal);
      Serial.print("\t Blue:");
      Serial.println(blueVal);
```

The values taken from the photo-resistors are on a 0-to-1023 scale, but in order to send the values to the RGB LED they need to be on a 0-to-255 scale. To transform the raw sensor values into the appropriate scale the program takes each of the three '..SensorVal' variables, divides it by four (because 1024 divided by 256 is 4), and sets this new transformed value equal to the corresponding '..Val' variable. The program then prints out these new values in an identical manner to the previous segment of code.

```
      analogWrite(greenLEDpin, greenVal);
      analogWrite(redLEDpin, redVal);
      analogWrite(blueLEDpin, blueVal);
   }
```

In the final segment of code the program sends the transformed sensor values to the RGB LEDs pins using the `analogWrite()` function introduced in the last project.

```
   int main(){
      setup();
      while(1) {
         loop();
      }
      return 0;
   }
```

The program ends by calling the **`setup()`** function once and repeatedly calling the **`loop()`** function an infinite number of times using an infinite while-loop.

If everything is correct the RGB LED should change color depending on how red, or blue, or green the surrounding light it. A quick way to test this is to shine a colored light on the photo-resistors. Any colored light should cause the RGB LED to change color. An even quicker way would be to temporarily remove one of the gels covering one of the photo-resistors and watch the RGB LED turn that color. Look at the text the program is printing out to make sure that the sensors are giving values that make sense and that the calculations are working. This project will also be the first time capacitors are used. They will have their leads on the power and ground of the servo which will smooth out voltage changes and generally allow the servo to function smoother.

9.7 Exercises

1. Instead of the RGB LED, use individual blue, green, and red LEDs
2. Instead of the photo-resistors, use a temperature sensor to control one of the RGB LED colors
3. Instead of the photo-resistors, use three buttons to control each of the RGB LED colors

10 Turning a Servo Motor with a Potentiometer

10.1 Project 6: Mood Cue

Project Description: Now that PWM and the `analogWrite()` function have been explained, larger and more fun devices than LEDs can be controlled. In this project the user will control a servo motor with a potentiometer.

10.2 New Concepts

Servo motors are used to control the angle of a mechanism very precisely. They consist of an electric motor and a sophisticated feedback system that tells the microcontroller what angle the servo is currently at.

New Functions

- `int map(int inputVal, int inputMin, int inputMax, int outputMin, int outputMax)`
- `void Servo.attach(int pin)`
- `void Servo.write(int angle)`

10.3 Required Components and Materials

- Breadboard
- 1 Potentiometer
- 2 100μF Capacitor
- 1 Servo

10.4 Building the Circuit

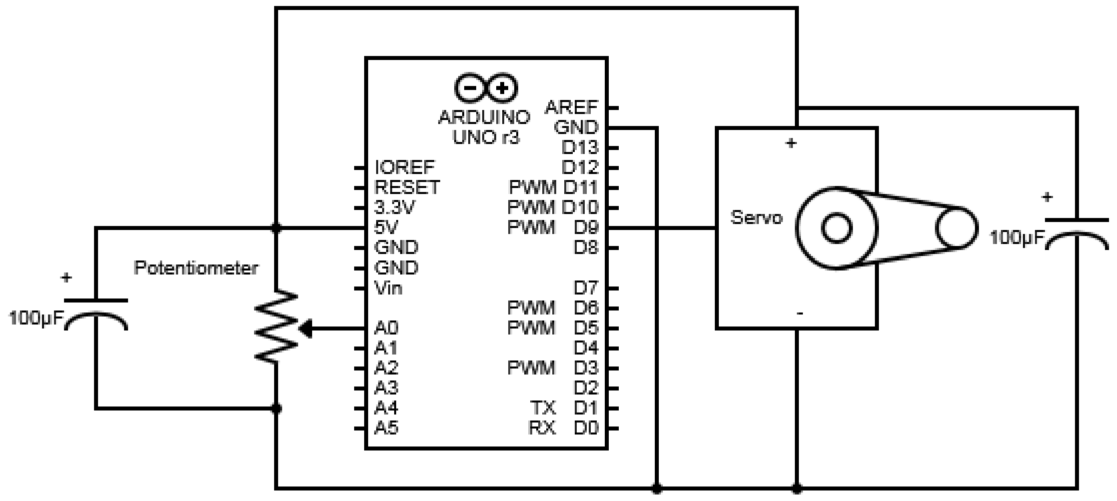
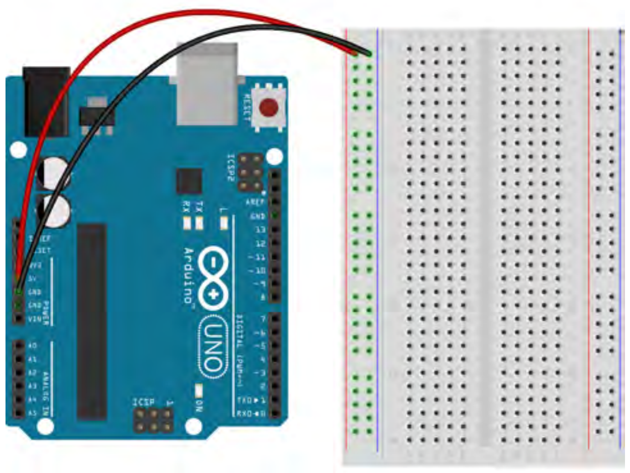


Figure 29: Mood Cue Circuit Diagram

As always, the first step is to plug in the power and ground wires.

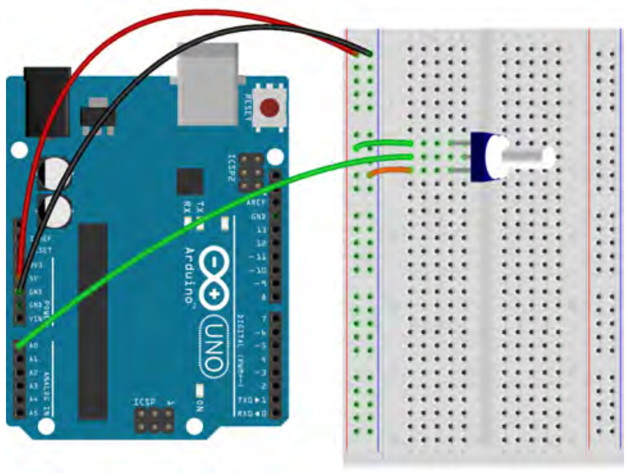


Step 1

Now to place the potentiometer. Due to the way the leads on the potentiometer are placed it needs to be plugged into the breadboard so that it spans the trench in the middle.

On one side of the potentiometer are two leads, these are the ground and power and should be connected with wires to the ground and power columns. It does not matter which of the two leads are connected to power or ground, the circuit will work either way.

On the other side of the potentiometer is a single lead. This is the lead that gives the program data. Plug a wire into the row containing this lead and plug the other end into the analog pin 'A0' on the analog side of the Arduino.



Step 2

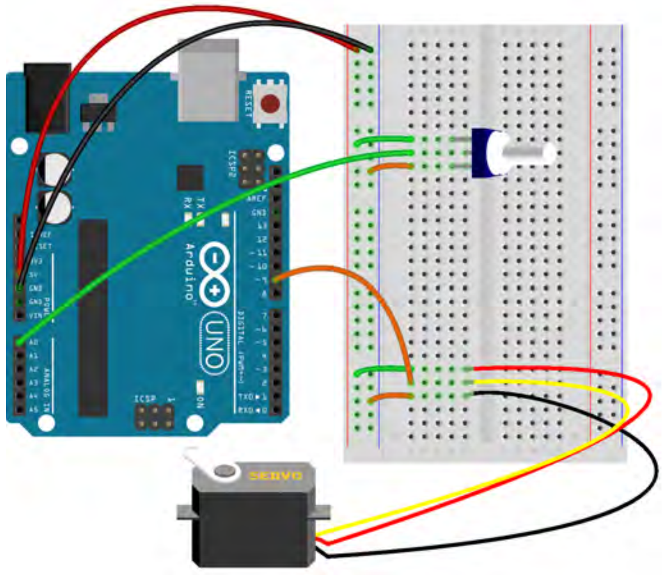
Next is the servo. The servo should have three wires connected to it, a red, black, and white wire. These wires end in a socket, that cannot plug into the breadboard, so break off three of the header pins that came with the kit and plug one side into the servo wires.

Step 3

The servo can now be plugged into the breadboard so plug it in so that none of the leads share a row with anything else.

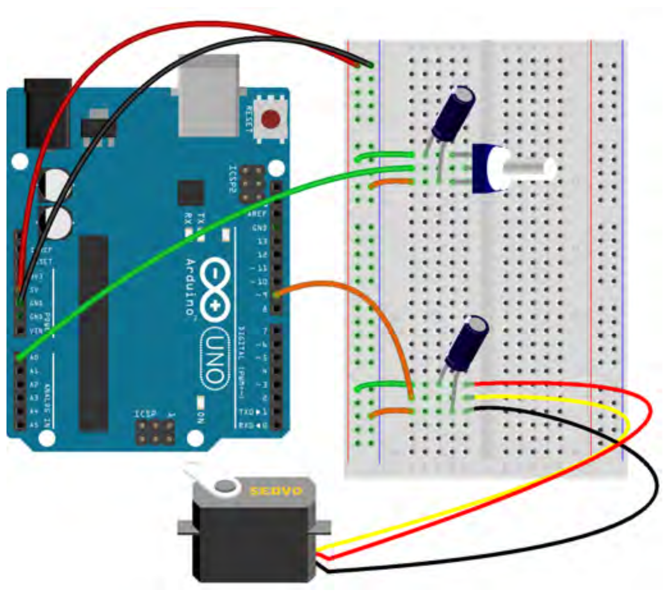
The red and black wires coming out of the servo are power and ground respectively and should be connected with wires to the power and ground columns of the breadboard.

The white servo wire is the lead used to get data from, or send commands to, the servo. Plug a wire into the same row that the white servo wire is plugged into, and plug the other end into digital pin 9 on the Arduino.



Step 4

The last thing is to plug in the decoupling capacitors across the potentiometer and servo. A decoupling capacitor will smooth out quick voltage changes across the potentiometer and servo and altogether make the movement of the servo smoother. The capacitors in the kit are called electrolytic capacitors and they are polar, meaning that one lead is for positive, or larger voltage, and the other lead is for the negative, or lower voltage. There will be a black strip along one side, near one of the leads of the capacitor and this marks the negative lead. For one of the capacitors, plug the negative lead into the breadboard row that is connected to both the potentiometer and to ground. Plug the other lead into the row that is connected to the potentiometer and to the power column. Repeat this with another capacitor and the servo motor leads. Make sure the capacitor lead next to the black strip is connected to ground and opposite lead is connected to power, because if it is plugged in backwards the capacitor could blow up. Now, that does sound kinda cool, but it is a really, really, bad idea. Think of the teacher. They're just trying to teach and if something blows up then they have to fill out all kinds of paperwork, get yelled at by parents, and maybe get fired. Plus nobody would get to use any of this equipment again. So, don't blow anything up on purpose. It's really not cool and would ruin everybody's fun. Don't be a fun ruiner.



Step 5

Pin	Code	Input	Output
A0	A0	✓	
D9	9		✓

Table 8: Mood Cue Project Pin Assignment Chart

10.5 ChDuino Basic Test

Project 6 introduces the servomotor. Although advanced motor movement requires writing code in the ChIDE, the GUI will allow you to perform basic motor movements.

10.6 Writing the Code

The following is the code for this project:

```
// file: moodCue.ch
// control a servo motor with a potentiometer

#include <arduino.h>

//Declare pin assignment variable
const int potPin = A0,
          servoPin = 9;

//Declare variables for the potentiometer value, current angle, and the value to be
//written to the servo
int potVal,
    angle;

//Declare an instance of the Servo class, called myServo
Servo myServo;

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

    //Tell the program that there is a servo on this pin
    myServo.attach(servoPin);
}

void loop(){
    //Read the input values from the potentiometer
    potVal = analogRead(potPin);

    Serial.print("potVal: ");
    Serial.print(potVal);

    //Transform the read value to the 0 to 179 range for the current angle, and the
    //0 to 255 range for the servo
    angle = map(potVal, 0, 1023, 0, 179);

    Serial.print(", angle: ");
    Serial.println(angle);

    //Write the transformed value to the servo to make it turn to a certain point
    myServo.write(angle);
    delay(200);
}

int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

Once again, this program begins with including the necessary header files.

```
const int potPin = A0,
        servoPin = 9;

int potVal,
    angle;

Servo myServo;
```

Declare two `const int` variables, one for the pin connected to the potentiometer called ’potPin’, and one for the pin connected to the servo called ’servoPin’. Next create two `int` variables. ’potVal’ will record the value read from the potentiometer and 'angle' will be the angle that the servo should turn to. Finally, create an instance of the `Servo` class by declaring a `Servo` type variable called 'myServo'. The servo class contains all of the special functions that pertain to using servos.

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

    myServo.attach(servoPin);
}
```

The final bit of setup is to a member function of the `Servo` class, called **`attach()`**, to let the Arduino know that there is a servo attached to this pin. The general form of the **`attach()`** function is shown below.

```
void Servo.attach(int pin)
```

The only argument is the an integer representing the pin number that the servo is connected to. Notice that in the code the **`attach()`** function is preceded by the 'myServo', the name of the `Servo` type variable created earlier. Typically, when a programmer wants to use a member function of a class they will create an instance of that class, then the name of that instance precedes the function name in the manner above. In this case it indicates to the code that the **`attach()`** function applies to the that particular servo, which becomes very necessary when there are more than one servo.

Remember that analog pins can only be inputs so there is no need to set the mode of 'potPin'.

```
void loop(){
    potVal = analogRead(potPin);

    Serial.print("potVal: ");
    Serial.print(potVal);
```

Set up the **`loop()`** function. The first thing to do inside the **`loop()`** function is read the data from the potentiometer pin with the **`analogRead()`** and assign it to the 'potVal' variable. The program then prints out the read value.

```
    angle = map(potVal, 0, 1023, 0, 179);
        servoOutVal);

    Serial.print(", angle: ");
    Serial.println(angle);
```

Remember that the values read from the analog pins are on a 0 to 1023 scale which means they need to be transformed to the 0 to 255 scale to be written to the servo. In previous projects, the calculation was done in the actual program, but this project uses the **`map()`** function, shown below, to do all the dirty calculation work.

```
void map(int inputVal, int inputMin, int inputMax, int outputMin, int outputMax)
```

The first argument is the value that needs to be transformed, in this case 'potVal'. The second and third arguments are the minimum and maximum of the scale that the first argument is already in, in this case 0 to 1023. The fourth and fifth arguments are the minimum and maximum values of the scale the first argument is to be transformed into. This servo will only turn 180 degrees, so, to transform 'potVal' into the 'angle' it will make the servo turn to, the fourth and fifth arguments of the `map()` function are 0 to 179. 'servoOutVal', or the value that will be written to the servo, should be on a 0 to 255 scale so the fourth and fifth arguments of the second `map()` function are 0 and 255. Then the `print()` function will print out the three values so that the user can verify that they make sense and the servo is responding how it is supposed to.

```
    myServo.write(angle);
    delay(200);
}
```

The program then uses a new member function of the `Servo` class, `write()`, to move the servo. The `write()` function uses PWM to send varying voltages to the servo which correspond to different angles of rotation. The generic form of this function is shown below.

```
void Servo.write(int angle)
```

The only argument for this function is an integer which represents the desired angle, in degrees, for the servo to turn to.

The program then pauses for 200 milliseconds to give the servo time to move before the `loop()` function is repeated. Finally, the last line closes the `loop()` function and the program is finished.

```
int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

The program ends by calling the `setup()` function once and repeatedly calling the `loop()` function an infinite number of times using an infinite while-loop.

If everything is working correctly then the servo should turn all the way in one direction if the potentiometer is turned all the way in one direction, and turn all the way in the other direction when the potentiometer is turned all the way in the other direction. any number of cool things can be attached to the servo that can now be turned at will.

10.7 Exercises

1. Use a photo-resistor to control the motor instead of the potentiometer
2. Translate the value read from the potentiometer into a value that can be written to the servo without using the `map()` function

11 Playing Notes with a Piezo and Photo-resistor

11.1 Project 7: Light Theremin

Project Description: In this project a photo-resistor will be used to control the frequency of a note played by a piezo, making an instrument that is played by light. The user will be able to control the notes the piezo plays by using their hands to cover the photo-resistor, controlling the amount of light that reaches it.

11.2 New Concepts

The main function of microcontrollers is to manage inputs and outputs, and now that the basics of how to acquire inputs and program outputs using Ch have been covered everything forward is expanding upon this idea using different input and outputs and combinations thereof. For instance, this program will use a photo-resistor input, which has already been used in this book, in conjunction with the piezo provided inside the **Arduino Starter Kit**, which has not been used so far. This program will also introduce the idea of calibration, because, as may have been noticed in previous project, sensors don't usually directly output the variable they are supposed to measure and some conversions are necessary. There more input and output devices a user is comfortable with, the more options they have when engineering a project.

New Functions

- `int millis(void)`
- `void tone(int pin, int frequency, unsigned long time)`
  or `void tone(int pin, int frequency)`

11.3 Required Components and Materials

- Breadboard
- 1 Photo-Resistor
- 1 10kΩ (Br-Bl-O) Resistor
- 1 Piezo

11.4 Building the Circuit

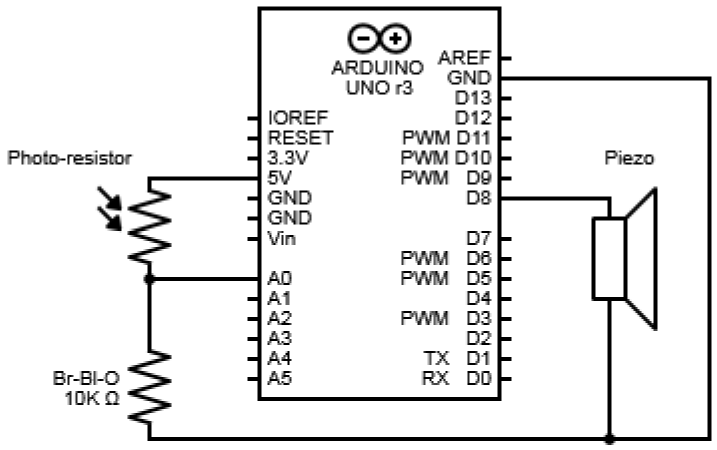
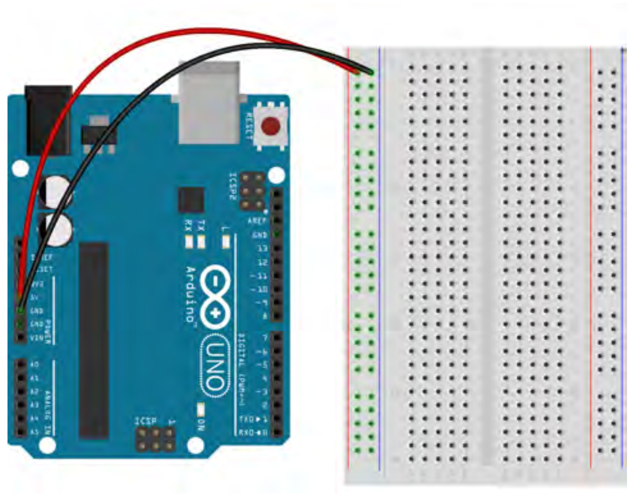


Figure 30: Light Theremin Circuit Diagram

The wiring of the photo-resistor will be essentially identical to the way they were wired in the Color Mixing Lamp project. As usual, the first step to the build is to plug in the power wires, one from '5V' and 'GND' on the Arduino to the '+' and '-' columns on the breadboard respectively.



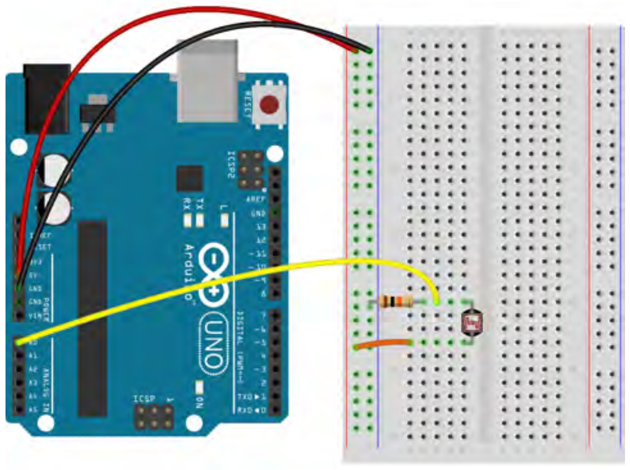
Step 1

Next, plug in the photo-resistor into the breadboard so that the two leads are in separate rows.

Take a Brown-Black-Orange resistor and plug one end into the same row as one of the leads of the photo-resistor. Plug the other end into the ground column.

Grab another wire and plug one end into the row with the other photo-resistor lead, plugging the other end of the wire into the positive power column.

Take one more wire a plug one end of it into the same row as the resistor and photo-resistor. Connect the other end to the analog pin ’A0’ on the Arduino.

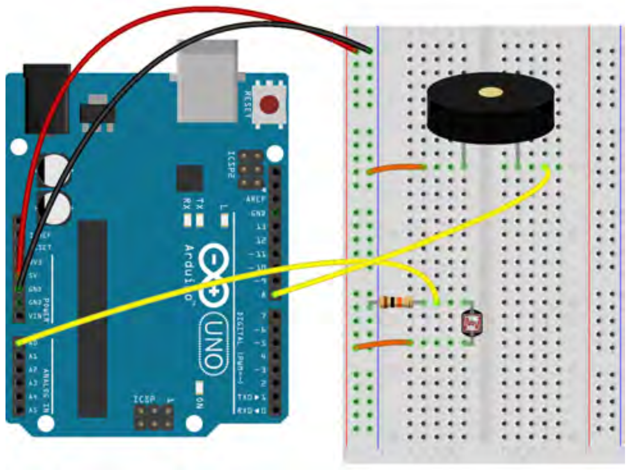


Step 2

Now to place the piezo buzzer. Plug the piezo into the breadboard so that the leads have their own rows.

Take a wire and plug one end into one of the rows with a piezo lead, it does not matter which one. Plug the other end into the ground column of the breadboard.

Lastly, take a wire and connect the row with a piezo lead, the one not used in the last step, to digital pin 8.



Step 3

Pin	Code	Input	Output
A0	A0	✓	
D8	8		✓

Table 9: Light Theremin Project Pin Assignment Chart

11.5 ChDuino Basic Test

Project 7 shows how pulse-width modification can be used to create many different sounds with the piezo. It also uses the photoresistor introduced in a previous project.

11.6 Writing the Code

The following is the code for this project:

```
// file: lightTheremin.ch
// Control the pitch of a buzzer using a photoresistor

#include <arduino.h>

//Declare pin assignment varible
const int sensorPin = A0;

//Declare variables for the sensor values, calibration time, and buzzer pitch
int calibTime = 5000,
    sensorHigh = 0,
    sensorLow = 1023,
    sensorValue,
    pitch ;

void setup(){
    Serial.begin(9600);
    //The calibration takes in values from the photoresistor for the set amount of time
    //If a value is higher than the previous high then it is set as the new high and the
    //opposite is done for the low values
    Serial.println("Calibrating..");
    while (millis() < calibTime) {
        sensorValue = analogRead(sensorPin);
        if (sensorValue > sensorHigh) {
            sensorHigh = sensorValue;
        }
        if (sensorValue < sensorLow) {
            sensorLow = sensorValue;
        }
    }
    Serial.println("Done Calibrating");
}

void loop(){
    //Read the value from the photoresistor
    sensorValue = analogRead(sensorPin);

    //Transform it to the 50 to 4000 scale for the buzzer
    pitch = map(sensorValue, sensorLow, sensorHigh, 50, 4000);

    tone(8, pitch, 20);
    delay(10);
}

int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

This program follows the same pattern by beginning with including the necessary header files.

```
const int sensorPin = A0;

int calibTime = 5000,
    sensorHigh = 0,
    sensorLow = 1023,
    sensorValue,
    pitch;
```

Only one pin variable is needed for the pin connected to the photo-resistor, so create a variable of type `int` with the `const` modifier called 'sensorPin'. Now the rest of the variables need to be created, and for the purposes of this project they are of the `int` type. One is needed to hold the raw value read from the photo-resistor, which is 'sensorValue'. Another should hold the value of the frequency of the note that the piezo will play, this is 'freq'. The other three variables have to do with the calibration process. 'calibTime' is the time that the calibration process will be allowed to run. This is in milliseconds for reasons that will be explained in a moment. The last two variables, 'sensorHigh' and 'sensorLow', will record the highest and lowest values that the photo-resistor returns during the calibration process. 'sensorHigh' is initialized to be equal to zero so that during calibration the value will have to go up. The opposite is true for 'sensorLow'.

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

    Serial.println("Calibrating..");
    while (millis() < calibTime) {
        sensorValue = analogRead(sensorPin);
        if (sensorValue > sensorHigh) {
            sensorHigh = sensorValue;
        }
        if (sensorValue < sensorLow) {
            sensorLow = sensorValue;
        }
    }
    Serial.println("Done Calibrating");
}
```

The segment of code above contains the calibration process. The first print statement prints out the string `"Calibrating.."` which lets the user know that the calibration process has begun. This is important because the calibration requires the users help. The second line starts a while-loop that contains the calibration process. The condition of the while-loop is `(millis() < calibTime)` and this is how the program controls how long the calibration process runs. The function `millis()` is shown below.

```
int millis(void)
```

It has no argument and returns the amount of time since the program started running in milliseconds. Therefore, the while-loop condition will be true while the time the program has been running is less than the calibration time variable, 'calibTime'. Of course, this is not perfectly accurate because it takes some time to include the headers and create the variables, but this is so small it can be considered unimportant for the purposes of this program. The first line inside of the `loop()` function uses the `analogRead()` function to read the value from the photo-resistor and assign that value to the 'sensorValue' variable. The program then uses an if-statement to see if the value just read from the sensor is greater than the current maximum value for the sensor represented by the 'sensorHigh' variable. If the read value is greater than the current maximum then the program enters the if-statement and sets 'sensorHigh' equal to 'sensorValue', or setting the read value as the new maximum value. The next line starts a similar, but opposite, if-statement to check if the read value is less than the current minimum, and if it is the program sets the read value as the new minimum. The user plays an important role in the calibration. To find the true minimum value, the user will need to cover the photo-resistor during part of the calibration. When the calibration process is done the program will print out the string `"Done Calibrating"`.

```
void loop(){
    sensorValue = analogRead(sensorPin);

    pitch = map(sensorValue, sensorLow, sensorHigh, 50, 4000);

    tone(8, pitch, 20);
    delay(10);
}
```

Now that the calibration is finished, all the preparation for the program is done and the body of the code is ready to be written. The first line in the code segment above starts the `loop()` function so that the program will run until the user stops it. The first action inside the `loop()` function is to read a new value from the pin connected to the photo-resistor, using the `analogRead()` function, and assigning the value to the 'sensorValue' variable. Similar to earlier projects, the values read from the sensor and the values that need to be written to the output are on different scales. In this case, the input from the photo-resistor is on a scale from 'sensorLow' to 'sensorHigh', and the output could be anywhere in the range of human hearing, from 20-20000 Hz, but for this project the scale is 50-4000 Hertz. This line maps the value just read from the photo-resistor from the 'sensorLow' to 'sensorHigh' scale into the 50 Hertz to 4000 Hertz scale. This mapped value is assigned to the 'pitch' variable. The next line uses the `tone()` function to control the piezo.

```
void tone(int pin, int frequency, unsigned long time)
```

The function has three arguments, the first is the number of the pin attached to the piezo, the second being an integer telling the function the frequency of the note to be played, and the third being a number telling the function how long to play the note in seconds. This last argument is actually optional, meaning that the function below is perfectly valid and will play the note as long as possible.

```
void tone(int pin, int frequency)
```

Finally, the second to last line of the program will print out the raw sensor value and the frequency calculated by mapping between the scales.

```
int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

The program ends by calling the `setup()` function once and repeatedly calling the `loop()` function an infinite number of times using an infinite while-loop.

When the program and circuit are working correctly, the user should be able to control how low or high the note played by the piezo is by moving their hand closer to or father from the photo-resistor.

11.7 Exercises

1. Adjust the range of frequencies in the fourth and fifth arguments of the `map()` function to make the sounds the piezo makes lower pitch.
2. Play around with the calibration time. Does calibrating for a longer or shorter time makes a difference?
3. Wire three LEDs so that one lights up when the frequency played is the lowest third of the frequency range, another lights up when the note is in the middle third of the range, and the last lights up when the note is in the highest third

12 Pressing Different Buttons to Play Different Notes on a Piezo

12.1 Project 8: Keyboard Instrument

Project Description: Like the last project, this one keeps in line with making electronic instruments by controlling the frequency of the notes the piezo plays with four push-buttons. Pressing different buttons will cause the piezo to produce different notes

12.2 New Concepts

This project will introduce the idea of arrays of variables. This program will also show how to use multiple push-button switches as a single analog input.

New Functions

- `void noTone(int pin)`

12.3 Required Components and Materials

- Breadboard
- 4 Push-button Switches
- 1 220Ω (R-R-Br) Resistor
- 2 10kΩ (Br-Bl-O) Resistor
- 1 1MΩ (Br-Bl-G) Resistor
- 1 Piezo

12.4 Building the Circuit

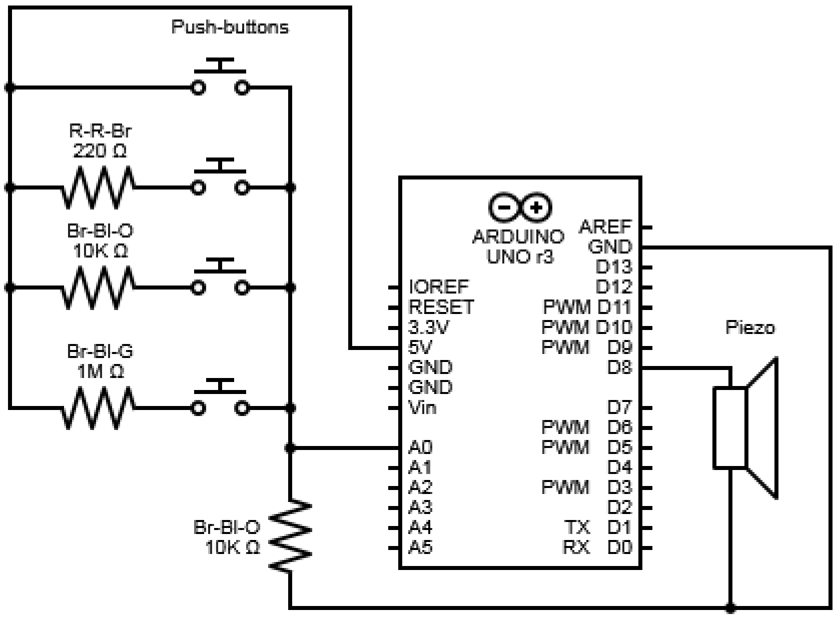
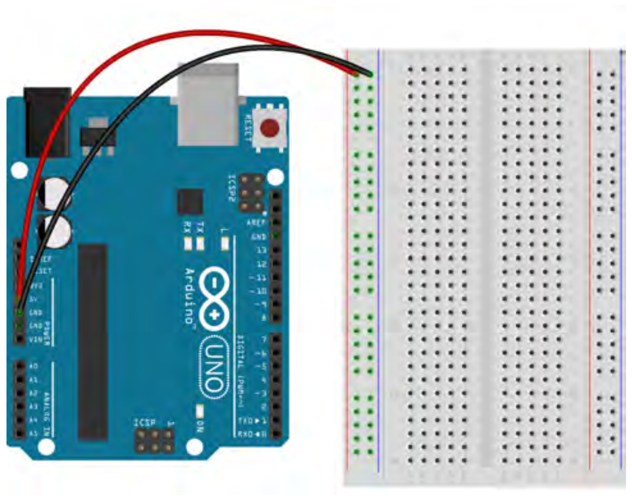


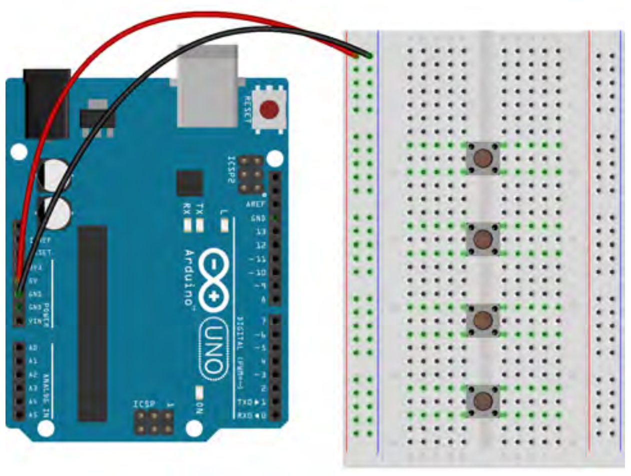
Figure 31: Keyboard Instrument Circuit Diagram

Once again, the first step is connecting the power and ground wires to the breadboard.



Step 1

This project will use four push-button switches. Take the four push-buttons and plug them into the breadboard so that each of them spans the trench dividing the two sides of the breadboard. There should be two leads from each button on one side of the trench and two on the other.

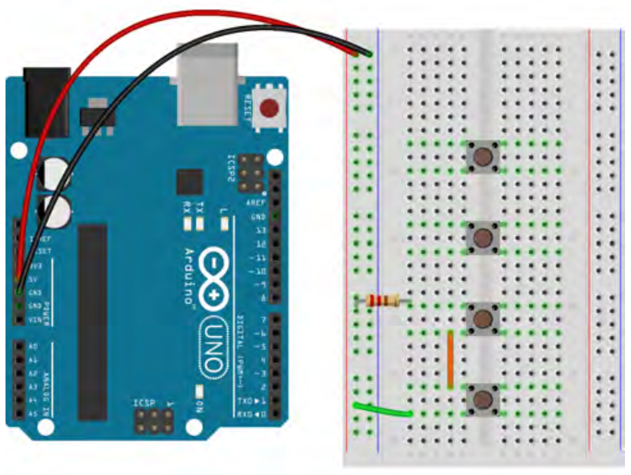


Step 2

The buttons will be in a rough line. Start at one end of this line. Take a wire and plug one end into the same row as a lead from the first button and put the other end in the power column.

Take another wire and plug it into the row with the other lead from the same button. Plug the other end of the wire into a row with a lead from the next button in the line.

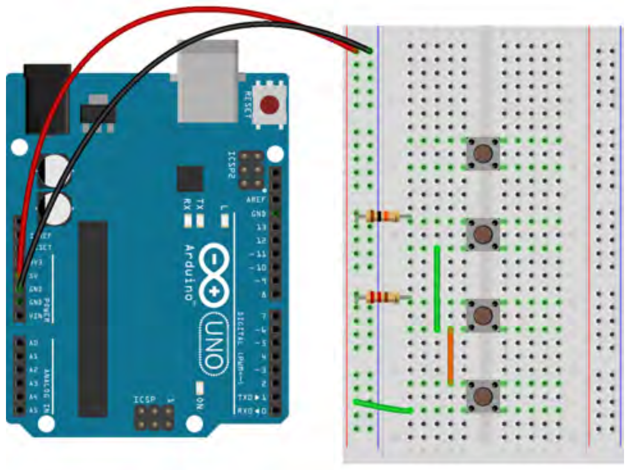
Grab a Red-Red-Brown resistor and insert one lead into the row with the other lead from the same button, the second button in the line. Plug the other end of the resistor into the power column.



Step 3

With another wire, plug one end into the row that has the lead from the second button and also the wire that connects the first and second buttons. Plug the other end into a row that contains a lead from the third button in the line.

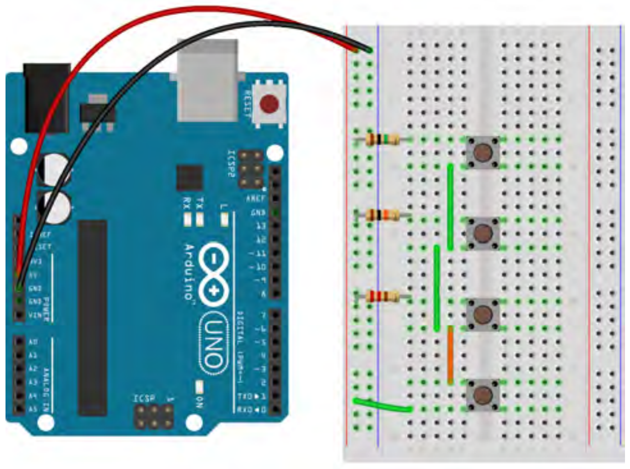
Connect the other lead, the one that did not just have a wire plugged into it, of the third button to the power column using a Brown-Black-Orange resistor.



Step 4

Now, on to the last button. Plug one end of a wire into the row with the lead from the third button and also the wire that connects the second and third buttons. The other end of that wire should be plugged into a row with one of the leads from the last button.

Connect the row with the other lead of the last button to the power column using a Brown-Black-Green resistor.

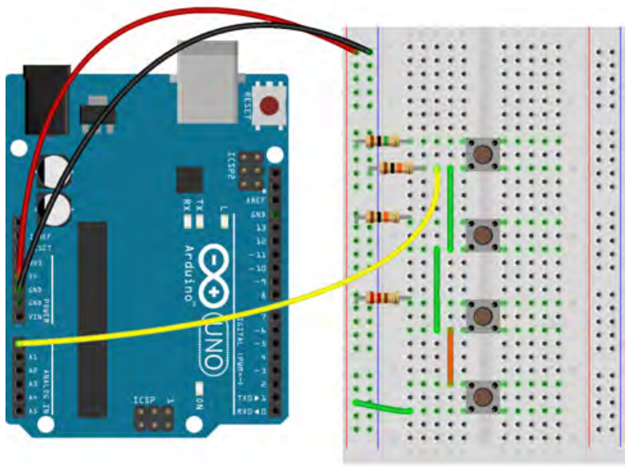


Step 5

Take another Brown-Black-Orange resistor and connect the row with a lead from the last button and also the wire connecting the third and last buttons to the ground column.

Use a wire to connect the analog pin 'A0' on the Arduino to the row that has the last button lead, the Brown-Black-Orange resistor, and the wire connecting the third and last buttons.

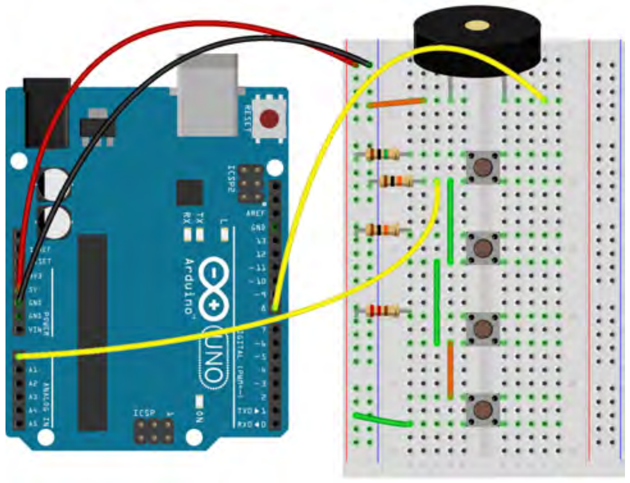
This complicated circuit wires each button and its corresponding resistor in parallel, meaning that they are each directly connected to the same input, the power column, and the same output, analog pin 'A0'. This, and the fact that the resistors all have different values of resistance, makes it so that when a button is pressed, a unique voltage is applied to pin 'A0'. So, when the program is looking at pin 'A0' it will see a different voltage depending upon what button, if any, is pressed and take different action depending on which voltage it sees.



Step 6

The last task is to wire the piezo. Plug the piezo's leads into the breadboard so that they have their own rows.

Connect one of the rows with a piezo lead to the ground column on the breadboard using a wire. Connect the other piezo lead row to digital pin 8 on the Arduino.



Step 7

Pin	Code	Input	Output
A0	A0	✓	
D8	8		✓

Table 10: Keyboard Instrument Project Pin Assignment Chart

12.5 ChDuino Basic Test

Circuits can use more than one push button, and can be set up so that pressing different combinations of buttons causes the circuit to give different results.

12.6 Writing the Code

The following is the code for this project:

```
// file: keyboardInstrument.ch
// play different frequencies over a buzzer by pressing different buttons

#include <arduino.h>

//Declare pin assignent variable
const int inputPin = A0;

//Declare an array to hold the different frequencies we want the buzzer to play
int notes[] = {262, 294, 330, 349};

//Declare a variable to hold the voltage value created by pressing a button
int keyVal;

void setup(){
    pinMode(8, OUTPUT);
    Serial.begin(9600);
}

void loop(){
    //Check the voltage on the input pin
    keyVal = analogRead(inputPin);
    Serial.println(keyVal);

    //Pressing different buttons gives different voltage values. When the voltage is
    //within a certain range it means a certain button was pressed and the corresponding
    //note is played over the buzzer for a duration of one second
    if (keyVal == 1023) {
        tone(8, notes[0]);
    }
    else if (keyVal >= 980 && keyVal <= 1010) {
        tone(8, notes[1]);
    }
    else if (keyVal >= 505 && keyVal <= 515) {
        tone(8, notes[2]);
    }
    else if (keyVal >= 5 && keyVal <= 10) {
        tone(8, notes[3]);
    }
    else {
        //noTone(8); // problem with Arduino code
    }
}

int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

This program begins by including the necessary headers.

```
const int inputPin = A0;

int notes[] = {262, 294, 330, 349};

int keyVal;
```

Since there is only one pin attached to the circuit, only one `const int` type variable needs to be created, called 'inputPin'. In the next line an array of integers is created, called 'notes'. An array is a group of variables of the same type and with the same name. Arrays are useful for holding groups of values or data that are used for the same purpose in the program, in this case the values are frequencies, because it would be annoying and messy to have to declare a unique variable for each of the values. To declare an array, add a left and right bracket after the name of the array. There are two basic ways to create an array, one where the values going into the array are already known, and one where they are not. This case is the former. The values inside of the braces, separated by commas, form each value of the array. The latter case is often used to create an array that will hold data read during the program and isn't known yet. This way of creating an array doesn't have any braces or values, but instead has a value inside of the brackets that follow the name of the array. The value inside of the brackets tells the program how many values, called elements, the array will hold. So, the second line above is equivalent to the code segment below. The values are assigned to the elements of the array individually

```
int notes[4];
notes[0] = 262;
notes[1] = 294;
notes[2] = 330;
notes[3] = 349;
```

instead of as a group. To access one element of an array, have the number of the element inside of the brackets. The numbering starts at zero. So, `notes[0]` is the first element, `notes[1]` is the second, and so on. Lastly, an integer, called 'keyVal' is created to hold the value read from the input pin.

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

The `setup()` function only involves beginning the serial communication.

```
void loop(){
    keyVal = analogRead(inputPin);
    Serial.println(keyVal);
```

The first line in the code segment above starts the `loop()` function. The first thing done inside the `loop()` function is to check the value of the input pin using the `analogRead()` function and assigning the value to the 'keyVal' variable. The program then prints out the value just read using a print statement.

```
    if (keyVal == 1023) {
        tone(8, notes[0]);
    }
    else if (keyVal >= 980 && keyVal <= 1010) {
        tone(8, notes[1]);
    }
    else if (keyVal >= 505 && keyVal <= 515) {
        tone(8, notes[2]);
    }
    else if (keyVal >= 5 && keyVal <= 10) {
        tone(8, notes[3]);
    }
    else {
        noTone(8);
    }
}
```

The last part of the code contains an if-elseif-else statement with multiple elseif-statements. The if-statement and the elseif-statements each have different conditions that correspond to the different values that pressing different buttons will give. The if-statement has the condition that 'keyVal' is equivalent to 1023, or the maximum value of the input. Notice that each of the elseif-statement conditions is actually a range, for example the second elseif-statement's condition is that 'keyVal' be greater or equal to 505 and less than 515. This is because there is some natural variability in the voltage, meaning that it will go up and down a few numbers, so the range just makes sure that the program recognizes the value a button creates even though it changes slightly. Remember that `&&` is the logical and, meaning that to satisfy the condition the first **AND** the second statement must be true. Within each of these statements is a `tone()` function. They will turn on the piezo for one second at the frequency of one of the elements of the 'notes' array. The else-statement at the end makes sure that the piezo is turned off when no buttons are pressed by calling a new function, `noTone()`. The generic form of this function is shown below.

```
void noTone(int pin)
```

This function will stop any tone that a piezo is playing. The only argument is the pin number that the piezo is connected to.

```
int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

The program ends by repeatedly calling the **`setup()`** function and calling the **`loop()`** function an infinite number of times using an infinite while-loop.

If everything is working correctly then the piezo inside the Arduino should play different notes when different buttons are pressed and play no note when no buttons are pressed.

12.7 Exercises

1. Add a fifth button to make the instrument capable of playing five notes.
2. Modify the code so that the piezo plays a little song when one of the buttons is pressed
3. Try making the buttons each its own digital input. What possibilities does this open up? What are the downsides?

13 Data Acquisition and Plotting Using a Photo-resistor

13.1 Project 9: Graphing Light

Project Description: This project will read and graph values coming in from a photo-resistor

13.2 New Concepts

The last project introduced the concept of arrays and with this a powerful task can be accomplished, graphing. This project will detail how to use arrays to collect data from a photo-resistor sensor and then graph it. Graphing is immensely helpful as it allows an engineer or scientist to gain a visual understanding of what is happening rather than trying to decipher a massive list of numbers.

New Functions

- `void CPlot.title(string_t title)`
- `void CPlot.label(int axis, string_t label)`
- `void CPlot.plot2DCurve(array double x[], array double y[], int n)`
- `void CPlot.plotting(void)`

13.3 Required Components and Materials

- Breadboard
- 1 Photo-Resistor
- 1 10kΩ (Br-Bl-O) Resistor

13.4 Building the Circuit

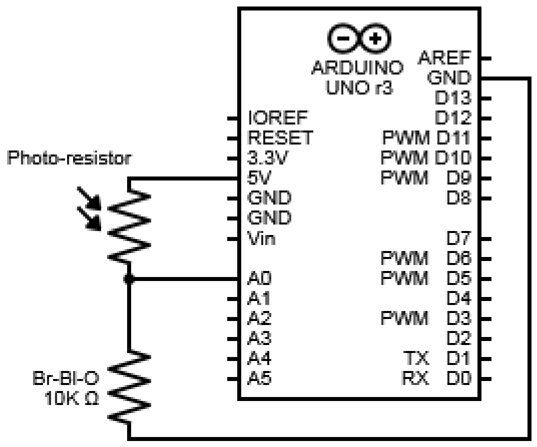
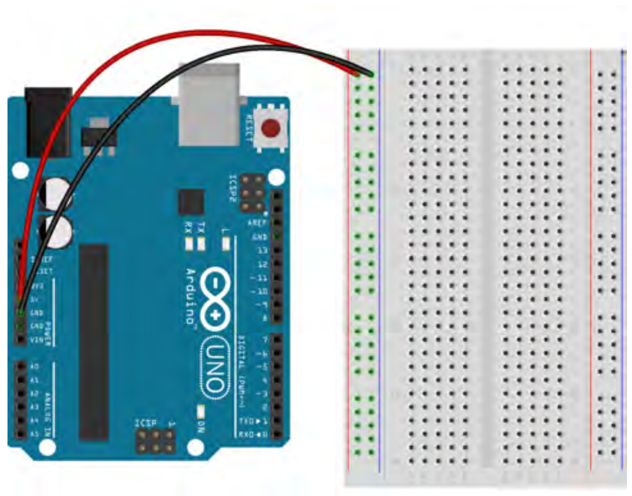


Figure 32: Graphing Light Circuit Diagram

The circuit for this project will actually be identical to the one used in Section 11 aside from the piezo. The first step to the build is to plug in the power wires, connecting the '5V' pin to the '+' column and the 'GND' pin to the '-' column.



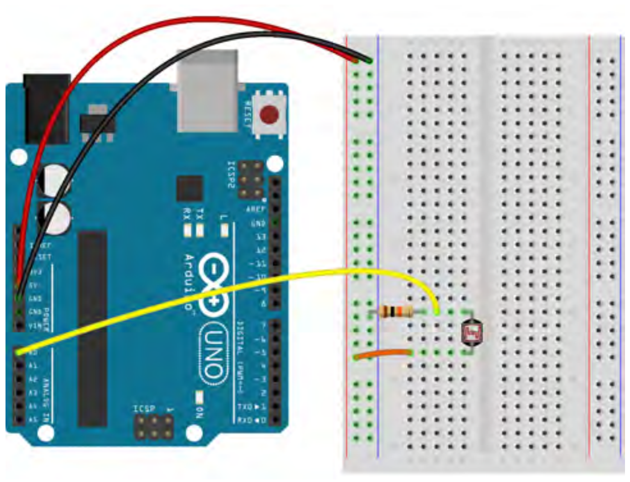
Step 1

Next, plug in the photo-resistor into the breadboard so that the two leads are in separate rows.

Take a Brown-Black-Orange resistor and plug one end into the same row as one of the leads of the photo-resistor. Plug the other end into the ground column.

Grab another wire and plug one end into the row with the other photo-resistor lead, plugging the other end of the wire into the positive power column.

Finally, take one more wire a plug one end of it into the same row as the resistor and photo-resistor. Connect the other end to the analog pin 'A0' on the Arduino.



Step 2

Pin	Code	Input	Output
A0	A0	✓	

Table 11: Graphing Light Project Pin Assignment Chart

13.5 Writing the Code

The following is the code for this project:

```
// file: graphingLight.ch
// Graph large numbers of values read from a photo-resistor against time

#include <arduino.h>
#include <chplot.h>

// Declare pin assignment variables
const int sensorPin = A0;

// Declare variable to keep track of location inside of the arrays
int element = 0,
    interval = 100,
    size = 100;

// Declare two arrays to hold all of the values from the sensor and the time
array double light[size], timeSec[size];
```

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

void loop(){
    // Read the input value from the photo-resistor and assign it to the current
    //element inside of the light array
    light[element] = analogRead(sensorPin);

    // Calculate the current time in seconds and assign it to the current element
    //inside of the sec array
    timeSec[element] = (element*interval)/1000.0;

    // Print out the data from this pass through the for-loop
    Serial.print("Sensor Value:");
    Serial.print(light[element])
    Serial.print("\t Time:");
    Serial.println(timeSec[element]);

    // Delay the program for 100 milliseconds before repeating the for-loop
    delay(interval);
}

int main(){
    setup();
    while(element < size) {
        loop();
        element++;
    }

    // Plot all of the data recorded

    CPlot plot;
    plot.title("Light vs. Time");
    plot.label(PLOT_AXIS_X, "Time(s)");
    plot.label(PLOT_AXIS_Y, "analogRead() Return Value");
    plot.data2DCurve(timeSec, light, size);
    plot.plotting();

    return 0;
}
```

There a slight change in the beginning portion of this code compared to the other projects, the inclusion of a new header file.

```
#include @<<arduino.h>>@
#include @<<chplot.h>>@
```

This program requires the inclusion of the `chplot.h` header file. This header file contains all of the function for creating graphs in Ch .

```
const int sensorPin = A0;

int element = 0,
    interval = 100,
    size = 100;
```

First, a `const int` type variable is declared to hold the number of the pin that the photo-resistor is connected to. Next, three integers are created. The first is an integer called ’element’. This will hold the number of the current element inside of the arrays that will be declared shortly. The second integer is called ’interval’ and will be the number of milliseconds between each time the program reads the photo-resistor. It is pre-set to 100 milliseconds and but can be easily be changes by the user. The last integer is called ’size’ and defines the total number of elements that the arrays will hold. It is also pre-set to 100.

```
array double light[size], timeSec[size];
```

The code segment above creates two arrays composed of `double` type variables. One is called ’light’ and will hold all of the values read from the photo-resistor, and the other is called ’timeSec’ and will hold all the times, in seconds, that the photo-resistor is read. They need to be of type `double` because the graphing function for Ch require it. Remember from the last project that arrays can be created two ways, the first being with empty brackets but set equal to all of the values that will be inside of the array, and the second being with a number inside of the brackets representing the number of elements that the array will eventually hold. The arrays declared above are of the second type. Because the variable ’size’, pre-set to 100, is inside of the brackets, then both will hold 100 elements, meaning that within each array will be 100 `double` variables. Since the specific values that will go into the arrays are not known yet, the size needs to be put into the bracket so that the program knows how much memory to save for the arrays.

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

The `setup()` function only contains the beginning of the serial communication.

```
void loop(){
    light[element] = analogRead(sensorPin);

    timeSec[element] = (element*interval)/1000.0;
```

Now for the code inside of the `loop()` function. First, the pin connected to the photo-resistor is read by the `analogRead()` function and the value is assigned to the current element of the ’light’ array. Next, the current time is calculated and assigned to the current element of the ’timeSec’ array. The ’element’ variable, or in other words the number of times the program has gone through the for-loop, is multiplied by ’interval’, giving the time in milliseconds since the program started repeating the for loop. The reason the number is ’interval’ is because later in the code, at the end of the for-loop, the program is paused for ’interval’ number of milliseconds. Thus in the pre-set case, the program goes through the for-loop roughly every 100 milliseconds. So, multiplying the current ’element’ value by 100 gives the total number of milliseconds since the for-loop started. Finally, because the time in seconds is desired, the number of milliseconds just calculated is divided by 1000, to give seconds, and the number is assigned to the current element within the ’timeSec’ array. Remember, ’timeSec’ is filled with `double` type variables and that dividing an integer by an integer will give an integer. Thus the number of milliseconds needs to be divided by 1000.0 instead of 1000 so that the result will be a decimal number.

```
        Serial.print("Sensor Value:");
        Serial.print(light[element])
        Serial.print("\t Time:");
        Serial.println(timeSec[element]);

        delay(interval);
    }
```

Next, the program prints out the value of the photo-resistor that was just read and the current time that was just calculated. The program then uses the `delay()` function to pause the program for 'interval' number of milliseconds in order to give a small amount of time between the readings of the sensor. Lastly, the brace closes out the for-loop.

```
    int main(){
        while(element < size) {
            loop();
            element++;
        }
```

The idea is to take a reading from the photo-resistor for each element inside of the arrays. This means that we need to read the photo-resistor 'size' number of times. So for this project, instead of using an infinite while-loop, which will repeat forever, a while-loop with the `(element < size)` condition is used because it will repeat a specific number of times, in this case 100 times. The 'element' variable is initialized at the start of the program to be 0. The line at the end of the while-loop, `element++`, means that the 'element' variable will be increased by one, the exact same as `element = element + 1`. So, the while-loop created above will start with 'element' equal to zero and will loop, increasing 'element' by one each time, until 'element' is equal to the 'size' of the array. Thus for this case, the code inside of the while-loop will run once for each value of 'element' from 0 to 99, once for each element inside of the 'light' and 'timeSec' arrays. Remember that the numbering of the elements of arrays starts at 0, so an array with 100 elements will have elements numbered from 0 to 99.

```
        CPlot plot;
        plot.title("Light vs. Time");
        plot.label(PLOT_AXIS_X, "Time(s)");
        plot.label(PLOT_AXIS_Y, "analogRead() Return Value");
        plot.data2DCurve(timeSec, light, size);
        plot.plotting();

        return 0;
    }
```

The last thing that this program does is graph all of the data that it just collected. It does this by using a series of functions contained in a class called `CPlot`. The first line in the code segment above creates an instance of this class, called 'plot'. The first function that is used is called `title()` and sets the title of the plot. The generic form is shown below.

```
    void CPlot.title(string_t title)
```

The `title()` function only has on argument, a string of type `string_t`, that will be the title of the plot. The next two lines make use of the `label()` function to set the two axis titles. The generic form is shown below.

```
void CPlot.label(int axis, string_t label)
```

The first argument tell the function which axis to work on. In the code above, the **PLOT_AXIS_X** and **PLOT_AXIS_Y** macros are used to specify the axis. These macros are defined inside of the **@<chplot.h>@** header file. The second argument is a string, of type **string_t**, which will become the label of the axis. The next line uses the **data2DCurve()** function to plot the two arrays of data that were created earlier in the program. The generic form is shown below.

```
void CPlot.plot2DCurve(array double x[], array double y[], int n)
```

The first two arguments are arrays that contain the data for the x and y axes respectively. In this case, the array of data that corresponds to the x-axis is in the 'timeSec' array and the data for the y-axis is in the 'light' array. The third argument, an integer called 'n', is the number of elements in the two arrays. In this case the third argument would be the 'size' variable. The last step in plotting is to use the **plotting()** function. This function actually creates the graph as a figure that will pop up in a new window. The generic form of this function is shown below.

```
void CPlot.plotting(void)
```

If everything is working correctly then when the program is running the values being read from the photo-resistor should be printed out constantly. In it's pre-set condition, after reading the photo-resistor 100 times the program should stop and graph the data just collected. The graph should look something like Figure 33 below. Try shading the photo-resistor while the program is running and use both the values being printed out and the graph at the end to verify the drop in the value of the photo-resistor. Which was easier to use?

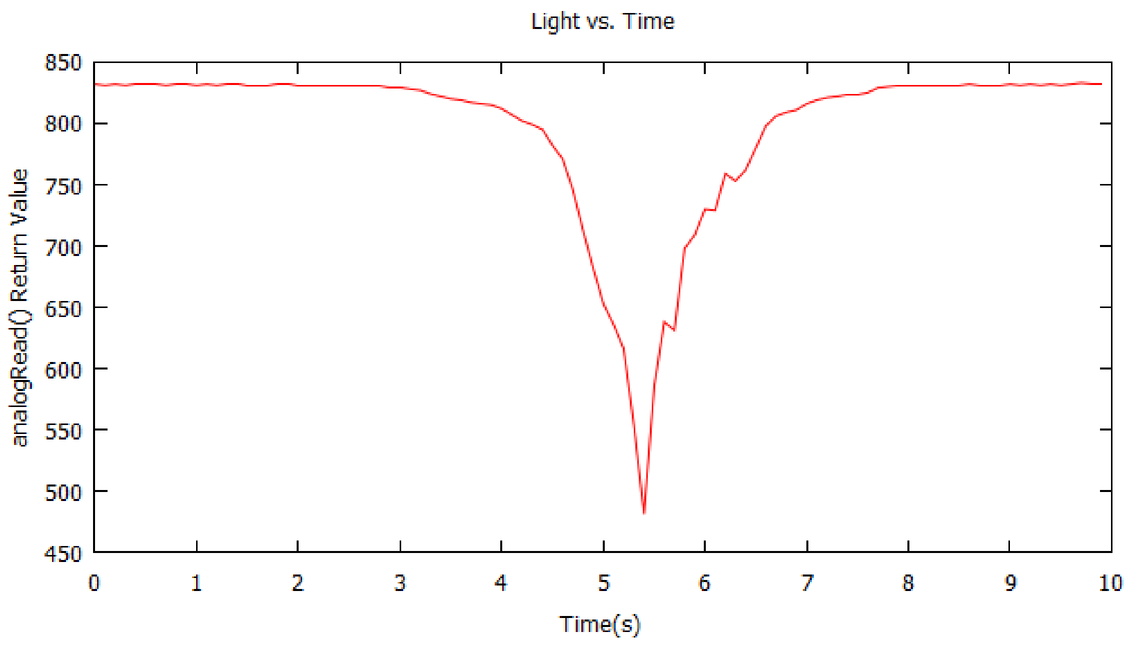


Figure 33: Graphing Light Project Example Graph

13.6 Exercises

1. Increase the 'interval' variable to 1000 and reduce the 'size' variable to 10. The program is still reading the photo-resistor for ten seconds but what differences are noticeable? Now decrease 'interval' to 10 and increase 'size' to 1000. What are the noticeable differences? What could be possible problems with this?

2. Replace the photo-resistor with the temperature sensor and modify the program to monitor the temperature in the room over the course of an hour. There is more than one way to do this.

14 Data Acquisition and Plotting with Multiple Inputs and Outputs Using a Photo-resistor and a Potentiometer

14.1 Project 10: 3D Graphing

Project Description Now that the last project gave a simple example of using arrays to gather data and then graphing the array, it is time to step it up a notch. This project will use two inputs instead of one, add an output, and graph the results on a 3D plane. The brightness of an LED will be controlled by both a potentiometer and a photo-resistor and the results will be graphed.

14.2 New Concepts

The project is a small example of how two inputs, especially analog inputs, can interact with each other to produce an output. It also shows how to make three dimensional plots in Ch.

New Functions

- `void CPlot.data3DCurve(array double x[], array double y[], array double z[], int n)`

14.3 Required Components and Materials

- Breadboard
- 1 Photo-Resistor
- 1 220Ω (R-R-Br) Resistor
- 1 10kΩ (Br-Bl-O) Resistor
- 1 Potentiometer
- 1 LED

14.4 Building the Circuit

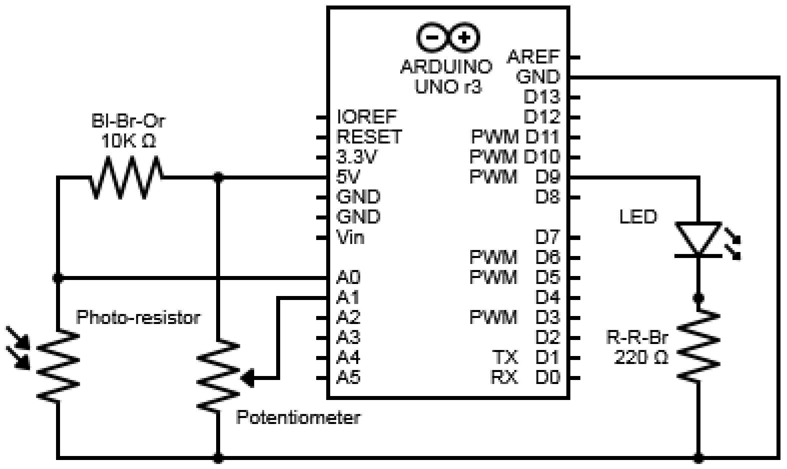
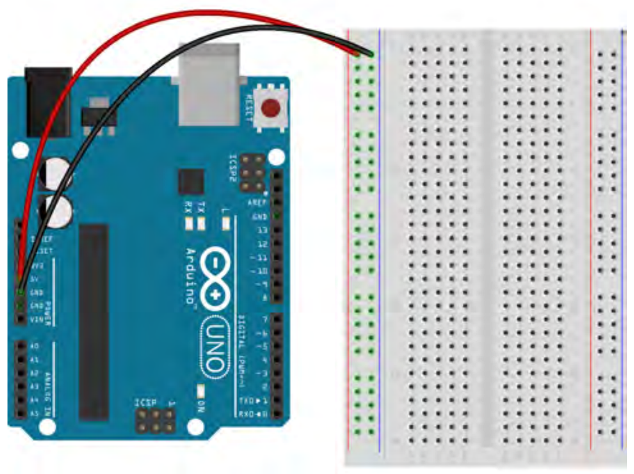


Figure 34: Graphing Light Circuit Diagram

First step, connect the power wires like at the beginning of the previous projects.



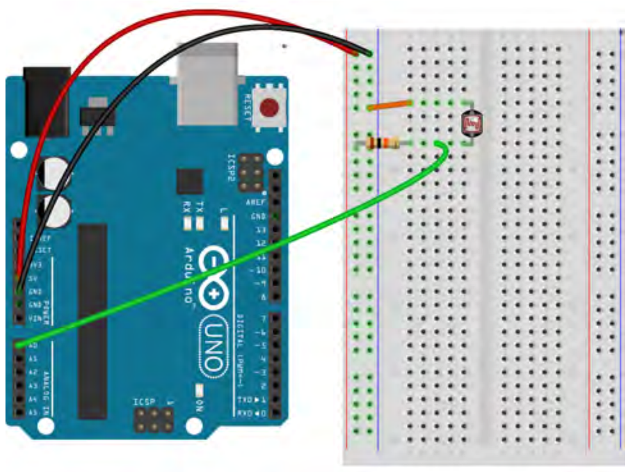
Step 1

Next, to place the photo-resistor. Plug the photo-resistor into the breadboard near one of the ends (to leave room for the potentiometer and the LED). Make sure it's two leads do not share a row with each other or anything else.

Take a wire and plug one end into one of the rows with a photo-resistor lead and plug the other end into the ground column.

Take an Brown-Black-Orange and plug one end into the row with the other photo-resistor lead. Plug the other end of the resistor into the '+' column.

The last part for the photo-resistor is to connect a wire between the 'A0' pin of the Arduino and the row that has the photo-resistor and the resistor.

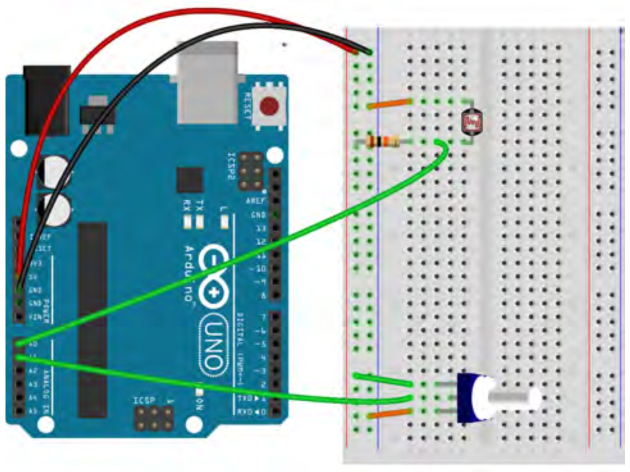


Step 2

The photo-resistor is done, now it is time for the potentiometer. Plug the potentiometer into the breadboard so that it's leads are spanning the trench running down the center of the breadboard. When placing it, position the potentiometer to the opposite side of the breadboard as the photo-resistor.

On one side of the potentiometer should have two leads. Connect one lead to the power rail and connect the other to the ground rail, using two wires.

Now connect the single lead on the other side of the potentiometer to the 'A1' pin on the Arduino using a wire.

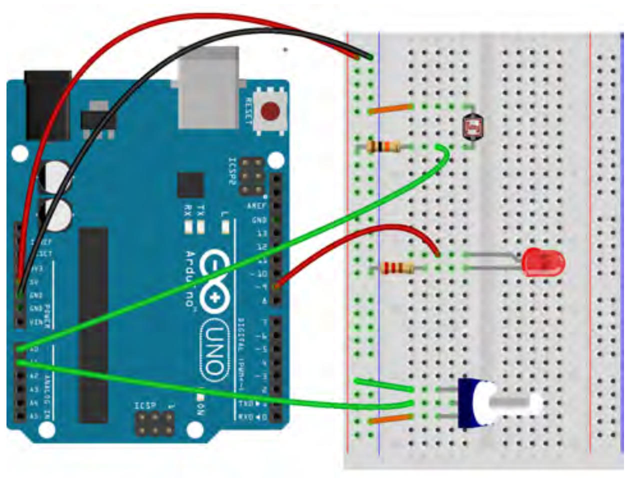


Step 3

It's time to place the LED. Plug an LED into the breadboard, near the middle, so that the two leads do not share a row with each other or anything else.

Take a Red-Red-Brown resistor and plug end into the row with the LED's negative lead, the shorter one. Plug the other end of the resistor into the ground column.

Finally use a wire to connect the row with the positive lead of the LED, the longer one, and the digital '9' pin on the Arduino.



Step 4

Pin	Code	Input	Output
A0	A0	✓	
A1	A1	✓	
D9	9		✓

Table 12: 3D Graphing Project Pin Assignment Chart

14.5 Writing the Code

The following is the code for this project:

```
// file: 3DGraphing.ch
// graph an output with two inputs on a 3D field

#include <arduino.h>
#include <chplot.h>

// Declare pin assignment variables
const int photoPin = A0,
           potPin = A1,
           LEDPin = 9;
```

```
// Declare sensor, calibration, and timing variables
int element = 0,
       photoVal,
       potVal,
       photoHigh,
       photoLow = 1023,
       calibTime = 10000,
       size = 100,
       interval = 100;

double photoRange;

// Declare arrays to hold sensor data
array double photo[size],
             pot[size],
             brightness[size];

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

    // Set the LED to output mode
    pinMode(LEDPin, OUTPUT);

    // Calibrate the photo-resistor
    Serial.println("Calibrating..");
    while (millis() < calibTime) {
        photoVal = analogRead(photoPin);
        if (photoVal > photoHigh) {
            photoHigh = photoVal;
        }
        if (photoVal < photoLow) {
            photoLow = photoVal;
        }
    }

    // Print out the calibrated high and low values, calculated the new range
    Serial.print("photoHigh:");
    Serial.println(photoHigh);
    Serial.print("photoLow:");
    Serial.println(photoLow);
    photoRange = photoHigh-photoLow;
}

void loop(){

    // Read the value from the photo-resistor and calculate the ratio value using the
    //calibration then assign this value to the current element of the array
    photoVal = analogRead(photoPin);
    photo[element] = (photoVal-photoLow)/photoRange;

    // Read the value from the potentiometer and calculate its ratio to maximum then
    //assign this value to the current element of the array
    potVal = analogRead(potPin);
    pot[element] = potVal/1024.0;

    // Calculate the output ratio by multiplying the two input ratios then multiply
    //the output ratio by the range and write the value to the LED
    brightness[element] = pot[element]*photo[element];
    analogWrite(LEDPin, brightness[element]*255);

    // Print out all of the calculated ratios and delay program before the loop repeats
    Serial.print("Photo-resistor:");
    Serial.print(photo[element]);
    Serial.print("\t Pot:");
    Serial.print(pot[element]);
    Serial.print("\t Brightness:");
    Serial.println(brightness[element]);
    delay(interval);
}

int main(){
    setup();
    while(element < size) {
        loop();
        element++;
    }

    // Plot the brightness of the LED against the input ratios of the photo-resisitor
    //and the potentiometer

    CPlot plot;
    plot.title("Brightness of LED controlled by Potentiometer and Photo-resistor");
    plot.label(PLOT_AXIS_X, "Potentiometer");
    plot.label(PLOT_AXIS_Y, "Photo-resistor");
    plot.label(PLOT_AXIS_Z, "Brightness");
    plot.data3DCurve(pot, photo, brightness, size);
    plot.plotting();

    return 0 ;
}
```

Like the last project, this program needs to include the `chplot.h` header file to be able to graph.

```
const int photoPin = A0,
          potPin = A1,
          LEDPin = 9;

int element = 0,
    photoVal,
    potVal,
    photoHigh = 0,
    photoLow = 1023,
    calibTime = 10000,
    size = 100,
    interval = 100;
```

First, the program starts off by creating all of the integer type variables that it will need. Three with the `const` modifier hold the number of the pins that the photo-resistor, potentiometer, and LED are attached to. Next, the program creates 'element' to keep track of the current element of the arrays, to be created in the next segment. After that, the program creates 'photoVal' and 'potVal' to be the raw read values from the photo-resistor and the potentiometer. 'photoHigh' and 'photoLow' will be used to keep track of the maximum and minimum value of the photo-resistor and 'calibTime' will set the length of time in a calibration process, like that in the Light Theremin project. 'calibTime' is in units of milliseconds and is preset to 10000, or ten seconds. 'photoHigh' is preset to 0 so that there is no where to go but up, and the opposite is true of 'photoLow'. Next, 'size' defines the size of the arrays is elements, and is preset to 100. Lastly, 'interval' defines the time, in milliseconds, between repetitions of a for-loop.

```
double photoRange = 0.0;

array double photo[size],
             pot[size],
             brightness[size];
```

In the above segment of code, the program defines all of the double type variables that it will need. First, the program creates a regular, individual, double type called 'photoRange'. This will hold the difference between the 'photoHigh' and 'photoLow' variables created in the last segment. Next comes the arrays. The program creates three arrays with 'size' number of double variables. Remember that 'size was preset to 100, so each of these arrays will have 100 elements. The arrays will hold data about the photo-resistor, potentiometer, and the brightness of the LED.

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

    pinMode(LEDPin, OUTPUT);
```

The line above sets the pin connected to the LED to output mode.

```
    Serial.println("Calibrating..");
    while (millis() < calibTime) {
        photoVal = analogRead(photoPin);
        if (photoVal > photoHigh) {
            photoHigh = photoVal;
        }
        if (photoVal < photoLow) {
            photoLow = photoVal;
        }
    }
```

The code segment above is the same calibration process used in the Light Theremin project. Remember, the `millis()` function returns how much time, in milliseconds, has passed since the program started. The condition of the while-loop above is that the `millis()` function be less than the variable 'calibTime', preset earlier to 10000 milliseconds. So, this while loop will repeat until 10000 milliseconds, or 10 seconds, has passed. In the first line inside of the while-loop, the program reads the pin connected to the photo-resistor and stores the value in the 'photoVal' variable. Next are two if-statements. If the value just read is larger than the current maximum, the variable 'photoHigh', then it becomes the new maximum. The if the value just read is smaller that the current minimum, the variable 'photoLow', then is becomes the minimum. Remember, the User must participate in the calibration process. So, when the calibration starts, shown by the program printing out the "Calibrating..." string, make sure to block out as much light from the sensor as possible by moving your hands around the photo-resistor the same way they will be moving during the actual program.

```
    Serial.print("photoHigh:");
    Serial.println(photoHigh);
    Serial.print("photoLow:")
    Serial.println(photoLow);
    photoRange = photoHigh-photoLow;
  }
```

This is the last little bit of preparatory code. The program will print out the maximum and minimum values found by the photo-resistor calibration, then calculate the difference between those values and save that in the 'photoRange' variable.

```
  void loop(){

    photoVal = analogRead(photoPin);
    photo[element] = (photoVal-photoLow)/photoRange;

    potVal = analogRead(potPin);
    pot[element] = potVal/1024.0;
```

Now for the main body of code. The segment above starts a for-loop that will hold the body. The initial condition of the for-loop is that the 'element' variable be set to zero, the ending condition is that 'element' be less than 'size', and the repeating condition is that 'element' increase by 1 every time. So, this for loop will start with 'element' equal to 0, increment 'element' by one every time the program goes through the loop, and continue this until 'element' is greater than or equal to 'size'. Once inside of the for-loop, the program reads the pin connected to the photo-resistor and saves the read value inside of the 'photoVal' variable. It then calculates the ratio of the read value to the possible range of values. This is achieved by subtracting the minimum value, found during the calibration, which makes the read value relative to the range. This is then divided by the range of the photo-resistor, giving the ratio. The value is saved inside of the current element of the 'photo' array. Next, the program reads the pin connected to the potentiometer and saves the value inside of the 'potVal' variable. It the calculates the ratio to the range, similar to what was done with the photo-resistor value, except this time it is much simpler. The range of the potentiometer is the full 1023 so no subtraction is needed. The ratio value is saved inside of the current element of the 'pot' array.

```
    brightness[element] = pot[element]*photo[element];
    analogWrite(LEDPin, brightness[element]*255);

    Serial.print("Photo-resistor:");
    Serial.print(photo[element]);
    Serial.print("\t Pot:");
    Serial.print(pot[element]);
    Serial.print("\t Brightness:");
    Serial.println(brightness[element]);
    delay(interval);
  }
```

The second half of the for-loop is in the code segment above. The first line multiplies the current elements of the 'pot' and 'photo' arrays together. The result of this calculation is saved inside of the current element of the 'brightness' array. The next line uses the `analogWrite()` function to write the 255, or the possible range of values to analog write, multiplied by current element of the 'brightness' array, the one just calculated, to the LED pin. This uses the PWM capabilities of the digital pin, as discussed in earlier projects. Remember, the values saved in the 'pot' and 'photo' arrays are ratios, or decimal numbers, so the values in the 'brightness' array will also be decimal numbers. So, the 'brightness' value is also a ratio, which is why it was multiplied by 255. The value of the 'brightness' ratio, and thus how bright the LED will be, depends of how big or small both of the photo-resistor and potentiometer values are. The photo-resistor be at its maximum value, but if the potentiometer is at 0 then the LED will not be lit, and vice versa. Finally, The program prints out the values of the current elements of the 'photo', 'pot', and 'brightness' arrays, then pauses for 'interval' number of milliseconds before repeating the loop.

```
int main(){
    setup();
    while(element < size) {
        loop();
        element++;
    }

    CPlot plot;
    plot.title("Brightness of LED controlled by Potentiometer and Photo-resistor");
    plot.label(PLOT_AXIS_X, "Potentiometer");
    plot.label(PLOT_AXIS_Y, "Photo-resistor");
    plot.label(PLOT_AXIS_Z, "Brightness");
    plot.data3DCurve(pot, photo, brightness, size);
    plot.plotting();

    return 0;
}
```

Similar to the last project, this project makes use of a while-loop with a `(element < size)` condition to make the loop function iterate a specific number of times, in this case 100 times.

After the for-loop has finished, the program goes to graph the data. It does this in a similar way to the last project, except it uses the `data3DCurve()`. First, an instance of the `CPlot` class, called 'plot', is created. Next, title and axis labels are set using the `title()` and `label()` functions. Then, the `data3DCurve()` function is used to plot the LED brightness data against the potentiometer and photo-resistor data. The generic form is shown below.

```
void CPlot.data3DCurve(array double x[], array double y[], array double z[], int n)
```

The first three arguments are the arrays of data for each axis, in this case 'pot', 'photo', and 'brightness' respectively. The last argument is an integer, 'n', which is the number of elements in the three arrays, in this case 'size'. Lastly, the `plotting()` function is called to create the plot.

If everything is working correctly then the brightness of the LED should vary from both the potentiometer being turned and the photo-resistor being blocked. Every pass through the for-loop, the program should print out the data it has read from the photo-resistor and potentiometer and has written to the LED. At the end, a three dimensional graph similar to 35 should pop up showing the path of the brightness of the LED.

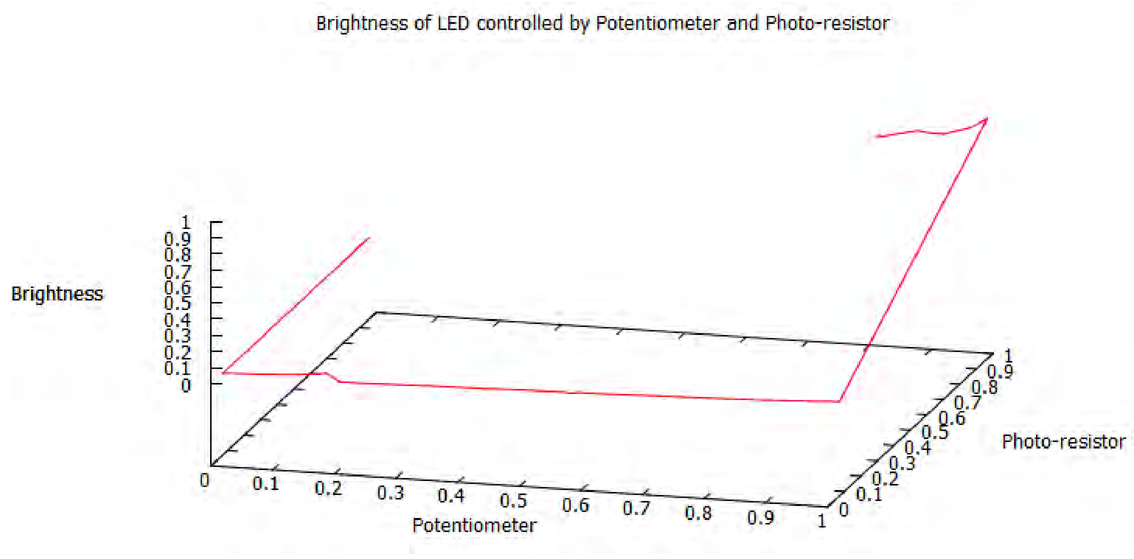


Figure 35: 3D Graphing Project Example Graph

14.6 Exercises

1. Try calculating the ratio of the photo-resistor value, the value of the 'photo' element in the same manner as the potentiometer. What happens? Why is calibration important?

2. Create two two-dimensional graphs with both having the brightness values for y, but with one having the potentiometer values as x and the other having the photo-resistor values for x. Do these graphs tell the whole story?

15 Using Data Acquisition to Verify Ohm's Law

15.1 Project 11: Science!

Project Description The ability to record, visualize, and analyze data, as demonstrated in the previous two projects, is an incredibly powerful tool in the advancement of science and engineering. It has lead to all of the empirical laws and formulas that make modern engineering possible. This project will use data acquisition to verify one of the oldest physical laws of electricity, Ohm's Law. Varying levels of voltage will be applied to a resistor and the current will be measured by a tool called a multimeter.

15.2 New Concepts

The project does introduce two new functions that will read inputs from the user. In addition, it will formally introduce the concepts of voltage, current, and resistance (which may have been mentioned previously) and will introduce a new tool, the multimeter.

New Functions

- `int Serial.parseInt(void)`
- `double Serial.parseFloat(void)`
- `void CPlot.legend(string_t name, int series)`

15.3 Required Components and Materials

- Breadboard
- 1 Multi-meter
- 1 220Ω (R-R-Br) Resistor

15.4 Building the Circuit

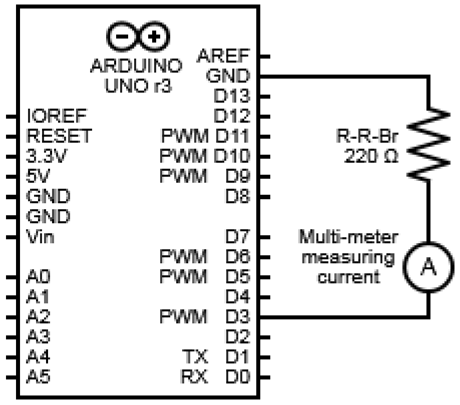
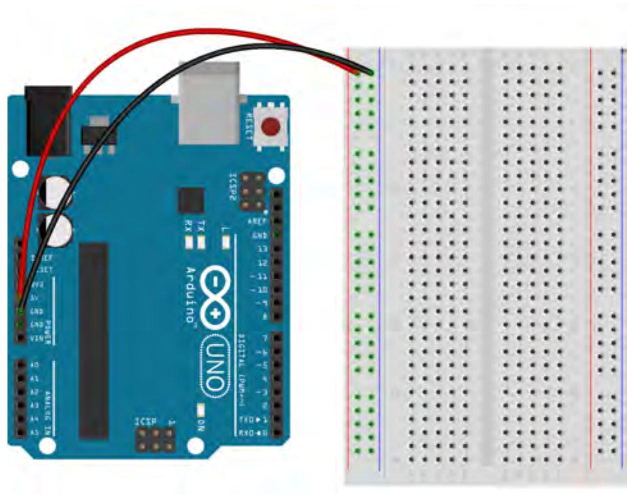


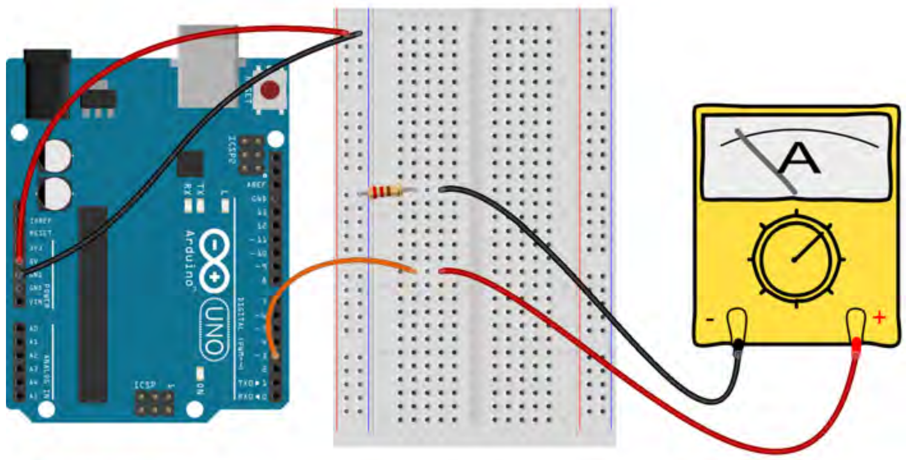
Figure 36: Science! Circuit Diagram

This will be a very simple circuit. First, connect the power and ground rails to '5V' and 'GND' using two wires.



Step 1

Next connect one lead of a Red-Red-Brown resistor to Digital Pin 3 on the Arduino. Connect the other lead to any of the rows. Take a wire and plug one end into any of the rows, but not the one with the resistor, and plug the other end into the 'GND' column. Thats it! The circuit will be completed by the multi-meter



Step 2

Pin	**Code**	**Input**	**Output**
D3	3		✓

Table 13: Science! Project Pin Assignment Chart

15.5 Writing the Code

The following is the code for this project:

```
// file: science.ch
// Graph both the theoretical and measeured relationship between
// voltage and current

#include <arduino.h>
#include <chplot.h>

// Declare pin assignment and array size variables
const int pin = 3,
       size = 16;

// Declare variables to keep track of location inside of the arrays,
// delay the program and hold the resistance value input from the user
int element = 0,
    interval = 1000,
    resistance = 0;

// Declare three arrays to hold the voltages and measured and theoretical
// currents
array double voltage[size], currentActual[size], currentTheo[size];
```

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

    // Set the pin mode of the pin to output mode
    pinMode(pin, OUTPUT);

    // Print out the prompt for the user to input the resistance
    Serial.println("Please Input the Value of the Resistor in Ohms");
    Serial.print("> ");

    // Read user input
    while(resistance == 0){
      resistance = Serial.parseInt();
    }

    // Set the first element of all the array to zero
    voltage[element] = 0;
    currentTheo[element] = 0;
    currentActual[element] = 0;
    element++;
}

void loop(){
    // Calculate the value to write for this interation of the loop
    // and write it
    voltage[element] = element*17;
    analogWrite(pin, voltage[element]);

    // Calculate the current voltage being applied
    voltage[element] = voltage[element]*5/255;

    // Calculate the theoretical current in milliamps
    currentTheo[element] = voltage[element]/resistance*1000;

    // Print out prompt for user to input the measured current
    Serial.print("Please Input the Measured Current in milliAmperes for ");
    Serial.print(voltage[element]);
    Serial.println(" Volts");
    Serial.print("> ");

    // Read user input
    while(currentActual[element] == 0){
       currentActual[element] = Serial.parseFloat();
    }

    // Delay the program for 1 second before repeating the loop
    delay(interval);
}

int main(){
    setup();
    while(element < size) {
        loop();
        element++;
    }

    // Print out the resistance value based on measurements
    Serial.print("Resistance Based on Measured Data (R): ");
    Serial.println(voltage[element-1]/currentActual[element-1]*1000);

    // Plot all of the data recorded
    CPlot plot;
    plot.title("Voltage vs. Current");
    plot.label(PLOT_AXIS_Y, "Current [mA]");
    plot.label(PLOT_AXIS_X, "Voltage [V]");
    plot.data2DCurve(voltage, currentActual, size);
    plot.data2DCurve(voltage, currentTheo, size);
    plot.legend("Actual", 0);
    plot.legend("Theoretical", 1);
    plot.plotting();

    return 0;
}
```

This program needs to include `chplot.h` in order to enable plotting capabilities.

```
const int pin = 3,
          size = 16;

int element = 0,
    interval = 1000,
    resistance = 0;

array double voltage[size], currentActual[size], currentTheo[size];
```

First, the program creates two integer varaibles with the `const` modifier. The first is called 'pin' and is the number of the pin used. The second is called 'size' and represents the size of the arrays created in the next few lines. Next, the program creates three more integers without any modifier. The first, 'element' will keep track of the number of times the program goes through the `loop()` function. Second, 'interval, represents the amount of time, in milliseconds, that the program will pause in between each iteration. Last, 'resistance', will be the value for the resistance of the resistor, in Ohms, that the user will input later. Finally, the program creates three arrays of `double` data type, one to hold all of the voltage values, another to hold all of the user inputted current measurements, and one more to hold all of the theoretical current values.

```
void setup(){
    Serial.begin(9600);
    pinMode(pin, OUTPUT);

    Serial.println("Please Input the Value of the Resistor in Ohms");
    Serial.print("> ");

    while(resistance == 0){
      resistance = Serial.parseInt();
    }
```

Inside the **setup()** function, the program prints out a prompt for the user to input the resistance value of the resistor used to build the circuit. To get the resistance value in Ohms from the color code on the resistor, please refer to Section 21.2 in the Appendix. The program then enters a while-loop with the condition `resistance == 0` meaning that the program will repeat the code inside of the loop as long as the 'resistance' variable is equal to zero. The only line inside the while-loop is a call to the new function, `parseInt()`, which is a member function of the `Serial` class. The generic form of this function is shown below.

```
int Serial.parseInt(void)
```

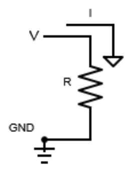
This function will read an integer value that a user inputs and return it. So, in the case above the `parseInt()` function tries to read any user inputted integer value and assigns it to the 'resistance' variable. Thus, the while-loop will continue while 'resistance' is zero until the user inputs a value, resistance becomes that value and stops being zero, and the program breaks out of the while-loop and continues. This is done so that the program will wait to continue until the user inputs something.

```
    voltage[element] = 0;
    currentTheo[element] = 0;
    currentActual[element] = 0;
    element++;
}
```

The last part of the **setup()** function sets the first elements of all three arrays to zero. Remember, 'element' was initialized to be zero and still should be at this point so the above code should assign values to the zeroth elements, or the first elements, in each array. This assumes that for zero voltage there will be zero current which is a good assumption. The last line uses the '**++**' operator to increase 'element' by one.

```
void loop(){
    voltage[element] = element*17;
    analogWrite(pin, voltage[element]);

    voltage[element] = voltage[element]/255*5;

    currentTheo[element] = voltage[element]/resistance*1000;
```

The first line of the `loop()` function sets the element of the 'voltage' array to 'element' multiplied by 17. This value is then written to digital pin 3 using the `analogWrite()` function. Remember that only integers between 0 and 255 can be written with `analogWrite()` so the value 17 is 255 divided by fifteen, which is the number of non-zero elements in the three arrays. Thus, the first two lines will increase the value, and thus the voltage, being written to the pin by equal steps of 17 for every time the program goes through the **loop()** function. The next line re-assigns the value of the element in the 'voltage' array to the actual voltage the pin will output. The actual voltage is found by dividing the value just written with `analogWrite()` by the maximum value that can be written, 255, giving the ratio of the voltage written to the maximum voltage. This ratio is then multiplied by 5, the maximum voltage in volts that can be output by the pin, which will give the actual voltage in volts that the pin will output on this iteration of the **loop()** function. The last line calculates what the theoretical current is based on Ohm's Law, a mathematical formula that relates voltage, current, and resistance, that will be explained now and is shown below.



$$I = \frac{V}{R}$$

In words, the formula states the when a voltage of value V, in volts, is applied to a resistor with value R, in Ohms, a current will be created of value I, in Amperes, equal to the voltage divided by the resistance. This project will see if this law is valid by calculating the theoretical current and measuring the actual current then comparing the two.

So, the last line in the last code segment calculates the theoretical current by dividing the voltage that the pin will output by the known resistance. It also multiplies the value by 1000, this is to make the value in units of milliamps rather than Amperes. A milliamp is a thousandth of an Ampere and because most of the currents in this project will be on the order of thousandths of an Amperes, this is the most appropriate and easiest unit to use.

```
    Serial.print("Please Input the Measured Current in milliAmperes for ");
    Serial.print(voltage[element]);
    Serial.println(" Volts");
    Serial.print("> ");

    while(currentActual[element] == 0){
       currentActual[element] = Serial.parseFloat();
    }

    delay(interval);
}
```

The last part of the `loop()` function first prints out a prompt for the user to input their measured current, in milliamps, for the given voltage. Please see Section 21.3 in the Appendix for how to use a multi-meter to measure current and voltage. The red probe should be connected to the same row as the wire coming from the pin and the black probe should be connected to the row with the resistor. Poke the tips of the probes into the pins in each row and they should connect to the metal within the row, but this might take some pressure or wiggling to get a good connection. Next the program enters into a while-loop similar to the one we saw in the `setup()` function earlier. This time the condition is that the current element of the 'currentActual' array be equal to zero, meaning that the program will continue to iterate through the while-loop as long as the current element of 'currentActual' is zero. There is only one line inside the while-loop and it sets the element of 'currentActual' equal to the returned value of a new function called `parseFloat()`, with the generic form shown below

```
double Serial.parseFloat(void)
```

This function is very similar to the `parseInt()` function introduced earlier except that it reads a `double` type variable input from the user rather than an integer. The last line of the `loop()` function pauses the program between each iteration of the `loop()` function for a number of milliseconds equal to the 'interval' variable.

```
int main(){
    setup();
    while(element < size) {
        loop();
        element++;
    }

    Serial.print("Resistance Based on Measured Data (R): ");
    Serial.println(voltage[element-1]/currentActual[element-1]*1000);

    CPlot plot;
    plot.title("Voltage vs. Current");
    plot.label(PLOT_AXIS_Y, "Current [mA]");
    plot.label(PLOT_AXIS_X, "Voltage [V]");
    plot.data2DCurve(voltage, currentActual, size);
    plot.data2DCurve(voltage, currentTheo, size);
    plot.legend("Actual", 0);
    plot.legend("Theoretical", 1);
    plot.plotting();

    return 0;
}
```

The first part of the **main()** function is similar to that of the last two projects where the **setup()** function is called once and the **loop()** is repeatedly called using a while-loop for as many times as there are elements in the three arrays. After the program has finished calling the **loop()** function, it prints out the resistance based on the measured data. By rearranging Ohm's law, it can be seen that the resistance is the voltage divided by the current. In other words, the resistance is the slope of the line created by plotting voltage against current. The value of voltage divided by current is multiplied by 1000 to change the units of the current back from milliamps to Amperes so that the resistance will be in the appropriate units of Ohms. The last thing the program does is plot all of the data collected using the plotting function introduced in the previous two sections. The x-axis is the voltages and the y-axis is the currents, in milliamps. Notice that you can plot two lines on the same plot simply by having two calls to the **data2DCurve()** function. There is one new function, **legend()**, whose generic form is shown below.

```
void CPlot.legend(string_t name, int series)
```

This function creates a legend for the plot. The first argument is a string representing the name of the data series, in this case `"Actual"` and `"Theoretical"`. The second argument is an integer representing the series number of the data series. For example, since the **data2DCurve()** call for the actual current data is called first, its series number should be 0. Likewise, because the theoretical current data is plotted second, its series number is 1.

If everything works correctly then the user should be able to input the resistance of the resistor, the voltage and current should increase incrementally, the user should be able to input their measured current every time it does, and at the end the program should output a plot of the theoretical and actual currents against voltage. The output plot should look similar to Figure 37 below.

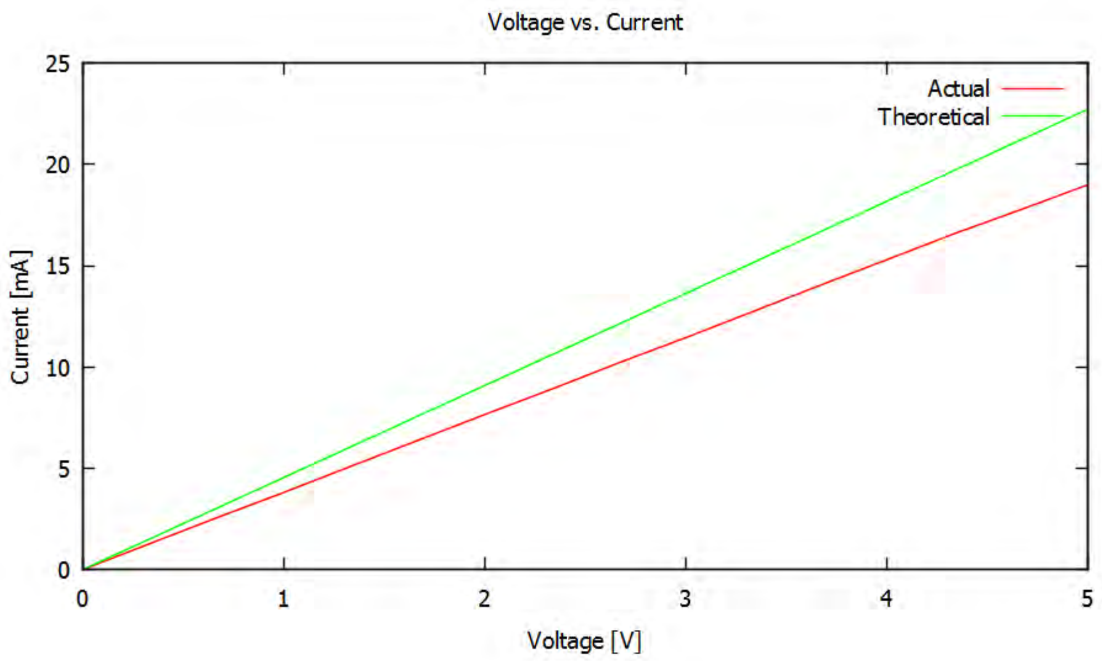


Figure 37: Science! Project Example Graph

15.6 Exercises

1. Do the plots of the theoretical current and the measured current match? What might be some reasons for any differences?

2. The multi-meter can also be used to measure resistance. Measure the actual resistance of your resistor and run the program again with this value. Is the measured current any closer to the theoretical current on the output plot?

3. The multi-meter can also be used to measure voltages. Connect the wire coming from digital pin 3 to the same row as the resistor and run the program again, measuring the voltage across the resistor at each point. Does the voltage match what it is supposed to be? Why might this be?

16 Timing a Program

16.1 Project 12: Digital Hourglass

Project Description There will be six LEDs that will light up in sequence at a user defined interval. When the last LED lights up, another interval will pass then the last LED will blink and all of the LEDs will turn off, starting the process over again. There will also be a button that, when pressed, will start the counting over again no matter where the counter was when the button was pressed.

16.2 New Concepts

This project will explore how to control the timing of events with code.

16.3 Required Components and Materials

- Breadboard
- 6 220Ω (R-R-Br) Resistor
- 1 10kΩ (Br-Bl-O) Resistor
- 1 Push-Button Switch
- 6 LED

16.4 Building the Circuit

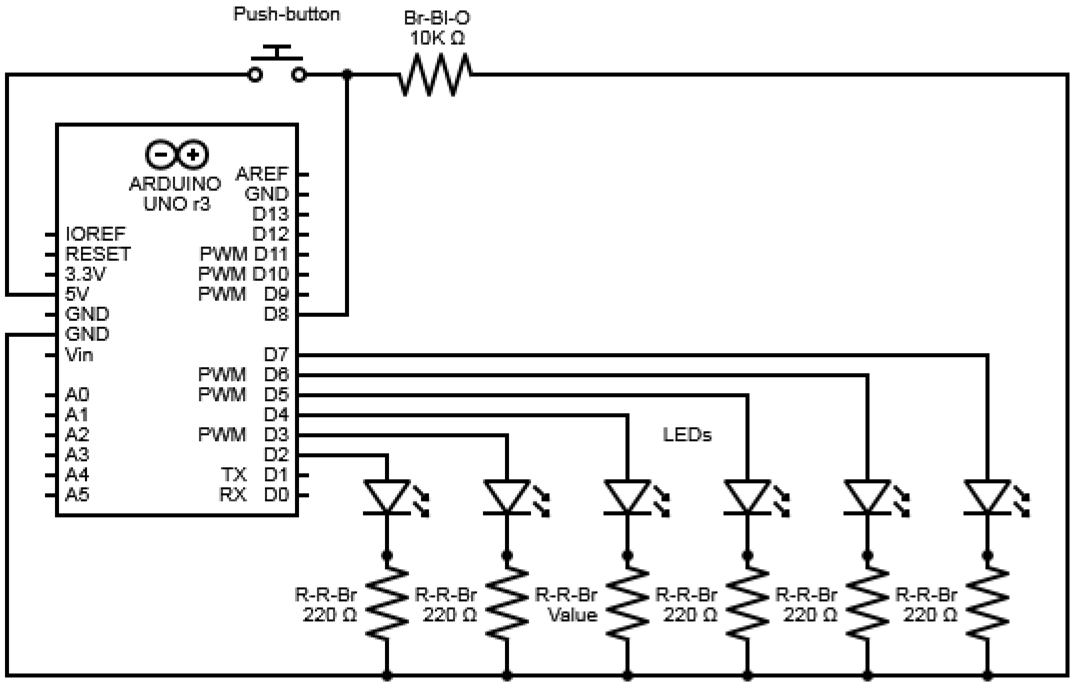
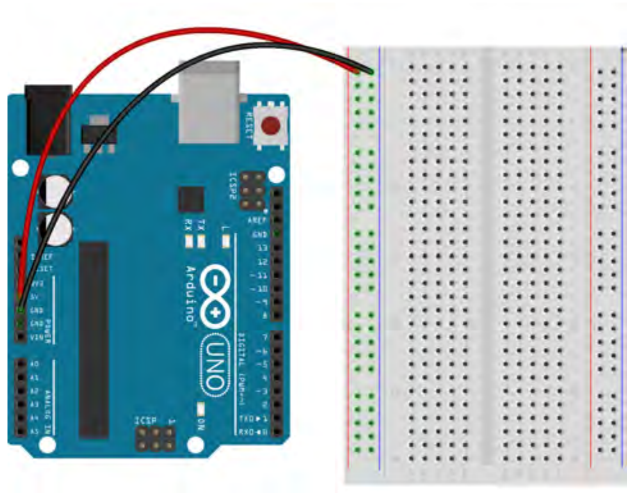


Figure 38: Digital Hourglass Circuit Diagram

First things first, connect the power and ground rails to '5V' and 'GND' using two wires.



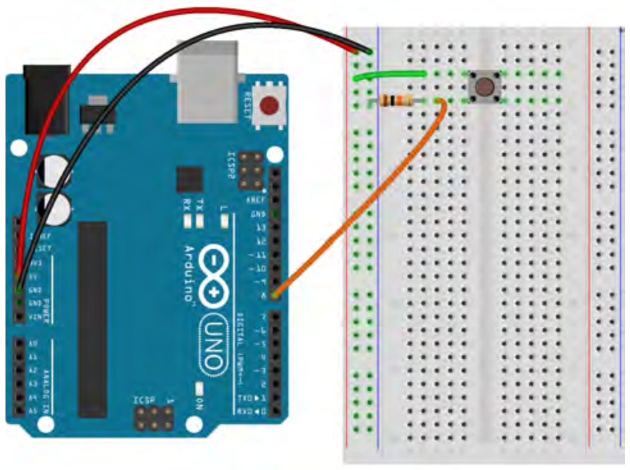
Step 1

Now, plug a push-button switch into the breadboard so that it is spanning the trench dividing the two sides of the breadboard.

Plug a wire into a row with one of the button leads and connect the other end to the power column of the breadboard.

Take a Brown-Black-Orange resistor. Plug one end into the row with the other push-button lead and plug the other end into the ground column of the breadboard.

Connect a wire to the same row that the resistor was just plugged into, and connect the other end to the digital pin 8 on the Arduino.

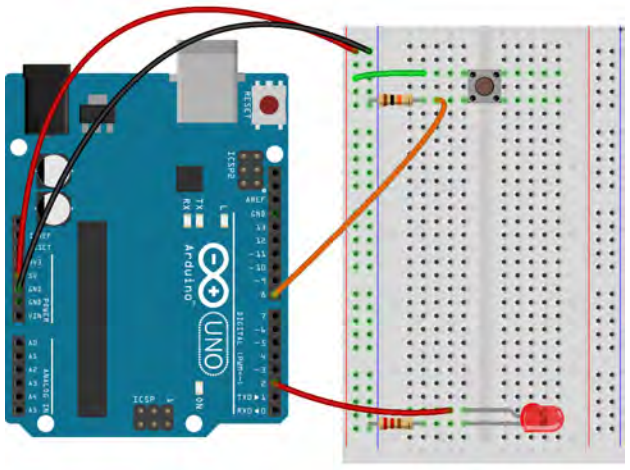


Step 2

Take an LED and plug it into the breadboard so that the leads do not share a row with each other or anything else. Remember that the longer lead is the positive one and so, if it hasn't been done already, put a kink into the longer lead before the LED is plugged in to mark it for later.

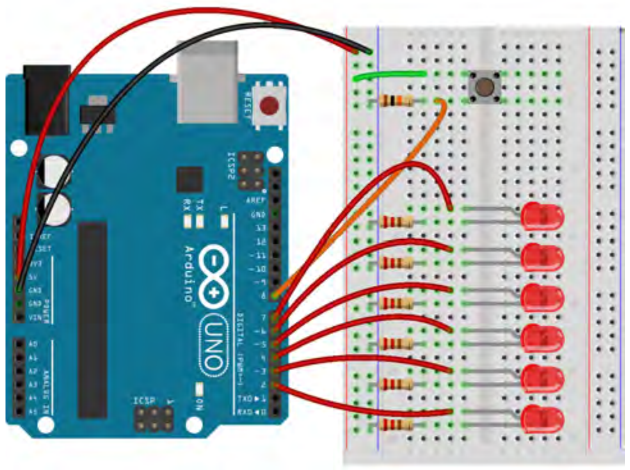
Plug one side of a Red-Red-Brown resistor into the negative, short lead of the LED, and plug the other end into the ground column.

Connect a wire to the row with the positive lead of the LED and to digital pin 2.



Step 3

Repeat this for the five other LEDs, each time moving up a pin.



Step 4

Pin	Code	Input	Output
D2	2		✓
D3	3		✓
D4	4		✓
D5	5		✓
D6	6		✓
D7	7		✓
D8	8	✓	

Table 14: Digital Hourglass Project Pin Assignment Chart

16.5 Writing the Code

The following is the code for this project:

```
// file: digitalHourglass.ch
// light up a sequence of LEDs one after the other at a specific time interval

#include <arduino.h>

//Declare pin assignment variable
const int switchPin = 8;

//Declare state, time, and varying pin assignment variables
int led = 2,
    currentSwitchState,
    previousSwitchState = 0,
    pinNumber,
    previousTime,
    currentTime,
    interval = 2000;

void setup(){
    //Set the modes of all I/O pins and set them low
    for (pinNumber = 2; pinNumber <8; pinNumber++) {
        pinMode(pinNumber, OUTPUT);
        digitalWrite(pinNumber, LOW);
    }

    pinMode(switchPin, INPUT);
}

void loop(){
    //Find the current time since the program started in milliseconds
    currentTime = millis();
    //When the time passed is greater than the interval, turn on the LED and move on to
    //the next on
    if ((currentTime-previousTime) > interval) {
        previousTime = currentTime;
        digitalWrite(led, HIGH);
        led++;
        //When all of the LEDs are on, blink the last LED twice then start over at
        //initial conditions
        if (led == 8){
            digitalWrite(7, LOW);
            delay(1000);
            digitalWrite(7, HIGH);
            delay(1000);
            digitalWrite(7, LOW);
            delay(1000);
            digitalWrite(7, HIGH);
            delay(1000);
            for (pinNumber = 2; pinNumber <8; pinNumber++) {
                digitalWrite(pinNumber, LOW);
            }
            led = 2;
            previousTime = millis();
        }
    }

    //Check to see if the switch has been pressed
    currentSwitchState = digitalRead(switchPin);

    //If the switch is pressed, start over at initial conditions
    if (currentSwitchState != previousSwitchState) {
        for (pinNumber = 2; pinNumber <8; pinNumber++) {
            digitalWrite(pinNumber, LOW);
        }
        led = 2;
        previousTime = currentTime;
    }

    //Set the current state to the previous state
    previousSwitchState = currentSwitchState;
}

int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

This program starts the same as the previous projects.

```
const int switchPin = 8;
```

```
int led = 2,
    currentSwitchState,
    previousSwitchState = 0,
    pinNumber,
    previousTime,
    currentTime,
    interval = 2000
```

The first variable to be created is a constant integer called 'switchPin' to hold on to the pin number that is connected to the push-button switch. Next, is to declare all of the regular integer variables. First is 'led' which will keep track of the LED that is going to be turned on next. The following two, 'currentSwitchState' and 'previousSwitchState', are fairly self-explanatory and will keep track of whether or not the button is pressed during the current iteration of the `loop()` function and whether or not it was pressed on the previous iteration. 'pinNumber' is a generic pin assignment variable that will be used to assign and turn on/off all of the LED pins. The last three deal with the timing of the LEDs. 'previousTime' and 'currentTime' will keep track of what the time since the beginning of the program currently and since the previous LED was lit up. Finally, 'interval' is the user-defined time period, in milliseconds, between the LEDs lighting up.

```
for (pinNumber = 2; pinNumber < 8; pinNumber++) {
    pinMode(pinNumber, OUTPUT);
    digitalWrite(pinNumber, LOW);
}

pinMode(switchPin, INPUT);
```

To have to write separate lines in order to set the pin mode and turn off every LED would be messy and tedious. This program avoids that by using a for loop to set the mode and turn off every LED in only a few lines of code. Remember, the first statement in the parenthesis of the for-loop is the initial condition. In this case, it sets the generic pin assignment variable 'pinNumber' to 2, the first LED pin. The second statement in the parenthesis, separated from the first by a semi colon, is the condition that, while true, makes the program keep repeating the loop. In this case, the loop will repeat itself until 'pinNumber' is not less than eight, or in other words, when 'pinNumber' is equal to 8. The last statement, separated from the second by a semicolon, is what happens at the end of every loop. In this case, 'pinNumber' is increased by one every loop. In short, this for-loop will run once for 'pinNumber' being from 2 to 7. The code inside of the loop itself sets 'pinNumber' to output mode and sets it to digital low, or zero volts. After the for-loop has set up each of the LED pins, the last line in the above segment sets 'switchPin' to input mode, and now all of the setup is complete.

```
void loop(){
    currentTime = millis();
```

The first thing the `loop()` function does is run the `millis()` function to get the time, in milliseconds, since the program began running and assigns this value to the 'currentTime' variable.

```
    if ((currentTime-previousTime) > interval) {
        previousTime = currentTime;
        digitalWrite(led, HIGH);
        led++;

        if (led == 8){
            digitalWrite(7, LOW);
            delay(1000);
            digitalWrite(7, HIGH);
            delay(1000);
            digitalWrite(7, LOW);
            delay(1000);
            digitalWrite(7, HIGH);
            delay(1000);
            for (pinNumber = 2; pinNumber <8; pinNumber++) {
                digitalWrite(pinNumber, LOW);
            }
            led = 2;
            previousTime = millis();
        }
    }
```

Now that the current time has been recorded, the program needs to see if the user-defined time interval has been reached. It does this in the condition of the first if-statement. If the difference between the current time and the previous time is greater than the time interval then the program enters the if-statement. Once inside the if-statement, the program sets the 'previousTime' variable equal to 'currentTime' so that the time difference checked for in the if-statement restarts, so to speak, every time the program enters that if-statement. Next, the program uses the `digitalWrite()` function to write a digital high, or 5 volts, to the current LED. In between the `digitalWrite()` functions are `delay()` funcions that pause the program for one second between blinks.

The program then increases the value of the 'led' variable by one so that the next time the program goes through the if-statement it will light up the next LED. Next there is another if-statement where the program checks to see if the LED variable is equal to 8, this would mean that the last LED, pin number 7, has been turned on and `led++;` increased the value to 8. So, the program will enter the second if-statement when all of the LEDs have been turned on. Once inside, the program turns the LED attached to pin 7 on and off twice, blinking it at an interval of one second. It then uses a for-loop similar to the one used earlier to turn all of the LEDs off so that the counter can start again. The program then sets the 'led' variable back to 2 and updates the 'previousTime' variable with the `millis()` function.

```
    currentSwitchState = digitalRead(switchPin);

    if (currentSwitchState != previousSwitchState) {
        for (pinNumber = 2; pinNumber <8; pinNumber++) {
            digitalWrite(pinNumber, LOW);
        }
        led = 2;
        previousTime = currentTime;
    }

    previousSwitchState = currentSwitchState;
```

The last segment of the code deals with the push-button switch and how the LEDs should react when it is pressed. This segment starts out by checking the state of the push-button using the `digitalRead()` function and assigning the result to the 'currentSwitchState' variable. If the button is pressed it will allow current to flow and the function will read a voltage, or digital high, and 'currentSwitchState' will be 1. After checking the push-button, the program has a if-statement with the condition that 'currentSwitchState' not be equal to 'previousSwitchState'. The `!=` operator is the 'not equal to' operator and will be true, or 1, if the left-hand and right-hand sides are not equal. So, this the program will enter this if-statement if the button is pressed and it was not the last time the program went through the **`loop()`** function, and vice versa. While only the first case is absolutely required, by checking if the button is not pressed and was, as well as if it is pressed and was not, the program makes sure that the timer for the LEDs starts when the button is released and not just when it is pressed. Inside of the if-statement is a for-loop that is the same as the for-loops as the two used earlier and will turn all of the LEDs off. After the for-loop the program resets the 'led' variable to 2 and sets the 'previousTime' equal to the 'currentTime'. The program now exits the if-statement and, lastly, sets the 'previousSwitchState' equal to the 'currentSwitchState'.

```
int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

The program ends by calling the **`setup()`** function once and repeatedly calling the **`loop()`** function an infinite number of times using an infinite while-loop.

When this code and this circuit are functioning correctly the LEDs should light up one after the other with a specific time delay between each. Once all of the LEDs are lit up then the last LED should blink twice and the process should start over again. If the push-button is pressed then the counting should start over again also. The time delay can easily be changed by increasing or decreasing the 'interval' variable.

16.6 Exercises

1. Modify the code and circuit to add another LED.
2. Replace the push-button with a photo-resistor and create a baseline variable where if the value from the photo-resistor exceeds it then the counter will start over.
3. Think of something else for the program to do when the counter is complete and all of the LEDs are lit up. Maybe a piezo goes off or they blink in a certain pattern.

17 Moving a DC Motor by Pressing a Button

17.1 Project 13: Motorized Pinwheel

Project Description: The motor will be controlled using a push-button switch, turning on when the switch is pressed and off when the switch is not pressed.

17.2 New Concepts

This project will introduce two important electrical components, the transistor and the diode, as well as a simple electric motor. By adding an electric motor to the repertoire of devices that can be controlled, it opens the door to controlling more powerful machines.

17.3 Required Components and Materials

- Breadboard
- 1 10kΩ (Br-Bl-O) Resistor
- 1 Push-Button Switch
- 1 Diode
- 1 IRF520 Transistor
- 1 DC Motor
- 1 9V Battery and Connector

17.4 Building the Circuit

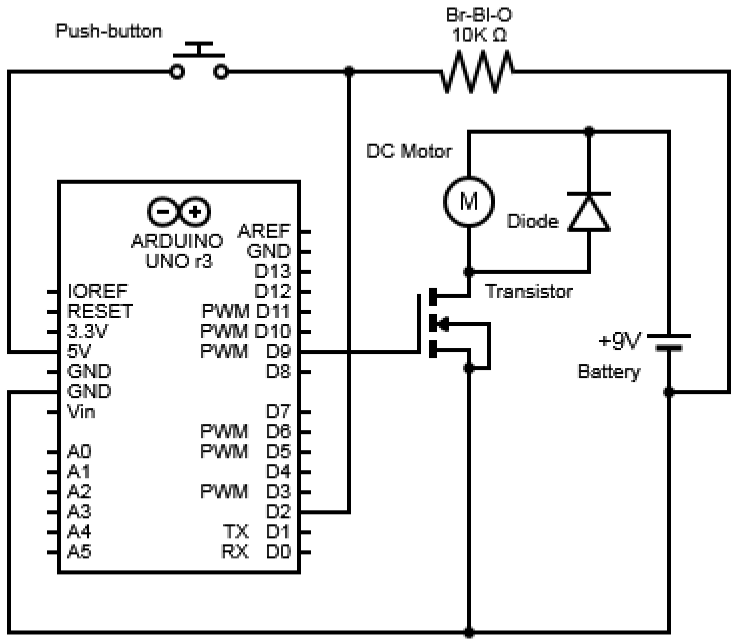
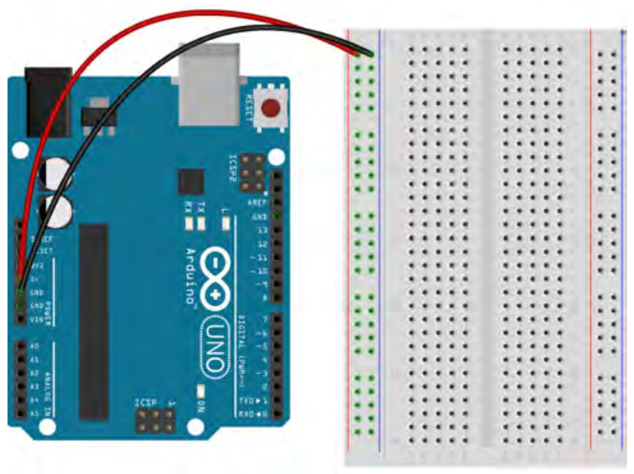


Figure 39: Motorized Pinwheel Circuit Diagram

As always, the first step is to set-up the power and ground wires.



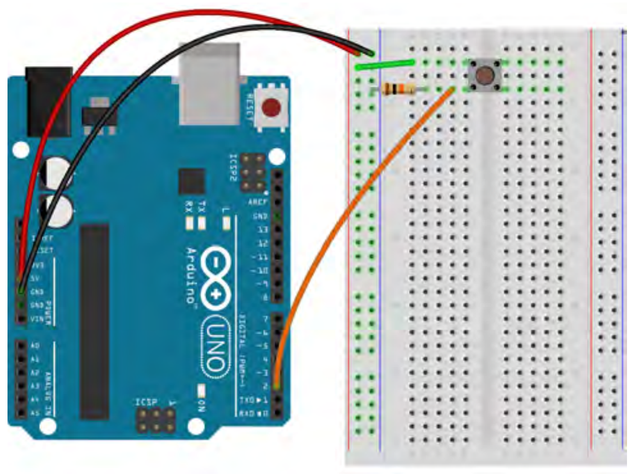
Step 1

Next, place a push-button on the breadboard so that it is spanning the trench in the center of the breadboard.

On one side of the breadboard, plug a wire into a row with one of the button leads and plug the other end into the power column.

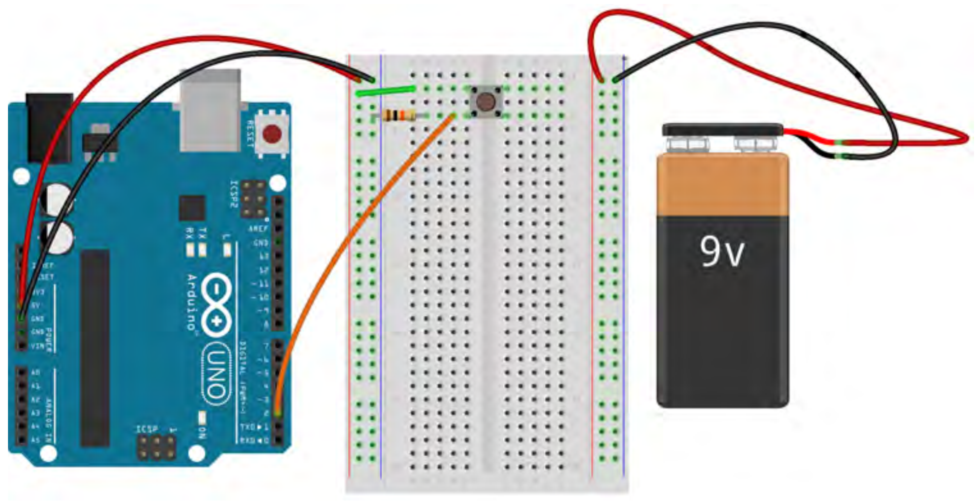
Take another wire and plug it into the row with the other lead of the push-button. Plug the other end into digital pin 2 on the Arduino.

Grab a Brown-Black-Orange resistor and connect one end to the row with the push-button lead that was just connected to pin 2. Plug the other end of the resistor into the ground column and the push-button will be set up.



Step 2

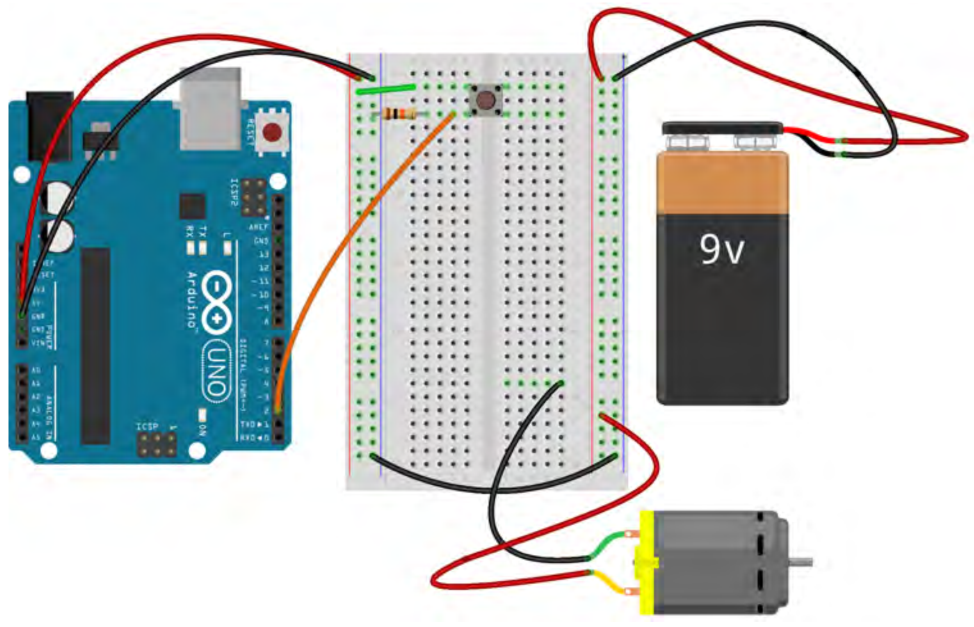
Now to set up the motor, connect a 9 volt battery to the battery snap, with the red and black wires coming off of it. Plug the red wire and black wires coming out of the battery snap into the opposite '+' and '-' columns of the ones that are already connected to the Arduino.



Step 3

Take the electric motor and there should be a red and black wire coming out of it. Plug the red wire into the same '+' column that the battery was just plugged into. Plug the black wire coming out of the motor into any row on the breadboard that is not already in use.

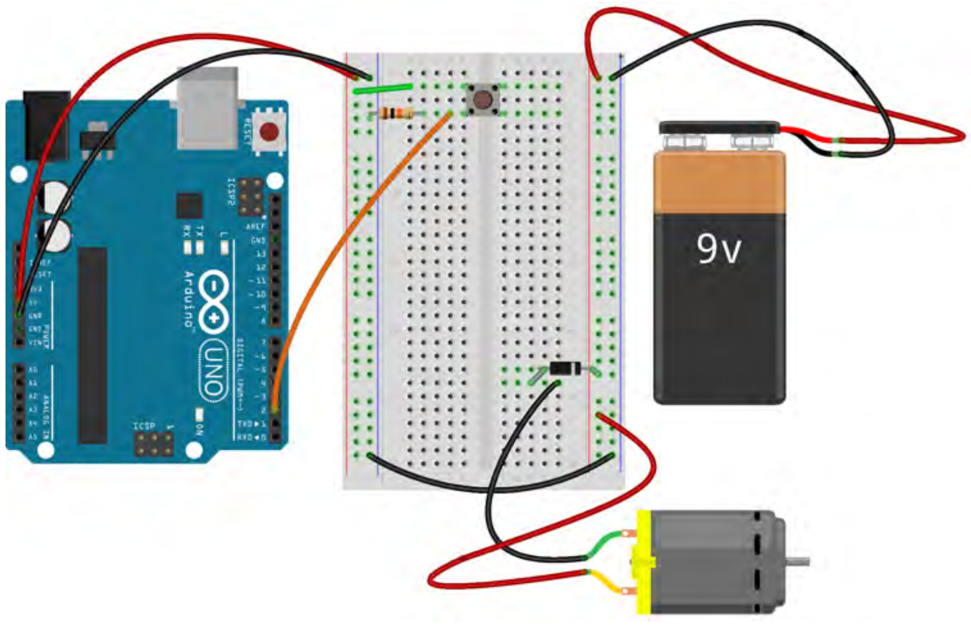
Connect a wire to between the two '-' columns, this creates a ground for the electric motor.



Step 4

Now, grab a diode. This should be a traditional diode which looks like a small black cylinder with a silver band around one end, this is the negative lead of the diode. Remember, a diode is an electrical component that lets current only flow in one direction, this direction being towards the side with the silver band. Plug the lead of the diode on the side with the silver band into the '+' column hooked up to the 9 volt battery. Plug the other end into the row that has the black wire from the motor.

When power is not being supplied to a motor, but it is still moving, it actually becomes an electricity generator. This diode, because it only allows current flow in one direction, will prevent any current generated by the motor from traveling backwards through the circuit and damaging anything.



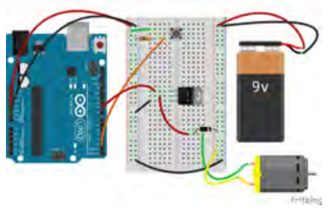
Step 5

Lastly, the transistor needs to be set-up. It is easy to think of transistors as electric switches. When a voltage is applied to one specific lead, it allows current to flow through the other two leads. Transistors can take all shapes and forms but the one that this program uses is called a MOSFET and looks like a little black box with three leads coming out of one end and a little metal lip coming out of the other end. Plug the transistor into the breadboard so that the three leads are in separate rows and do not share rows with anything else.

When looking at the transistor and facing the side with the writing on it, the left lead is called the gate and if transistors are electric switches then the gate is the button that is pressed to close the switch. Connect a wire to the row with the gate of the transistor and to digital pin 9 on the Arduino.

The other two leads of the transistor are called the source and the drain. Connect a wire from the drain, the middle pin of the transistor, to the row that has the black wire from the motor.

Lastly, connect the source, the remaining pin on the transistor, to the ground column of the Arduino. Now if the microcontroller sends a big enough signal to the gate then current can flow between the source and drain, turning the motor on.



Step 6

Pin	Code	Input	Output
D2	2	✓	
D9	9		✓

Table 15: Motorized Pinwheel Project Pin Assignment Chart

17.5 ChDuino Basic Test

Project 13 introduces an electric motor. Using the GUI, you can turn the motor on and off, and can control the speed that the motor spins at.

17.6 Writing the Code

The following is the code for this project:

```
// file: motorizedPinwheel.ch
// control a motor with a push-button switch

#include <arduino.h>

//Declare pin assignment variables
const int switchPin = 2,
          motorPin = 9;

//Declare a variable to track the state of the push-button switch
int switchState = 0;

void setup(){
    //Set the modes of the i/o pins
    pinMode(switchPin, INPUT);
    pinMode(motorPin, OUTPUT);
}

void loop(){
    //Read the input from the switch
    switchState = digitalRead(switchPin);

    //Turn on the motor if the button is pressed, and turn it off if it isn't pressed
    if(switchState == 1){
        digitalWrite(motorPin, HIGH);
    }
    else {
        digitalWrite(motorPin, LOW);
    }
}

int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

Once again, the program starts by including the header file.

```
const int switchPin = 2,
        motorPin = 9;

int switchState;

void setup(){
    pinMode(switchPin, INPUT);
    pinMode(motorPin, OUTPUT);
}
```

Two `const int` variables need to be created for the pin assignments, 'switchPin' for the pin connected to the push-button and 'motorPin' for the pin connected to the gate of the transistor. Next, a regular integer pin, called 'switchState', is created to record whether or not the button is pressed. The last part is the `setup()` function which uses the `pinMode()` function to let the microcontroller know that 'switchPin' is an input pin and 'motorPin' is an output pin.

```
void loop(){
    switchState = digitalRead(switchPin);

    if(switchState == 1){
        digitalWrite(motorPin, HIGH);
    }
    else {
        digitalWrite(motorPin, LOW);
    }
}
```

This code is fairly simple compared to some of the previous projects. First thing inside the `loop()` function, the program uses `digitalRead()` to read the push-button and the result is saved in 'switchState'. Next, there is an if-else statement. The if-statement checks to see if 'switchState' is equal to 1, meaning that the button is pressed. If it is then the program uses the `digitalWrite()` function to send a voltage to the pin connected to the gate of the transistor, letting current flow and turning the motor on. Lastly, the else-statement makes sure that if 'switchState' is anything else, meaning that the button is not pressed, then the program turns 'motorPin' off, not letting current to flow through the transistor and turning the motor off.

```
int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

The program ends by calling the **`setup()`** function once and repeatedly calling the **`loop()`** function an infinite number of times using an infinite while-loop.

If everything if working as it should then the motor should start to spin when the button is pressed and stop spinning when the button is released.

17.7 Exercises

1. Modify the circuit so that the motor spins in the opposite direction.
2. Modify the code so that the motor spins slower. Hint: PWM
3. Combine the light theremin project and this one and make a motor that spin faster or slower based on hand movements.
4. Modify the project by connecting a push-button to the Arduino and change the code so that pressing the button will start the motor and pressing it again will stop the motor.

18 Controlling the Speed and Direction of a DC Motor

18.1 Project 14: Zoetrope

Project Description: This project will expand upon controlling an electric motor as introduced in the last project. This time around there will be two push-button switches, one to control the direction that the motor spins and another to turn the motor on and off. The speed of the motor will be controlled by a potentiometer.

18.2 New Concepts

This program will introduce the H-bridge and integrated chips in general. Integrated chips, or ICs, are essentially pre-built circuits that come in a little silicon block with pins sticking out. The benefit of ICs are that, because they are prefabricated, they can provide the same circuit function in a much smaller space than it would take to build organically. The H-bridge is a specific type of IC that is used to change the polarity on a device, or switch the positive and negative power leads.

18.3 Required Components and Materials

- Breadboard
- 2 10kΩ (Br-Bl-O) Resistor
- 2 Push-Button Switch
- 1 Potentiometer
- 1 L293D H-Bridge IC
- 1 DC Motor
- 1 9V Battery and Connector

18.4 Building the Circuit

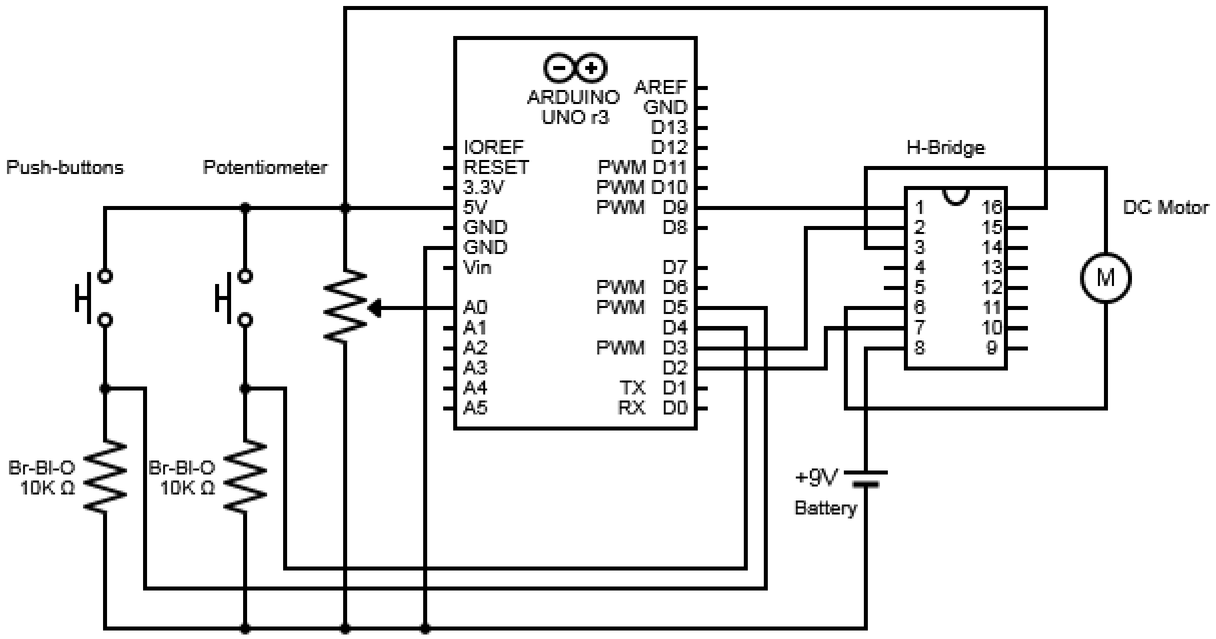
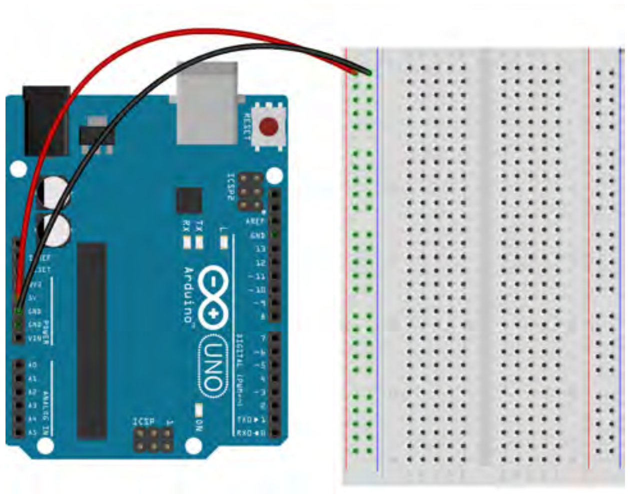


Figure 40: Zoetrope Circuit Diagram

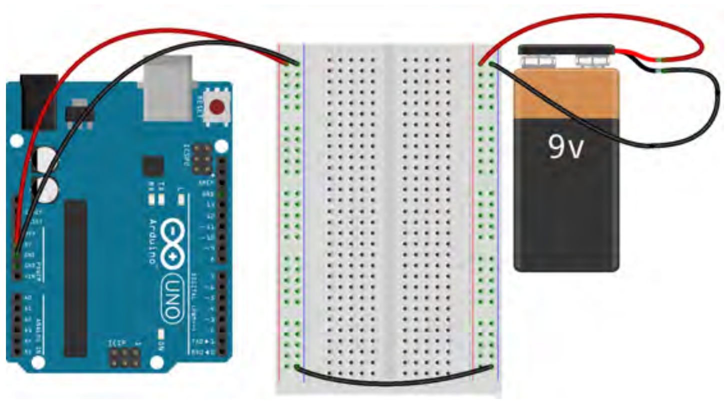
Connect the power and ground columns on the breadboard to the '5V' and 'GND' pins on the Arduino.



Step 1

Also, plug the red and black wires coming from the 9 volt battery connector into the other '+' and '-' columns, respectively.

Place a wire connecting the two '-' columns. First, place the push-button switches.



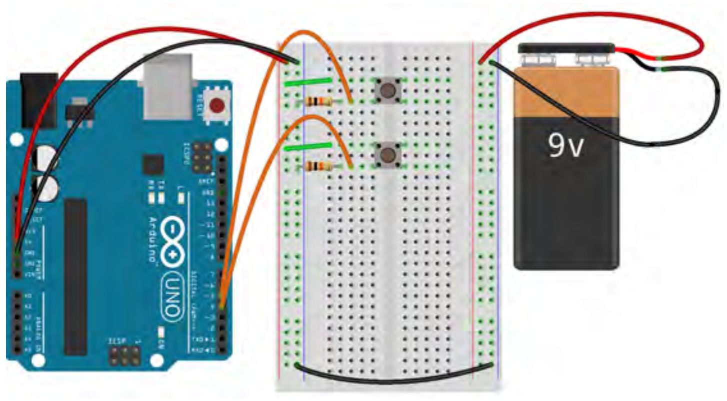
Step 2

Plug in two push-buttons so that they are spanning the trench in the middle of the breadboard. Have the buttons to one side of the breadboard, as there needs to be room left for the potentiometer and the H-bridge.

Use two wires and connect one pin of each button to the power column.

Now take two Brown-Black-Orange resistors and connect one to the row with the remaining lead of each button. Connect the other end of the resistor to the ground column.

Take two wires and plug one end of each into the two rows that the resistors were just plugged into. Take the other ends of the two wires and plug them into digital pins 4 and 5 on the Arduino.

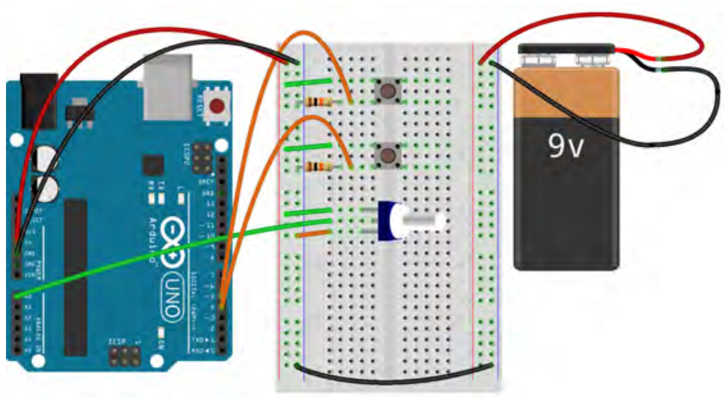


Step 3

On to the potentiometer. Place the potentiometer so that it is spanning the trench.

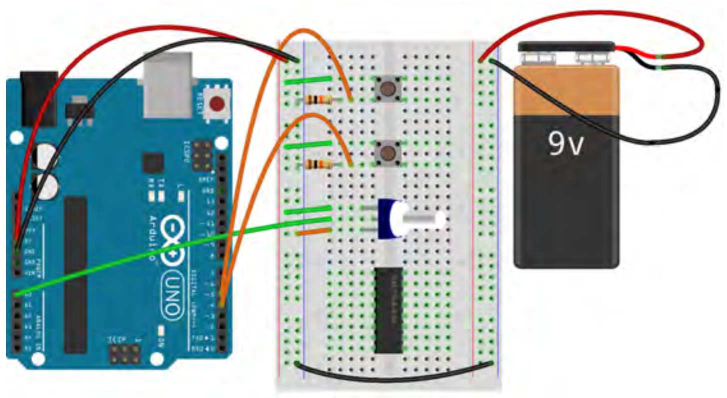
On the side of the potentiometer with two leads, connect one of these leads to power and one to ground, it does not matter which.

The other side of the potentiometer will only have one lead. Remember, this is the lead that the microcontroller will read. Connect this lead to analog pin A0 on the Arduino.



Step 4

Now it is time to place the H-bridge. There should be two rows of pins going along one side of the chip. Orient the chip so that it is in line with and also spanning the trench of the breadboard. The two rows of pins should be perfectly separated to plug into the rows of the breadboard on either side of the trench.



Step 5

Once the chip has been plugged in, notice that on one end of the top of the chip there is a little semi-circle cut out of the material. Each of the pins on the chip is numbered to make wiring it easier and the semi-circle helps orient the pin to make it easy to find which pin is which number. Looking at the chip so that this semi-circle is going to the left, the bottom-left pin is pin 1. The numbering then goes along the bottom of the chip so that the pin on the bottom-right is pin 8. Now the numbering goes up so that the pin on the top-right is pin 9. Finally, the numbering comes back to the left along the top of the chip so that the top, left, pin is pin 16. For the sake of simplicity, Figure 41 below shows the numbering of the pins for a 16 pin IC, as explained above, and Table 16 indicates what all of the those pins should be connected to.

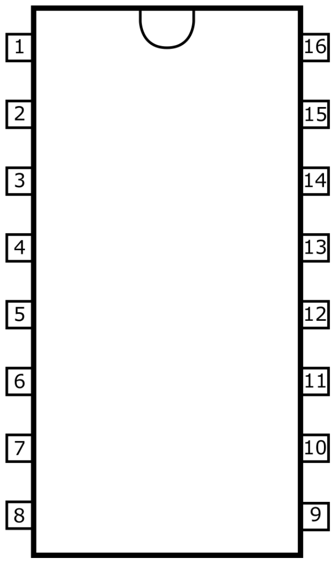
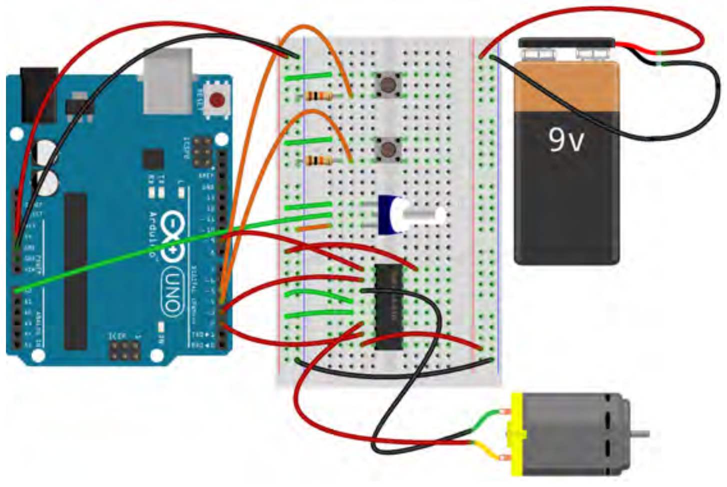


Figure 41: Pin Numbering Diagram for a 16 pin IC Chip

H-bridge Pin	**Connected To**
1	Arduino Pin 9
2	Arduino Pin 3
3	Motor Black Wire
4	Ground Column
5	Ground Column
6	Motor Red Wire
7	Arduino Pin 2
8	9 Volt Power
9	Nothing
10	Nothing
11	Nothing
12	Nothing
13	Nothing
14	Nothing
15	Nothing
16	5 Volt Power

Table 16: H-Bridge Pin Connection Chart



Step 6

Pin	Code	Input	Output
A0	A0	✓	
D2	2		✓
D3	3		✓
D4	4	✓	
D5	5	✓	
D9	9		✓

Table 17: Zoetrope Project Pin Assignment Chart

18.5 ChDuino Basic Test

This project introduces an integrated circuit. The integrated circuit is capable of more complex circuit manipulation, and allows the Arduino to change the direction of current for each pin.

18.6 Writing the Code

The following is the code for this project:

```
// file: zoetrope.ch
// Control a motor with push-buttons switches for on/off
// and direction and a potentiometer to control speed

#include <arduino.h>

//Declare pin assignment variables
const int directionControlPin1 = 2,
          directionControlPin2 = 3,
            enablePin = 9,
            directionSwitchPin = 4,
            onOffSwitchStatePin = 5,
            potPin = A0;

//Declare variables to indicate the different states
//of the motor and switches
int onOffSwitchState,
    previousOnOffSwitchState = 0,
    directionSwitchState,
    previousDirectionSwitchState = 0,
    motorEnabled = 0,
    motorSpeed,
    motorDirection = 1;

void setup(){
    //Set the mode for all of the i/o pins
    pinMode(directionControlPin1, OUTPUT);
    pinMode(directionControlPin2, OUTPUT);
    pinMode(enablePin, OUTPUT);
    pinMode(directionSwitchPin, INPUT);
    pinMode(onOffSwitchStatePin, INPUT);

    //Set the intial motor speed to zero
    digitalWrite(enablePin, LOW);
}

void loop(){
    //Check the state of the push-button switches
    onOffSwitchState = digitalRead(onOffSwitchStatePin);
    delay(100);
    directionSwitchState = digitalRead(directionSwitchPin);

    //Check the potentiometer and transform the value to the 0 to 255 scale
    motorSpeed = analogRead(potPin)/4;

    //If the state of the on/off button has changed and is high, change the
    //"motorEnabled" flag to its opposite so that it is high
    if(onOffSwitchState != previousOnOffSwitchState && onOffSwitchState == HIGH) {
        motorEnabled = !motorEnabled;
    }

    //If the state of the direction button has changed and is high, change the
    //"motorDirection" flag to its opposite so that it is high
    if(directionSwitchState != previousDirectionSwitchState && directionSwitchState == HIGH) {
        motorDirection = !motorDirection;
    }

    //If the "motorDirection" flag is high write the control pins to spin one direction,
    //if low write the opposite values to spin the motor in the other direction
    if(motorDirection == 1){
        digitalWrite(directionControlPin1, HIGH);
        digitalWrite(directionControlPin2, LOW);
    }
    else {
        digitalWrite(directionControlPin1, LOW);
        digitalWrite(directionControlPin2, HIGH);
    }

    //If the "motorEnabled" flag is high write the motor speed, on a scale of 0 to 255,
    //to the motor enable pin
    if(motorEnabled == 1){
        analogWrite(enablePin, motorSpeed);
    }
    else {
        analogWrite(enablePin, 0);
    }

    //Change the current state to the previous state
    previousDirectionSwitchState = directionSwitchState;
    previousOnOffSwitchState = onOffSwitchState;
}

int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

This code begins the same as all of the previous projects

```
const int directionControlPin1 = 2,
        directionControlPin2 = 3,
        enablePin = 9,
        directionSwitchPin = 4,
        onOffSwitchStatePin = 5,
        potPin = A0;

int onOffSwitchState,
    previousOnOffSwitchState = 0,
    directionSwitchState,
    previousDirectionSwitchState = 0,
    motorEnabled = 0,
    motorSpeed,
    motorDirection = 1;
```

There are a lot of pins and variables to assign in this program. The H-bridge uses two pins, connected to Arduino pins 2 and 3, to control the direction the motor spins, hence the pin assignment variables 'directionControlPin1' and 'directionControlPin2' are created to hold these values. The H-bridge uses one pin, connected to Arduino pin 9, to turn the motor on/off and to control its speed. The variable holding this pin assignment is called the 'enablePin'. This program has three inputs attached to pins, 'directionSwitchPin' for the push-button that controls which direction the motor spins, 'onOffSwitchStatePin' for the push-button that turns the motor on/off, and 'potPin' for the potentiometer controlling the speed of the motor. The rest of the variables, with the exception of 'motorSpeed', are flags. Remember, a flag is a variable that indicates whether something is true or not, whether an event happened or not, or if something is in one state or another. Flags are usually either 1 or 0, 1 for high or true and 0 for low or false. The first two `int` type variables, 'onOffSwitchState' and 'previousOnOffSwitchState', keep track of whether the push-button that turns the motor on/off is pressed or not. 'directionSwitchState' and 'previousDirectionSwitchState' keep track of whether the push-button that changes the motor's direction is pressed or not. The 'motorEnabled' flag keeps track of whether the motor should be on or off. 'motorDirection' keeps track of what direction the motor should be spinning in, equal to 1 for one direction and 0 for the other. Finally, 'motorSpeed' is the value that will be written to 'enablePin' to change the speed of the motor.

```
void setup(){
   pinMode(directionControlPin1, OUTPUT);
   pinMode(directionControlPin2, OUTPUT);
   pinMode(enablePin, OUTPUT);
   pinMode(directionSwitchPin, INPUT);
   pinMode(onOffSwitchStatePin, INPUT);

   digitalWrite(enablePin, LOW);
}
```

The next step is to create the `setup()` function and set the mode and initial condition for all of the pins. The first three lines in the segment above use the
`pinMode()` function to set all of the pins connected to the H-bridge, the two direction pins and the enable pin, to output mode. The next two lines set the pins connected to the two push-buttons to input mode. Remember that the potentiometer is connected to an analog pin, and analog pins are automatically set as inputs and do not require a `pinMode()` function. The last line writes a digital low, or 0 volts, to the enable pin of the H-bridge, which turns the motor off.

```
void loop(){
   onOffSwitchState = digitalRead(onOffSwitchStatePin);
   delay(100);
   directionSwitchState = digitalRead(directionSwitchPin);

   motorSpeed = analogRead(potPin)/4;
```

Once inside of the `loop()` function, the program reads the three input pins. The first two pins are connected to the push-buttons, hence they are digital inputs, and are read with the `digitalRead()` function. The results of these reads, 1 if the button is pressed and 0 if it is not, are stored in the each button's flag variable. The program pauses for one hundred milliseconds between the reads, using the `delay()` function, in order to give the program enough time to execute the read function. The last line reads the level of the pin connected to the potentiometer with the `analogRead()` function. Remember that values read from analog pins are on the 0 to 1023 scale, but analog values that are going to be written need to be on the 0 to 255 scale. So, the last line transforms the raw value from the potentiometer pin to the 0 to 255 scale by dividing it by 4 and then assigning it to 'motorSpeed', saving a step over previous projects.

```
   if(onOffSwitchState != previousOnOffSwitchState && onOffSwitchState == HIGH) {
      motorEnabled = !motorEnabled;
   }
```

```
    if(directionSwitchState != previousDirectionSwitchState &&
        directionSwitchState == HIGH) {
        motorDirection = !motorDirection;
    }
```

After the inputs have been read, the program needs to use these inputs and see if it needs to do anything new. The first step in doing this is updating the 'motorDirection' and 'motorEnabled' flags. The first line has an if-statement with the condition that 'onOffSwitchState' not be equal to 'previousOnOffSwitchState', meaning that the button is pressed or not pressed when it was the opposite the last time through the `loop()` function, and 'onOffSwitchState' must be equal to 1, meaning the button has been pressed. So, if the on/off button is pressed, and did not used to be pressed, then the program enters the if-statement. Once inside, the program sets the 'motorEnabled' flag to its opposite. The `!` operator in this case means opposite, so if 'motorEnabled' is 0 is becomes 1 and vice versa. The program then moves on to a similar if-statement, only this time with the flag variables for direction. So, if the button that controls the motors direction is pressed, and it wasn't on the previous iteration of the `loop()` function, then the program enters the if-statement. Once inside, the program sets the 'motorDirection' flag to it opposite, '!motorDirection'.

```
    if(motorDirection == 1){
        digitalWrite(directionControlPin1, HIGH);
        digitalWrite(directionControlPin2, LOW);
    }
    else {
        digitalWrite(directionControlPin1, LOW);
        digitalWrite(directionControlPin2, HIGH);
    }
```

Now that the flag variables have been updated, the program needs to make use of that information to give commands to the H-bridge in order to control the motor. First, the commands to change the direction the motor is spinning. The first line starts an if-else statement where the condition of the if-statement is that the 'motorDirection' flag is equal to 1. If this is the case, then the program uses the `digitalWrite()` function to set the first direction control pin on the H-bridge to high and set the second pin to low. This will spin the motor in a one direction. If 'motorDirection' is not 1, the program enters the else-statement and sets the first direction pin to low and the second to high, causing the motor to spin in the other direction.

```
    if(motorEnabled == 1){
        analogWrite(enablePin, motorSpeed);
    }
    else {
        analogWrite(enablePin, 0);
    }
```

Next, the program uses the information from the 'motorEnable' flag to turn the motor on or off. The first line starts an if-else statement where the condition of the if-statement is that the 'motorEnabled' flag be equal to 1. If this is true, the program enters the if-statement and uses the `analogWrite()` function to send the motor speed to the enable pin of the H-bridge. If 'motorEnabled' is anything other than 1 then the program will enter the else-statement and write a speed of 0 to the enable pin, turning the motor off.

```
    previousDirectionSwitchState = directionSwitchState;
    previousOnOffSwitchState = onOffSwitchState;
}
```

Lastly, the program sets the previous direction and on/off state flags equal to the current ones. This is so the program can remember if the button was pressed during the previous time going through the `loop()` function, and is used in the earlier if-statements.

```
int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

The program ends by calling the `setup()` function once and repeatedly calling the `loop()` function an infinite number of times using an infinite while-loop.

If the circuit and program are running correctly, then one of the push-buttons should turn the motor on and off, the other button should switch the direction the motor spins, and turning the potentiometer should speed up and slow down the motor.

18.7 Exercises

1. Wire two different color LEDs. One color is turned on when the motor spins one direction, and the other color lights up when the motor turns in the other direction.

19 Writing on an LCD Panel

19.1 Project 15: Crystal Ball

Project Description: When the program starts, the LCD screen will display a prompt for the user to ask it a question. The user tilts a tilt sensor and the LCD screen will then display a random response.

19.2 New Concepts

This project will introduce a lot of new things, including the most complicated output device yet, the LCD screen. LCD stands for Liquid Crystal Display where liquid crystals are a special material that can flow like a liquid, but its atoms are structured like a crystal. When a current is applied to a liquid crystal it changes its shape, allowing different colors of light to pass and forming the image. This project will also introduce many functions for controlling the LCD as well as the switch-statement for coding and a new type of switch called a tilt sensor.

New Functions

- `LiquidCrystal(int rsPin, int enablePin, int dataPin1, int dataPin2, int dataPin3, int dataPin4)`
- `void LiquidCrystal.begin(int column, int row, int dotSizeMode)`
- `void LiquidCrystal.print("String")`
- `void LiquidCrystal.setCursor(int column, int row)`
- `void LiquidCrystal.clear(void)`
- `int random(int min, int max)`
  or `int random(int max)`

19.3 Required Components and Materials

- Breadboard
- 1 220Ω (R-R-Br) Resistor
- 1 10kΩ (Br-Bl-O) Resistor
- 1 Tilt Sensor
- 1 Potentiometer
- 1 LCM1602C LCD Panel

19.4 Building the Circuit

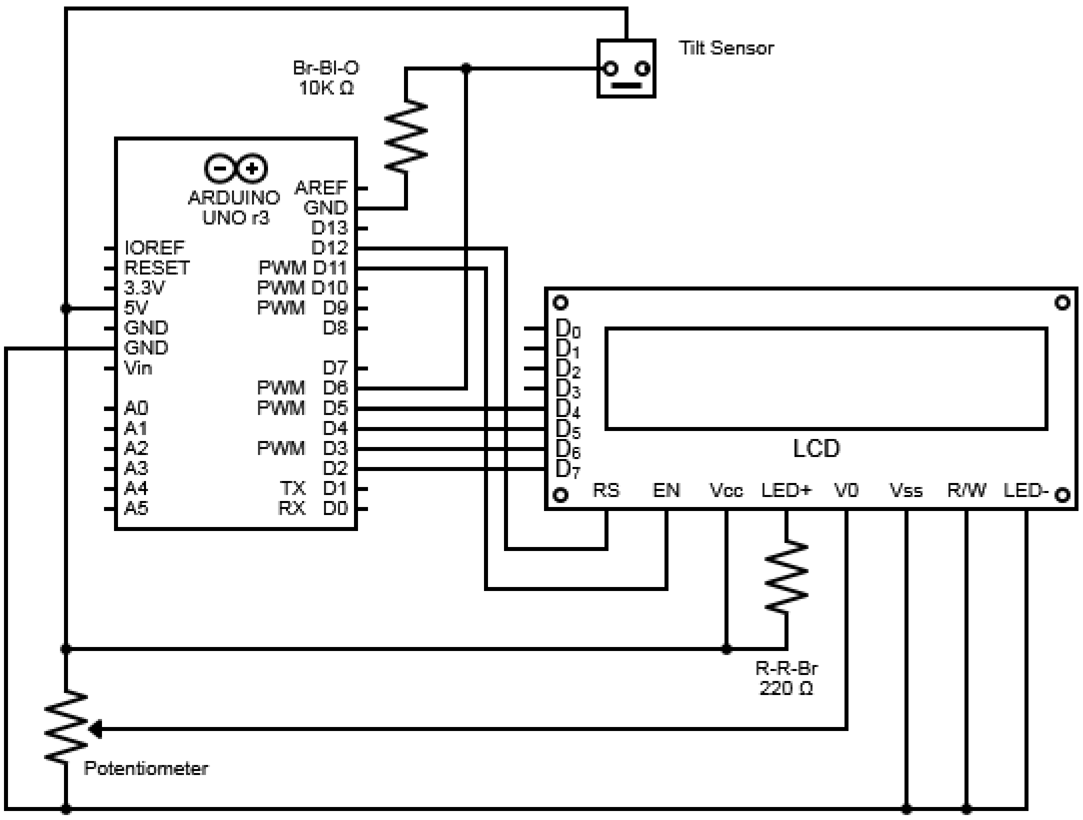
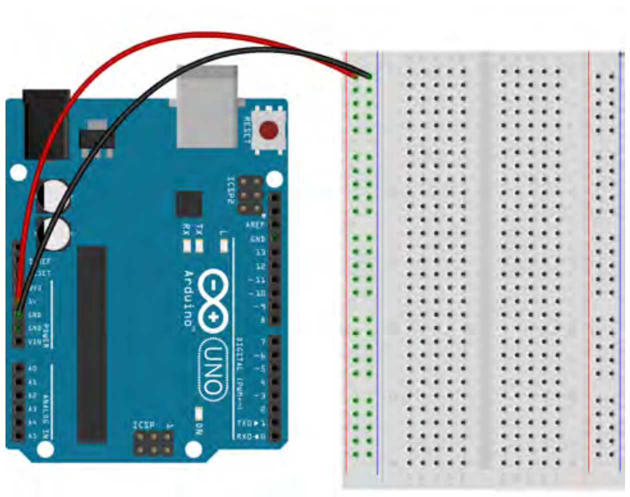


Figure 42: Crystal Ball Circuit Diagram

As usual, the first step in the build is to connect the power and ground columns of the breadboard to the Arduino.



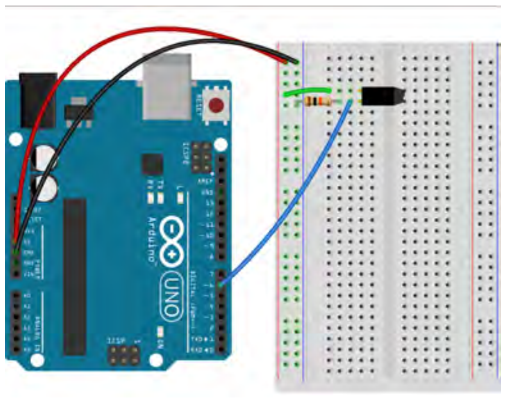
Step 1

The next step is to connect the tilt sensor. The tilt sensor is a black rectangular prism with four small leads sticking out of one end. There is a little metal ball inside of the sensor, and when the sensor is right way up the ball will sit on two little metal pads, letting current pass. When the sensor is tilted, the metal ball rolls off of the pads and current ceases to flow. Plug the leads into the rows of the breadboard.

Connect a wire from the row with one of the tilt sensor leads to the power column.

Take a Brown-Black-Orange resistor and connect one end to the row with the other leads, the ones not connected to anything, of the tilt sensor. Connect the other end of the resistor into the ground column.

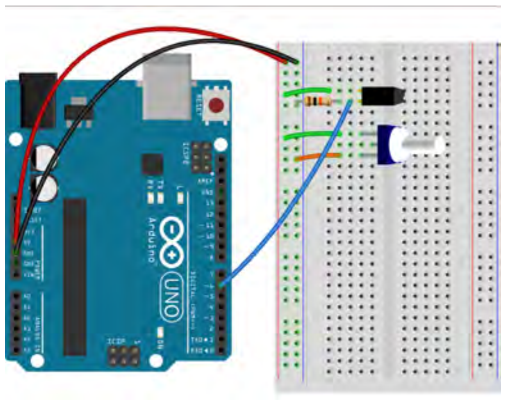
Now, plug a wire into the same row that the resistor was just plugged into and plug the other end into digital pin 6 on the Arduino.



Step 2

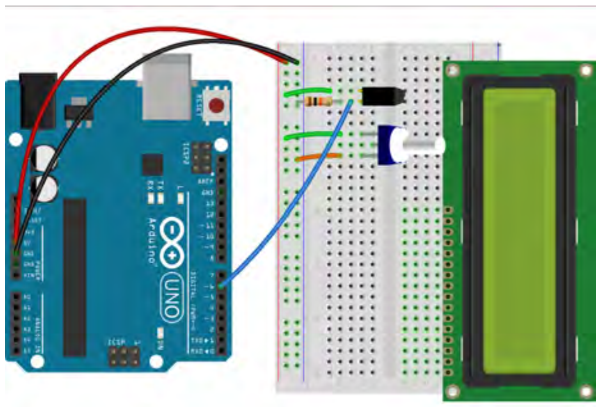
Grab a potentiometer and plug it into the breadboard so that it is spanning the trench in the middle of the board. This potentiometer will be used to adjust the contrast of the display.

One side will have two leads. Connect one of these leads to power and the other to ground, it does not matter which. The other lead of the potentiometer will be connected to pin 3 of the LCD later.



Step 3

Now to wire the LCD screen. On the back of the LCD there should be a row of sixteen leads, plug these leads into one side of the breadboard so that each lead has its own row.

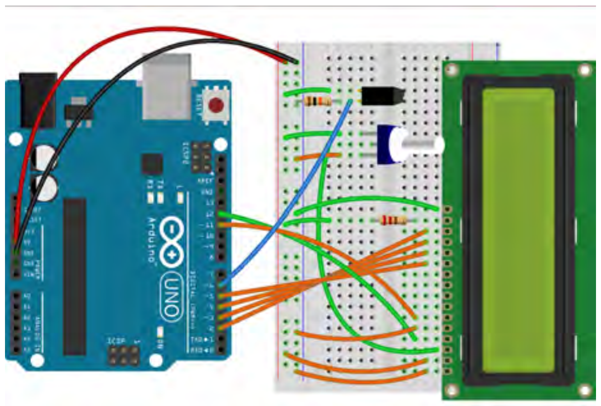


Step 4

For the sake of simplicity Table 18 below indicates what all of the LCD pins should be connected to. On the LCD, where the pins come out, there should be a 1 and a 16 on either side of the pins, marking the direction of the numbering. Pin 15 is connected to a Red-Red-Brown resistor which is then connected to power.

LCD Pin	Connected To
1 (Vss)	Ground Column
2 (Vcc)	Power Column
3 (V0)	Potentiometer
4 (RS)	Digital Pin 12
5 (R/W)	Ground Column
6 (EN)	Digital Pin 11
7 (D0)	Nothing
8 (D1)	Nothing
9 (D2)	Nothing
10 (D3)	Nothing
11 (D4)	Digital Pin 5
12 (D5)	Digital Pin 4
13 (D6)	Digital Pin 3
14 (D7)	Digital Pin 2
15 (LED+)	Resistor → Power
16 (LED-)	Ground

Table 18: LCD Screen Pin Connection Chart



Step 5

Pin	Code	Input	Output
D6	6	✓	

Table 19: Crystal Ball Project Pin Assignment Chart

19.5 Writing the Code

The following is the code for this project:

```
// file: crystalBall.ch
// write characters to an LCD display

#include <arduino.h>

//Declare the pin assignment variable for the switch
const int switchPin = 6;

//Declare variables to keep track of the state of the switch and to hold the random
//number that determines the programs reply
int switchState,
    prevSwitchState,
    reply;

//Create an instance of a LiquidCrytal class, called lcd and
//Initialize pin assignments for the LCD for 4 bit mode
//First argument is the RS pin
//Second argument is the Enable pin
//The third though sixth arguments are the data pins
LiquidCrystal lcd = LiquidCrystal(12, 11, 5, 4, 3, 2);

void setup(){
    //Turn on the LCD for 16 columns and 2 rows and the standard dot size
    lcd.begin(16,2,0);

    //Set the switch pin mode to input
    pinMode(switchPin, INPUT);

    //Print to the first row of the LCD, change the position to the second row,
    // print to the second row of the LCD
    lcd.print("Ask the");
    lcd.setCursor(0,1);
    lcd.print("Crystal Ball...");
}

void loop(){
    //Read the state from the switch pin, if it has changed and is now high then begin
    //the reply process
    switchState = digitalRead(switchPin);

    if (switchState != prevSwitchState){
        if (switchState == LOW) {

            //create a random integer between 0 and 7
            reply = random(8);

            //Clear the LCD screen, set it to the first row and print, set it to the
            //second row and print a certain string based on the random integer number
            lcd.clear();
            lcd.setCursor(0,0);
            lcd.print("The ball says:");
            lcd.setCursor(0,1);
            switch (reply){
                case 0:
                    lcd.print("Yes");
                    break;
                case 1:
                    lcd.print("Most Likely");
                    break;
                case 2:
                    lcd.print("Certainly");
                    break;
                case 3:
                    lcd.print("Outlook Good");
                    break;
                case 4:
                    lcd.print("Unsure");
                    break;
                case 5:
                    lcd.print("Ask Again");
                    break;
                case 6:
                    lcd.print("Doubtful");
                    break;
                case 7:
                    lcd.print("No");
                    break;
            }
        }
    }
    //Set the current switch state to the previous switch state
    prevSwitchState = switchState;
}

int main(){
    setup();
    while(1) {
        loop();
    }
    return 0 ;
}
```

All of the functions that control the LCD are already inside of the `arduino.h` header file so nothing needs to be changed to the first section of the code.

```
const int switchPin = 6;

int switchState,
    prevSwitchState,
    reply;
```

As the Arduino pin connected to the tilt sensor is the only one not connected to the LCD, it is the only one that needs a pin assignment variable, in this case called ’switchPin’. The pins connected to the LCD will be handled by a special function in the next code segment. Three other variables are declared. ’switchState’ and ’prevSwitchState’ are flags that keep track of whether or not the tilt sensor is on, and ’reply’ will hold a random number that will determine the response that is displayed on the LCD.

```
LiquidCrystal lcd = LiquidCrystal(12, 11, 5, 4, 3, 2);

void setup(){
    lcd.begin(16,2,0);

    pinMode(switchPin, INPUT);
```

The first line in the segment above creates and instance of the LiquidCrystal class, called ’lcd’. The `LiquidCrystal` class contains all of the special functions for controlling the LCD screen. Notice that this declaration of a class variable is different than the previous examples in this book, this one is set equal to a function with arguments. This declaration uses what is called a constructor to take information into ’lcd’ as it is being created. A constructor is like a member function that is the same name as the class and is run when the class instance is created. The generic form of this constructor is shown below.

```
LiquidCrystal(int rsPin, int enablePin, int dataPin1, int dataPin2, int dataPin3,
int dataPin4)
```

The Arduino uses this constructor to assign and set the pin mode if the pins connected to the LCD. The rs pin, pin 4 on the LCD, is what controls where the words will print out on the screen. The enable pin, pin 6 on the LCD, lets the LCD know that it is going to receive a command from the Arduino. The data pins, pins 11-14 are how the LCD receives information on what character it is going to display. There are actually eight data pins but only four are required and only four are used in this project. After initializing the pins, the program then defines the `setup()` function and starts the LCD with the `begin()` member function of the `LiquidCrystal` class.

```
void LiquidCrystal.begin(int column, int row, int dotSizeMode)
```

The first argument is the number of columns in the display, in this case 16, the second argument is the number of rows, in this case 2, and the last argument is the mode for the size of the dots that make up the characters on the screen, in this case the mode is zero which is the default. Lastly, the program with the last line sets the pin mode for the pin connected to the tilt sensor to input mode.

```
    lcd.print("Ask the");
    lcd.setCursor(0,1);
    lcd.print("Crystal Ball...");
}
```

The next step in the code is to print out the initial prompt to the LCD. The program does this with the `print()` function. The program then closes the `setup()` function with a '}'.

```
void LiquidCrystal.print("String")
```

The argument for this function is a string of characters to be printed, surrounded by quotations. After printing the first line `"Ask the"` the program needs to change where the cursor on the LCD is positioned so that the next print function will print `"Crystal Ball.."`to the next line on the LCD. This is done with the `setCursor()` function.

```
void LiquidCrystal.setCursor(int column, int row)
```

This function will tell the LCD to start printing at any column and row combination. Note, the numbering for the columns and rows starts at zero and then goes up, so the first row is row 0.

```
void loop(){
   switchState = digitalRead(switchPin);

   if (switchState != prevSwitchState){
      if (switchState == 1) {
```

The first thing that the program does inside of the `loop()` function is to use `digitalRead()` to see if the tilt sensor has been tilted and save it into the 'switchState' flag. This is followed by two if-statements that the program will only enter if the state of the tilt sensor is different than it was on the previous iteration of the `loop()` function and if the 'switchState' flag is 1, meaning that the tilt sensor is right side up.

```
            reply = random(8);

            lcd.clear();
            lcd.setCursor(0,0);
            lcd.print("The ball says:");
            lcd.setCursor(0,1);
```

Once inside of the two if-statements, the program creates a random number between 0 and 7 using the `random()` function.

```
int random(int max)
```

The function will return a random number between 0 and the number 'max', not including 'max'. The `random()` function also has an optional argument to set the minimum value, so the code below is valid.

```
int random(int min, int max)
```

This number is assigned to the 'reply' variable which will be used later in the code. The program then uses the `clear()` function to erase anything on the LCDs screen. The generic form of the `clear()` function is shown belown.

```
void LiquidCrystal.clear(void)
```

It then uses the setCursor() function to move the cursor to the first column and first row, prints the "The Ball Says:" string, and moves the cursor again, this time to the second row.

```
            switch (reply){
                case 0:
                    lcd.print("Yes");
                    break;
                case 1:
                    lcd.print("Most Likely");
                    break;
                case 2:
                    lcd.print("Certainly");
                    break;
                case 3:
                    lcd.print("Outlook Good");
                    break;
                case 4:
                    lcd.print("Unsure");
                    break;
                case 5:
                    lcd.print("Ask Again");
                    break;
                case 6:
                    lcd.print("Doubtful");
                    break;
                case 7:
                    lcd.print("No");
                    break;
            }
```

The code segment above comprises what is called a switch-statement. A switch-statement takes a variable and, depending on what value that variable has, will execute a certain piece of code. The variable within the parenthesis is the variable that the switch statement will depend on. Given a certain value for the variable, in other words in the case that the variable is a certain value, the program will execute some particular code then leave the switch-statement with a break statement. So, in the case of the code segment above the variable that the switch-statement depends upon is the 'reply' variable which was created as a random number between 0 and 7. Within the switch-statement braces are the different code segments that will run depending upon what random number 'reply' is. Take the second line to the fourth line; In the case that 'reply' is equal to 0, the program will print out the string "Yes". It will then break out of the switch-statement and move on in the code with the break statement. Likewise, the next three lines will print out "Most Likely" for the case that 'reply' is equal to 1. The next three lines will print out something else for the case 'reply' is 2 and the switch-statement continues in this manner. So, the switch statement will print out different strings to the LCD screen depending on the random variable 'reply'.

```
        }
    }
    prevSwitchState = switchState;
}
```

Finally, the last segment of code closes the open if-statements and then the loop() function, but not before setting the 'previousSwitchState' flag equal to the current 'switchState'. This will record whether or not the tilt sensor has been activated on this pass through the **loop()** function so that information can be given to the if-statements on the next pass through the **loop()** function.

```
int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

The program ends by calling the **setup()** function once and repeatedly calling the **loop()** function an infinite number of times using an infinite while-loop.

The program should initially print out a prompt for the user to ask the ball something. Then, when the tilt sensor is activated, the LCD should randomly print one of eight responses. Note that the LCD contrast will likely need to be adjusted. Do this by slowing turning the potentiometer all the way throughout its range until something can be seen on the screen.

19.6 Exercises

1. Try using a if-elseif-else statement instead of the switch statement. Which is best?
2. Modify the code to use only one if-statement instead of two using the logical and operator.
3. Add a couple new strings that the LCD can respond with.

20 Using a Piezo as a Vibration Sensor

20.1 Project 16: Knock Lock

Project Description: This project will create a system that will turn servo to a ”locked” position when a push-button is pressed and will stay it that position until the user knocks three times, using the piezo to detect the vibrations of the knocks. The user should knock the same surface that the piezo is resting on. If three valid knocks are detected then the servo will move to the ”unlocked” position.

20.2 New Concepts

In the process of building this project, a new variable type, `bool` will be introduced as well as how to create new functions.

20.3 Required Components and Materials

- Breadboard
- 3 220Ω (R-R-Br) Resistor
- 1 10kΩ (Br-Bl-O) Resistor
- 1 1MΩ (Br-Bl-G) Resistor
- 1 100μF Capacitor
- 1 Push-Button Switch
- 3 LED
- 1 Piezo
- 1 Servo

20.4 Building the Circuit

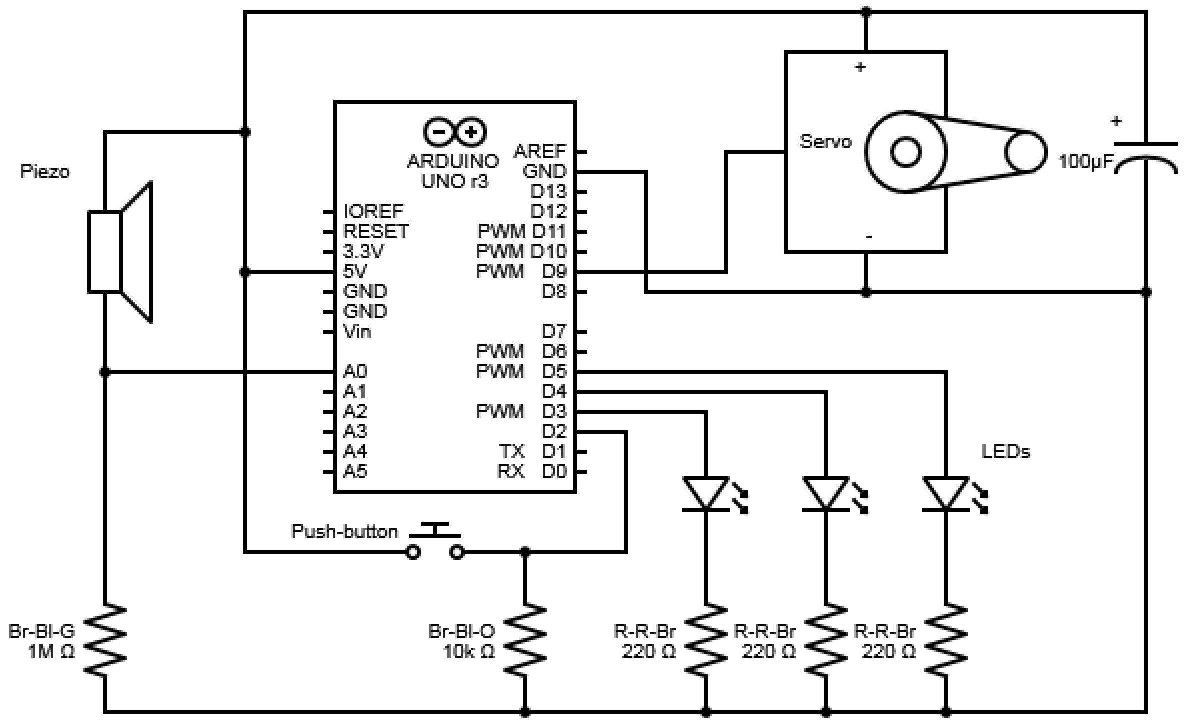
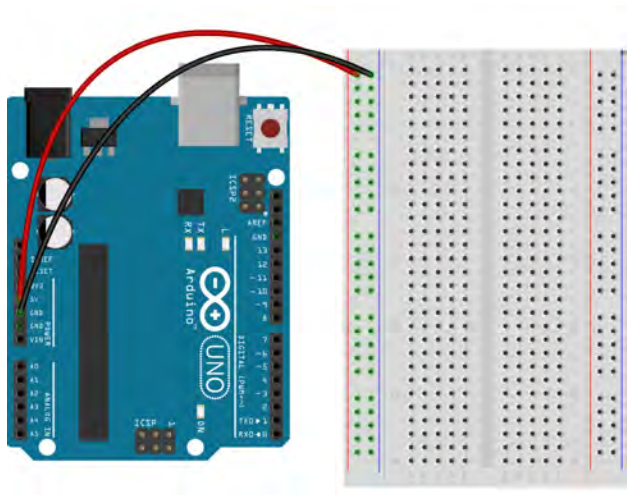


Figure 43: Knock Lock Circuit Diagram

This circuit has a lot of components, so let's get into it. First, connect wires from the power and ground columns on the breadboard to the '5V' and 'GND' pins on the Arduino, respectively.



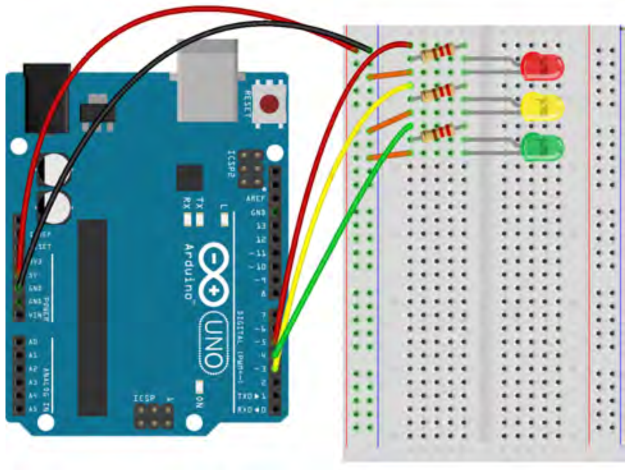
Step 1

Now, to wire the LEDs. On one side of the breadboard plug in a green, yellow, and red LED so that all of the leads have their own row.

For each of the LEDs attach a wire to the row with the negative, shorter, lead of the LED and plug the other end of the wire into the ground column.

Again for all of the LEDs, plug one end of a Red-Red-Brown resistor into the row with the positive, longer, lead of the LED. Plug the other end of the resistors into rows that do not have anything else plugged into them.

Now, take three wires and plug one end of each into the rows that the resistors were just plugged into, the rows that do not have LED leads. The other ends of the wires belong in the digital pins on the Arduino. The wire connected to the red LED should be plugged into digital pin 5, the wire connected to the green LED should be connected to digital pin 4, and the wire connected to the yellow LED should be connected to digital pin 3.

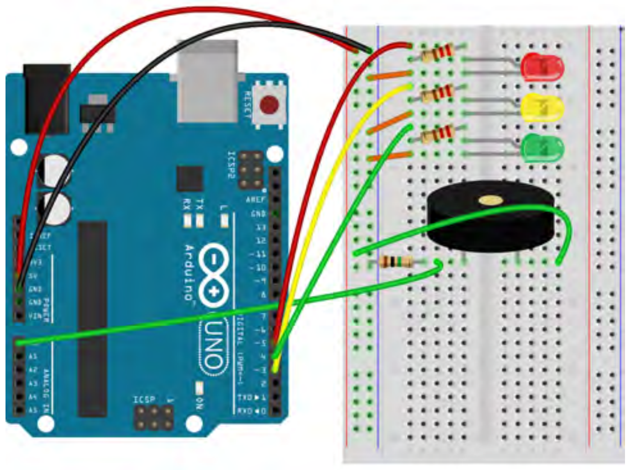


Step 2

Next, the piezo will be wired. Plug the piezo into the breadboard so that both of the leads have their own rows. Plug one end of a wire into the row with one of the piezo leads, it does not matter which one, and plug the other end of the wire into the power column.

For the other piezo lead, plug one end of a Brown-Black-Green resistor into its row and connect the other end of the resistor to the ground column.

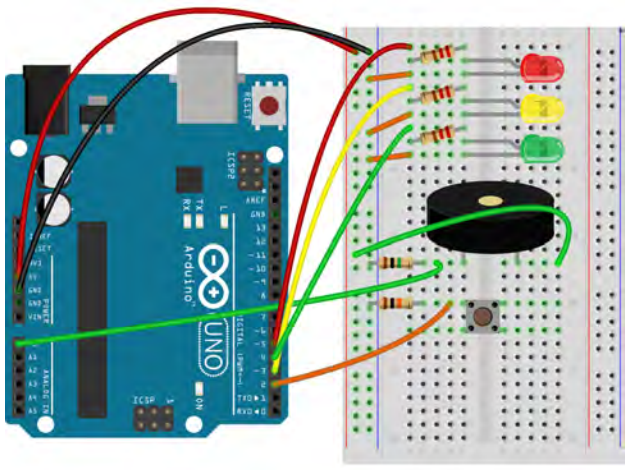
Next, connect one end of a wire into the same row that the piezo lead and the resistor are connected to and connect the other end of the wire to analog pin A0 on the Arduino.



Step 3

Time to wire the push-button switch that will put the servo into the ”locked” position. Plug a push-button switch into the breadboard so that it is spanning the trench and each lead has its own row.

For the two leads on one side of the breadboard, connect one to the power column and the other into digital pin 2 on the Arduino. Also, connect a Brown-Black-Orange resistor between the ground column and the row that has a push-button lead and the wire that was just connected to digital pin 2.

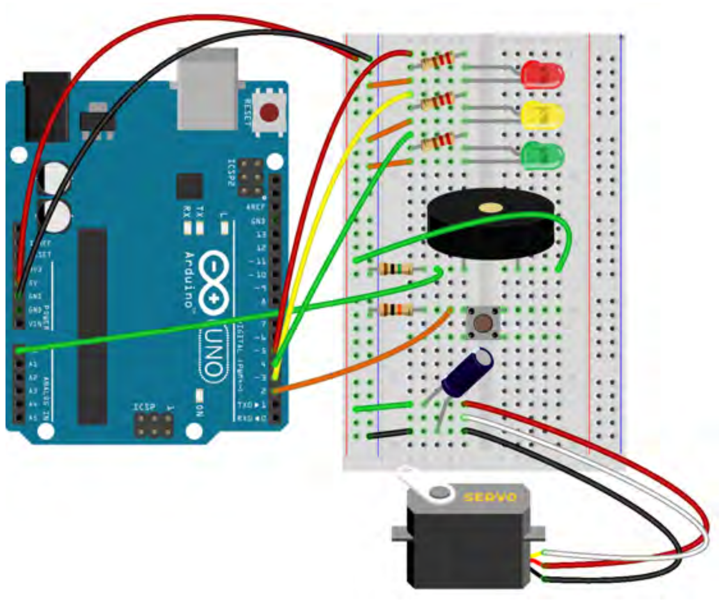


Step 4

Lastly, the servo needs to be wired. There should be three wires coming out of the servo, red, black, and white. Plug the three wires into the breadboard so that the each have their own row.

Connect the row with the red wire to the power column, the row with the black wire to the ground column, and the row with the white wire to digital pin 9 on the Arduino.

The last step is to take a capacitor, one of the blue electrolytic ones, and plug the negative lead, the one close to the black stripe along the side, into the row with the black servo wire. Plug the positive lead, the one not next to the black stripe, into the row with red servo wire and everything is finished.



Step 5

Pin	**Code**	**Input**	**Output**
A0	A0	✓	
D2	2	✓	
D3	3		✓
D4	4		✓
D5	5		✓
D9	9		✓

Table 20: Zoetrope Project Pin Assignment Chart

20.5 ChDuino Basic Test

This project once again involves the piezo, but this time it is used as an analog microphone.

20.6 Writing the Code

The following is the code for this project:

```
// file: knockLock.ch
// use piezo buzzer to detect the vibrations of a series of knocks
```

```
#include <arduino.h>

//Forward declare the function that determine a knock
bool checkForKnock(int value);

//Declare an instance of the Servo class, called myServo
Servo myServo;

//Declare the pin assignment variables and the quiet and loud knock constants
const int piezo = A0,
          switchPin = 2,
          yellowLed = 3,
          greenLed = 4,
          redLed = 5,
          servoPin = 9,
          quietKnock = 50,
          loudKnock = 100;

//Declare variable to remember the number of knocks and hold input from the piezo and
//push-button switch
int knockVal,
    switchVal,
    numberOfKnocks = 0;

//Declare "locked" to be a boolean false, or zero
bool locked = false;

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

    //Tell the program that there is a servo on this pin
    myServo.attach(servoPin);

    //Set the modes of all the i/o pins
    pinMode(greenLed, OUTPUT);
    pinMode(yellowLed, OUTPUT);
    pinMode(redLed, OUTPUT);
    pinMode(switchPin, INPUT);

    //Set the initial conditions so that the servo is in the unlocked position and
    //the green LED is on
    digitalWrite(greenLed, HIGH);
    myServo.write(0);
    Serial.println("The Box is Unlocked!!");
}

void loop(){
    if (locked == false) {

        //Check to see if the button is pressed and if so set the servo to the locked
        //position and turn of the red LED
        switchVal = digitalRead(switchPin);
        if (switchVal == HIGH) {
            locked = true;
            digitalWrite(greenLed, LOW);
            digitalWrite(redLed, HIGH);
            myServo.write(90);
            Serial.println("The Box is Locked!!");
            delay(1000);
        }
    }

    if (locked == true) {

        //Read the input value from the piezo
        knockVal = analogRead(piezo);

        //If the value from the piezo is valid run the check function to determine if
        //the reading is considered a knock if so increase the number of knocks
        if (numberOfKnocks < 3 && knockVal > 0) {
            if (checkForKnock(knockVal) == true) {
                numberOfKnocks++;
            }
            Serial.print(3 - numberOfKnocks);
            Serial.println(" more knocks to go");
        }

        //When the number of knocks reaches three set the servo to the unlocked position,
        //turn on the green LED, and reset variables to initial conditions
        if (numberOfKnocks >= 3) {
            locked = false;
            numberOfKnocks = 0;
            myServo.write(0);
            delay(20);
            digitalWrite(greenLed, HIGH);
            digitalWrite(redLed, LOW);
            Serial.println("The Box is Unlocked!");
        }
    }
}

bool checkForKnock(int value) {

    //If the value read from the piezo is within a certain range it is considered a
    //knock, the yellow LED blinks, and the function returns a boolean true
    if (value > quietKnock && value < loudKnock) {
        digitalWrite(yellowLed, HIGH);
        delay(50);
        digitalWrite(yellowLed, LOW);
        Serial.print("Valid knock of value: ");
        Serial.println(value);
        return true;
    }
    //If the value is not in a certain range then the value is rejected as a knock and
    //the function returns a boolean false
    else {
        Serial.print("Bad knock value: ");
        Serial.println(value);
        return false;
    }
}

int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

Everything is the same in the beginning as previous projects.

```
bool checkForKnock(int value);
```

The line above represents a key aspect of creating a function, called forward declaration. This lets the program know that a function called `checkForKnock()`, that will require an `int` value and return a `bool` variable, actually exists. This is so program does not freak out when it sees a function it does not recognize. The forward declaration is actually not required to create a function. The function definition, in its entirety, can be put in place of the forward declaration, but if the user has multiple functions this can get pretty messy. Thus, it is common programming practice to define the functions after the main part of the code, so it is cleaner to look at. If the functions are used inside of the code, but aren't defined until after the code, the program will have no idea what to do. This is what the forward declaration prevents. While line 13 is the forward declaration, the `checkForKnock()` function is not defined until later, after the code.

```
Servo mySServo;

const int piezo = A0,
        switchPin = 2,
        yellowLed = 3,
        greenLed = 4,
        redLed = 5,
        servoPin = 9,
        quietKnock = 50,
        loudKnock = 100;

int knockVal,
    switchVal,
    numberOfKnocks = 0;
```

Now, it is time to create all of the variables necessary for this project. An instance of the `Servo` class, called 'myServo, is created. `const int` variables are created for all of the Arduino pins that are used in this project as well as two values, 'quietKnock' and 'loudKnock', that define the range inside which the value of a knock detected by the piezo will be considered valid. The user can change these values to make the program more or less sensitive to knocks. Following these declarations, three regular integer variables are created. The first, 'knockVal', is to hold the values read from the piezo. The second, 'switchVal', is a flag to see if the push-button is being pressed. The last, 'numberOfKnocks', records how many valid knocks the function has detected so far.

```
bool locked = false;
```

The last declaration is a variable of the bool type called 'locked'. Boolean variables can only be true or false, or 1 or 0. They are usually flags. In this case, 'locked' will be true when the box is locked and false when it is not. As can be seen from the line above that the initial condition of this variable is assuming that the box is unlocked.

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

    myServo.attach(servoPin);

    pinMode(greenLed, OUTPUT);
    pinMode(yellowLed, OUTPUT);
    pinMode(redLed, OUTPUT);
    pinMode(switchPin, INPUT);

    digitalWrite(greenLed, HIGH);
    myServo.write(0);
    Serial.println("The Box is Unlocked!!");
}
```

The next part of the code defines the **setup()** function. The **attach()** function is used to let the program know there is a servo on the 'servoPin' pin. It sets the mode for all of the pins. The LEDs are outputs and the push-button is an input. The piezo is an analog input, but analog devices are always assumed to be inputs so the mode does not have to be set. Next, to create the initial condition that the box is unlocked, the program turns the pin attached to the green LED on using the digitalWrite() and turns the servo all the way in one direction by writing a 0 value with the **write()** member function of the Servo class. The program then prints out a statement letting the user know that the box is unlocked.

```
void loop(){
    if (locked == false) {

        switchVal = digitalRead(switchPin);
        if (switchVal == 1) {
            locked = true;
            digitalWrite(greenLed, LOW);
            digitalWrite(redLed, HIGH);
            myServo.write(90);
            Serial.println("The Box is Locked!!");
            delay(1000);
        }
    }
```

Now, the actual body of the program can begin by starting the **loop()** function. The first item inside of the **loop()** function is an if-statement with the condition that the bool variable 'locked' be equal to false. This means that the box is unlocked. So, if the box is unlocked then the program will enter the if-statement. Immediately inside of the if-statement the 'switchPin' is read to see if the button is pressed and the return value is assigned to the 'switchVal' variable. Next is another if-statement with the condition that the 'switchVal' variable be equal to 1, meaning that the button has been pressed. If the button is pressed then the program sets the 'locked' variable to `true`, turns the green LED pin off, turns the red LED pin on, and writes a value of 90 degrees to the servo in order to turn it 90 degrees, locking the box. Lastly, the program prints out a statement to tell the user that the box is now locked and pauses for 1000 milliseconds, or one second, to give time for the above actions to happen. The program then exits the if-statement. So, if the box was not locked and the button is pressed then the above segment will lock the box and change which LED is lit.

```
    if (locked == true) {

        knockVal = analogRead(piezo);

        if (numberOfKnocks < 3 && knockVal > 0) {
            if (checkForKnock(knockVal) == true) {
                numberOfKnocks++;
            }
            Serial.print(3 - numberOfKnocks);
            Serial.println(" more knocks to go");
        }
```

Following this is another if-statement, this time the condition is that 'locked' be equal to `true`, meaning that the box is in locked mode. Inside of this if-statement, the program first reads the value from the piezo. If there are any vibrations then the piezo will return a relative variable between 0 and 1023. Next is another if-statement with the condition that the 'numberOfKnocks' variable is less than 3 and that the 'knockVal' variable is greater than 0. So, if the program has not yet recorded three knocks and the value just measured from the piezo is anything but zero then the program enters the if-statement. Inside is yet another if-statement with the condition that the `checkForKnock()` function, given the input of the 'knockVal' variable, is equal to `true`. The `checkForKnock()` function will be defined after the main part of the code, but it will essentially check to see if the value recorded from the piezo is the right value to be a knock. If it is then the function will return `true`. So, the program will enter the if-statement when the value recorded from the piezo is a valid number. Inside, the statement runs the `numberOfKnocks++` line, increasing the value of 'numberOfKnocks' by one. Remember that the '++' operator at the end of a variable will increase it by one. The program then exits the third if-statement, prints out how many more knocks to go until the box will be unlocked and then exits the second if-statement.

```
        if (numberOfKnocks >= 3) {
            locked = false;
            numberOfKnocks = 0;
            analogWrite(servoPin, 0);
            delay(20);
            digitalWrite(greenLed, HIGH);
            digitalWrite(redLed, LOW);
            Serial.println("The Box is Unlocked!");
        }
    }
}
```

The last part of the main body of the code is another if-statement. This one has the condition that the 'numberOfKnocks' variable is greater than or equal to 3. So, if the program has recorded 3 or more knocks it will enter the if-statement. Once inside, the program sets the 'locked' variable to `false`, resets the 'numberOfKnocks' variable, writes a 0 to the servo pin to turn the servo to the unlocked position, and turns the green LED on while turning the red LED off. Lastly, the program prints out a statement, letting the user know the box is now in the unlocked mode, then exits the if-statement in this segment of code and the one in the previous segment that had the condition that the servo be in the locked position. Finally the program exits the `loop()` function. So, going back to the if-statement in the last segment that checked if the box was locked, if the box is locked, the program reads the piezo and checks to see if the program has recorded more or fewer than three knocks. If there has been fewer than three knocks then the program checks if the value it just recorded from the piezo constitutes a knock. If it does, the value for the number of knocks is increased. Going back, if there has been three or more knocks then the program resets the knock counter and locked variables, unlocks the box, and switches which LED is lit. This program will lock the box when the button is pressed and will not unlock it until it detects three valid knocks.

```
bool checkForKnock(int value) {

    if (value > quietKnock && value < loudKnock) {
        digitalWrite(yellowLed, HIGH);
        delay(50);
        digitalWrite(yellowLed, LOW);
        Serial.print("Valid knock of value: ");
        Serial.println(value);
        return true;
    }

    else {
        Serial.print("Bad knock value: ");
        Serial.println(value);
        return false;
    }
}
```

Now that the main body of code has been written, it is the place to define the `checkForKnock()` function that was forward defined earlier. The function has one argument, an integer generically called 'value', and for the purposes of this project this value will be the value recorded from the piezo. The function will return a boolean variable. The variable type positioned before the name of the function, like `bool` in the first line, is the type of variable that the function will return. The code inside of the braces is the code that will execute every time the function is used. In this case, the function is an if-else statement. The if-statement has the condition that 'value' be greater than the 'quietKnock' and less than the 'loudKnock' variables. This means that the function will enter the if-statement if the value that it is given, 'knockVal' in this project, is inside of the range defined by 'quietKnock' and 'loudKnock', and is thus a valid knock. In that case the function will blink the yellow LED for 50 milliseconds, prints a message to the user that the knock value was valid, and then returns a boolean `true`. If 'value' is not within the range defined by the 'quietKnock' and 'loudKnock' constants, then the function enters the else-statement. Inside of the else-statement, the function prints a message to the user that the value indicates it is not a valid knock then returns a boolean `false`. That is the end of the function.

```
int main(){
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

The program ends by calling the **setup()** function once and repeatedly calling the **loop()** function an infinite number of times using an infinite while-loop.

If it is working correctly then servo should move to the locked position and the red LED should be on after the push-button has been pressed. Once locked, the program should move the servo to the unlocked position and turn the green LED on only after it has detected three valid knocks from the user. Each time a valid knock is confirmed the yellow LED should blink.

20.7 Exercises

1. Change the 'quietKnock' and 'loudKnock' variables so that the piezo will detect someone blowing on it, or when the table is shaken hard (earthquake monitor)
2. Create functions for the unlocking and locking process
3. Create a function for calibration in the light theremin project code.

21 Appendix

21.1 Differences Between ”Learning Arduino in Ch for the Absolute Beginner” and ”Learning Arduino in C”

There are two versions of this textbook, one that teaches how to program an Arduino in the C programming language and another that teaches how to program the Arduino in the C/C++ interpretor Ch. The idea behind creating these two versions was so that the material would be engaging and understandable to student of different programming backgrounds and skill-sets. The ”Learning Arduino in Ch for the Absolute Beginner” book was designed for those students with the smallest amount of programming experiences, such as for the absolute beginner students in middle school. The ”Learning Arduino in C” book was designed for student with slightly more experience, though still beginners, and has a target age range of students in high school or college. The bulleted list below details the exact syntactical differences between the two books.

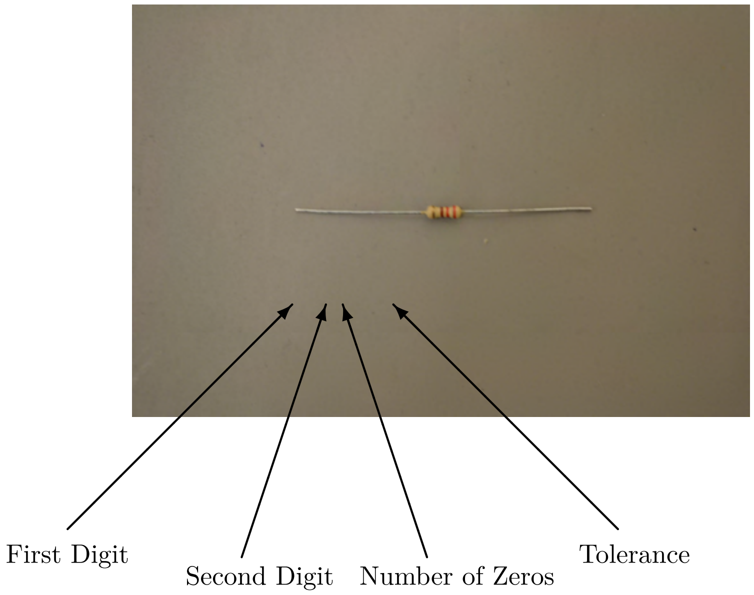
- The Ch version does not use the `const` modifier when declaring variables
- The Ch version uses the classic `printf()` function to print text and variables instead of the Arduino's Serial class functions
- The Ch version does not use the `setup()` and `loop()` functions defined by Arduino, where the `loop()` function is replaced by an infinite `while`-loop
- The Ch version does not use the `main()` function
- The Ch version uses `if-else` statements instead of the `switch`-statement
- The Ch version uses `while`-loops instead of `for`-loops
- The Ch version does not use the `bool` variable type
- The Ch version does not go over creating user defined functions
- The Ch version uses the `delaySeconds()` function instead of the `delay()` function

21.2 Resistor Color Codes

Resistors are small elements and it is impractical to write their resistance values out in text because nobody would be able to read it. That is why the color coding system for resistors was developed. The idea is simple, on one side of the resistor are three bands that can be any color in the spectrum and on the other is a single band that is usually gold or silver (note that this single band can be brown or red but this is much less common and should not appear in this kit). The colors represent different numbers according to Table 21 below. An easy way to remember the order is with the mnemonic Better Be Right Or Your Great Big Plan Goes Wrong. If the resistor is oriented so the three colored bands are on the left, then going from left to right the first band is the first digit, the second band is the second digit, and the third band is the number of zeros behind these two digits. So for the example of a Brown-Black-Orange resistor, the first band, being brown, means that the first digit is one, the second band, being black, means that the second digit is also zero, the third band, being orange, means that there are three zeros behind those two digits. Thus a Brown-Black-Orange resistor has a resistance value, measured in units of Ohms, of 1-0-000, or 10000. The fourth band, the one that is usually gold or silver, is the tolerance of the value above. Tolerance means that the actual, real life, resistance of the resistor is guaranteed to be within a certain percent above or below the value represented by the first three colored bands. Gold represents a 5% tolerance, and silver represents a 10% tolerance.

	Color	**Number**
⬛	Black	0
🟫	Brown	1
🟥	Red	2
🟧	Orange	3
🟨	Yellow	4
🟩	Green	5
🟦	Blue	6
🟪	Purple/Violet	7
▨	Gray	8
⬜	White	9

Table 21: Resistor Color-Number Conversion Chart



21.3 Using the Multi-Meter

A multi-meter is a very useful tool in electric circuitry. It can measure voltages and currents for both AC and DC as well a the resistance of electronic components. Typical multi-meter systems will consist of the main multi-meter unit and two long probing wires, one red and one black. As can be seen in Figure 44 below, near the bottom of the front of the multi-meter are three plugs: 'COM', 'VΩmA', and '10ADC'. Most typical multi-meters will have the same three plugs, though they may have some small differences in the labels. The black probing wire should always be connected to the 'COM' plug and the red wire should almost always be connected to the 'VΩmA' plug. Notice that the points around the central dial are divided into section. These sections determine what the multi-meter is going to measure and the points within each section change the range of values that can be measured. How to measure current, voltage, and resistance will be explained in the following sections. For all measurements, the red probe should be connected to the positive, or higher, voltage and the black probe should be connected to the ground, or lower, voltage

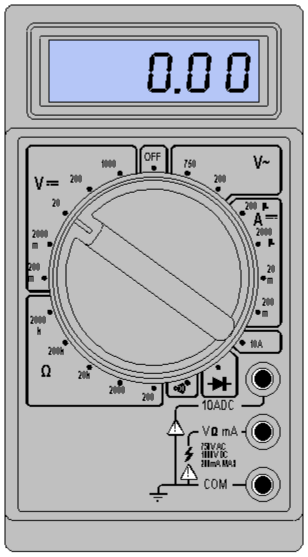


Figure 44: Typical Multi-meter Diagram

21.3.1 Current

For any current measuring application in this book, the current will be DC, or Direct Current. Thus the section around the dial required to measure this current is the one labeled 'DCA', meaning Direct Current Amperes, Amperes being a unit of current. The scope of this book will deal mostly with currents around the one thousandth of an Ampere range so the '20m' point inside of the 'DCA' section would be the best to use and will display units of milliamps. When measuring current, **it is very important that the probes should ALWAYS be spanning two, unconnected points in a circuit**, in other words, the probes should be completing the circuit. This is opposite to the method to measure voltage. If the probes are placed across a component while measuring current, like on either sides of a resistor, it could blow the fuse inside of the multi-meter which is not good.

21.3.2 Voltage

For any voltage measuring application in this book, the voltage will be DC and will be on the order of regular volts. Thus, the point on the dial which would be most appropriate to measure the voltages in this book would be the '20' point inside of the 'DCV' section, standing for Direct Current Volts. This measuring point will display units of volts. When measuring voltage, the probes should be connected across a component.

For example, one probe would touch one side of a resistor and the other probe would touch the other side. This is opposite to the method for measuring current. **It is very important that the probes should NEVER be spanning two, unconnected points in a circuit** as this can blow a fuse which is not good.

21.3.3 Resistance

The section around the dial to measure resistance is labeled 'Ω'. The specific point to use within this section will depend on how large the resistance of a resistor is supposed to be. The '200' point would be appropriate for a 220Ω resistor and the '2000k' would be appropriate for a 1MΩ resistor. To measure the resistance of a resistor, separate the resistor from the circuit and touch the probes on either side of the resistor.

21.4 Macros

All macros are the same for both the Arduino and Ch languages

Macro	Value	Meaning
`INPUT`	0	Indicates pin should be set to input mode
`OUTPUT`	1	Indicates pin should be set to output mode
`LOW`	0	Indicates a digital pin should be set to 0 volts
`HIGH`	1	Indicates a digital pin should be set to 5 volts
`A0`	14	Indicates a function should apply to analog pin 0
`A1`	15	Indicates a function should apply to analog pin 1
`A2`	16	Indicates a function should apply to analog pin 2
`A3`	17	Indicates a function should apply to analog pin 3
`A4`	18	Indicates a function should apply to analog pin 4
`A5`	19	Indicates a function should apply to analog pin 5
`A6`	20	Indicates a function should apply to analog pin 6
`A7`	21	Indicates a function should apply to analog pin 7

Table 22: Macros and their values

21.5 Functions in Ch

`void setTrigPin(int pin)`
- Sets the pin used as the trigger pin when using ultrasound sensor. The default is pin 6.

`double readUltrasoundSensor(int trigPin, int echoPin)`
- Returns the distance, in inches, of an object as detected by an ultrasound sensor. The function is implemented as following:

```
          digitalWrite(trigPin,LOW);
          delayMicroseconds(2);
          digitalWrite(trigPin,HIGH);
          delayMicroseconds(10);
          digitalWrite(trigPin,LOW);
          pulseDuration = pulseIn(echoPin, HIGH);
          distance = dist = duration *0.013397*0.5;
```

21.6 Functions in Ch and Arduino IDE

`void pinMode(int pin, int mode)`
- Sets the mode of a pin on the Arduino to either INPUT or OUTPUT mode.

`int digitalRead(int pin)`
- Reads the voltage of one of the Arduino's digital pins. Will return 0 if the voltage is less than 3 volts and return a 1 if its greater than 3 volts.

`void digitalWrite(int pin, int value)`
- Sets one of the Arduino's digital pins to either HIGH (5 volts), or LOW (0 volts).

`int analogRead(int pin)`
- Reads the voltage of one of the Arduino's analog pins. Will return a value between 0 and 1023 based on the voltage level of the pin.

`void analogWrite(int pin, int value)`
- Writes a value between 0 and 255 to one of the Arduino's digital pins that is capable of PWM. The effective voltage will be between 0 and 5 volts, the exact voltage depending upon the value written.

`unsigned long pulseIn(int pin, int mode, unsigned long timeout)`
- Measures the time, in microseconds in between pulses on a pin.

`void delay(int milliseconds)`

- Pauses the program for a certain number of milliseconds.

`int map(int inputVal, int inputMin, int inputMax, int outputMin, int outputMax)`

- Transforms a value in a certain input number range into the equivalent value in a certain output number range, according to its ratio to the size of the scale.

`int millis(void)`

- Returns the time, in milliseconds, since the beginning of the program.

`int micros(void)`

- Returns the time, in microseconds, since the beginning of the program.

`void randomSeed(unsigned long seed)`

- Initializes the pseudo-random number generator, causing it to start at an arbitrary point in its random sequence.

`int random(int max)` or
`int random(int min, int max)`

- Returns a random number chosen from a range of numbers defined by a minimum and maximum. If no minimum is specified the default is 0.

`void tone(int pin, int frequency)` or
`void tone(int pin, int frequency, unsigned long time)`

- plays a note on a piezo with a certain frequency for a certain amount of time in milliseconds. If no time is specified, the default value is 4294967.295 seconds, the largest value an `unsigned long` can be.

`void noTone(int pin)`

- Turns off a piezo.

`void isAlphaNumeric(int char)`

- Returns true if the character is alphanumeric.

`void isAlpha(int char)`

- Returns true if the character is alpha.

`void isAscii(int char)`

- Returns true if the character is ASCII.

`void isWhitespace(int char)`

- Returns true if the character is a white space.

`void isControl(int char)`
- Returns true if the character is a control character.

`void isDigit(int char)`
- Returns true if the character is a digit.

`void isGraph(int char)`
- Returns true if the character is a printable character.

`void isLowerCase(int char)`
- Returns true if the character is a lowercase character.

`void isPrintable(int char)`
- Returns true if the character is a printable character.

`void isPunct(int char)`
- Returns true if the character is a punctuation character.

`void isSpace(int char)`
- Returns true if the character is a space character.

`void isUpperCase(int char)`
- Returns true if the character is an uppercase character.

`void isHexadecimalDigit(int char)`
- Returns true if the character is a valid hexadecimal digit.

`void shiftOut(int dataPin, int clockPin, int bitOrder, byte value)`
- Shifts out a byte of data one bit at a time. Starts from either the most (leftmost) or least significant(right) bit Each bit is written in turn to a data pin.

`byte shiftIn(int dataPin, int clockPin, int bitOrder)`
- Shifts in a byte of data one bit at a time. Starts from either the most (leftmost) or least significant(right) bit.

`byte lowByte(x)`
- Extracts the low-order (rightmost) byte of a variable x. The function works with arguments of any data type.

`byte highByte(x)`

- Extracts the high-order (leftmost) byte of a variable x. The function works with arguments of any data type.

`int bitRead(x, n)`

- Reads the bit number n from variable x. The value of n starts from 0 for the least significant bit. The function works with arguments of any data type.

`int bitSet(x, n)`

- Sets (writes a 1) bit number n of a numerical variable x. The value of n starts from 0 for the least significant bit. The function works with arguments of any data type.

`int bitClear(x, n)`

- Clears (writes a 0) bit number n of a numerical variable x. The value of n starts from 0 for the least significant bit. The function works with arguments of any data type.

`int bitWrite(x, n, b)`

- Writes value b (0 or 1) to bit n of a numerical variable x. The function works with arguments of any data type.

`int constrain(x, a, b)` or
`double constrain(x, a, b)`

- Constrains the value of numerical variable x to be within the lower bound *a* and the upper bound *b*. The function returns x if it is within *a* and *b*, *a* if x is less than *a* and *b* if x is greater than *b*. The function works with arguments of any data type.

21.7 Classes

21.7.1 The Serial Class in Ch and Arduino IDE

`int Serial.available(void)`
- Returns the number of bytes available to read on the serial port.

`void Serial.begin(int baud)`
- In Ch, this function checks to see if serial communication is functioning and at the right Baud Rate. In the Arduino IDE, this function tells the Arduino to initialize serial communication and enables the use of the `print()` and `println()` function.

`bool Serial.connected(void)`
- In Ch, this function returns always true. It is the Arduino IDE equivalent of if(Serial) that returns true when the serial communication has been properly established and the serial port is ready.

`void Serial.end(void)`
- In Arduino IDE this function disables the serial communication. In ChIDE it has no effect.

`void Serial.flush(void)`
- In ChIDE this function flushes the Input/Output buffer. In Arduino IDE this function flushes the serial buffer.

`double Serial.parseFloat(void)`
- In ChIDE this function returns the first valid floating point number from the Input/Output buffer. In Arduino IDE this function returns the first valid floating point number from the serial buffer.

`double Serial.parseInt(void)`
- In ChIDE this function returns the first valid integer number from the Input/Output buffer. In Arduino IDE this function returns the first valid integer number from the serial buffer.

`int Serial.peek(void)`
- In ChIDE it returns the next byte read from the Input/Output pane, without removing it from the internal buffer. In Arduino IDE it returns the next byte of incoming serial data without removing it from the internal serial buffer.

`int Serial.print(...)`
- Displays a string or variable, of any type, to the Input/Output Pane in ChIDE. It returns the number of bytes written.

`int Serial.println(...)`
- Displays a string or variable, of any type, to the Input/Output Pane in ChIDE and will start a new line after printing is complete. It returns the number of bytes written.

`int Serial.readB()`
- Reads incoming serial data, one byte at a time. It is the equivalent of Serial.read() in Arduino IDE.

`int Serial.writeB(...)`
- Displays a string or variable, of any type, to the Input/Output Pane in ChIDE. It returns the number of bytes written. This function is the equivalent of Serial.write() in Arduino IDE.

21.7.2 The Servo Class in Ch and Arduino IDE

`void Servo.attach(int pin)`
- Tells the Arduino that a servo is connected to this pin so that the servo can be indexed and PWM can be set up.

`void Servo.write(int angle)`
- Turns the servo to the specified angle, in degrees.

21.7.3 The LiquidCrystal Class in Ch and Arduino IDE

```
LiquidCrystal(int rsPin, int enablePin, int dataPin1, int dataPin2, int dataPin3,
   int dataPin4)
```
- This is a constructor that initializes all of the pins required to control an LCD screen in 4 bit mode

`void LiquidCrystal.begin(int column, int row, int dotSizeMode)`
- Turns an LCD screen on, indicating the number of columns and rows the screen has and also setting the size of the dots that make up the characters

`void LiquidCrystal.print("String")`
- Prints out a string of characters on the LCD screen

`void LiquidCrystal.setCursor(int column, int row)`
- Sets the position on the LCD screen where printing will start

`void LiquidCrystal.clear(void)`
- Clears anything currently displayed on the LCD

`void LiquidCrystal.home(void)`
- Positions the cursor on the upper left corner of the LCD.

`void LiquidCrystal.noDisplay(void)`
- Turns off the LCD display. without losing the text currently shown on it.

`void LiquidCrystal.display(void)`
- Turns on the LCD display.

`void LiquidCrystal.noBlink(void)`
- Turns off the blinking LCD cursor.

`void LiquidCrystal.blink(void)`
- Displays the blinking LCD cursor.

`void LiquidCrystal.noCursor(void)`
- Hides the LCD cursor.

`void LiquidCrystal.cursor(void)`
- Displays the LCD cursor.

`void LiquidCrystal.scrollDisplayLeft(void)`
- Scrolls the contents of the display (text and cursor) one space to the left.

`void LiquidCrystal.scrollDisplayRight(void)`
- Scrolls the contents of the display (text and cursor) one space to the right.

`void LiquidCrystal.leftToRight(void)`
- Sets the direction for the text written to the LCD to left-to-right, the default.

`void LiquidCrystal.rightToLeft(void)`
- Sets the direction for the text written to the LCD to right-to-left.

`void LiquidCrystal.autoscroll(void)`
- Turns on the automatic scrolling of the LCD.

`void LiquidCrystal.noAutoscroll(void)`
- Turns off the automatic scrolling of the LCD.

`void LiquidCrystal.createChar(byte num, byte data[])`
- Creates a custom character (gliph) for use on the LCD. Up to eight characters, numbered from 0 to 7 are supported. The appearance of each character is specified by an array of eight bytes, one for each row. The num argument specifies the character to create and the data array specifies the appearance of the character.

21.7.4 The CPlot Class in Ch

`void CPlot.label(int axis, string_t label)`

- Sets the label of an axis. The specific axis depends on the first argument which is usually a macro of the form `PLOT_AXIS_X` or `PLOT_AXIS_Y`.

`void CPlot.title(string_t title)`

- Sets the title of a plot

`void CPlot.data2DCurve(array double x[], array double y[], int n)`

- Plots two arrays of data against each other. The arrays must have 'n' number of elements each.

`void CPlot.data3DCurve(array double x[], array double y[], array double z[], int n)`

- Plots three arrays of data against each other. The arrays must have 'n' number of elements each.

`void CPlot.plotting(void)`

- Creates a plot based on parameters set by previous functions

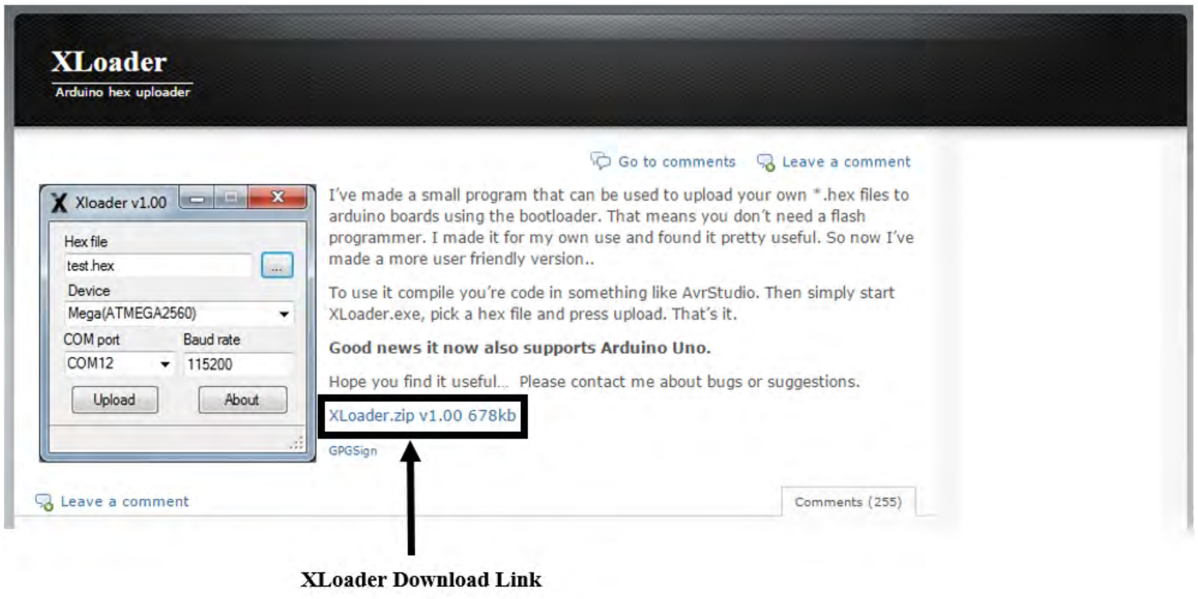
21.8 Input/Output Alternatives

Ch and Arduino IDE	Ch
`int i = 1;` `Serial.print("i = ");` `Serial.print(i);`	`int i = 1;` `printf("i = %d", i);`
`int i = 1;` `Serial.print("i = ");` `Serial.println(i);`	`int i = 1;` `printf("i = %d\n", i);`
`unsigned int i = 1;` `Serial.print("i = ");` `Serial.println(i);`	`unsigned int i = 1;` `printf("i = %u\n", i);`
`double d = 1.0;` `Serial.print("d = ");` `Serial.println(d);`	`double d = 1.0;` `printf("d = %lf\n", d);`
`float f = 1.0;` `Serial.print("f = ");` `Serial.println(f);`	`float f = 1.0;` `printf("f = %f\n", f);`
`long l = 10000;` `Serial.print("l = ");` `Serial.println(l);`	`long l = 10000;` `printf("l = %l\n", l);`
`unsigned long l = 10000;` `Serial.print("l = ");` `Serial.println(l);`	`unsigned long l = 10000;` `printf("l = %lu\n", l);`
`char c = 'c';` `Serial.print("c = ");` `Serial.println(c);`	`char c = 'c';` `printf("c = %c\n", c);`
`Serial.println("String");`	`printf("String\n");`

21.9 Installing Arduino Firmware For Ch

If the Arduino board that you are using does not already have the firmware installed in order for it to work with Ch, for example if it is fresh out of the box, then please follow these instructions. First, the firmware file, called ”ChArduino.hex”, should be located in one of the folders in C:/Ch/package/charduino/Firmware depending on which board is being used. If the firmware files are not there, go to http://c-stem.ucdavis.edu/ and, in the top right corner, press either the ”Register” or ”Log In” button, depending if you already have an account. Click on the ”Downloads” tab on the left side bar. From here, scroll down the page until you see the ”Click here to Download the latest Arduino Driver” link. Click the link and you should be prompted to download a file called ”arduino.hex”.

The next step is to install a program called XLoader that will install the arduino.hex file onto the Arduino. Navigate to http://russemotto.com/xloader/. From here, click on the link near the center called ”XLoader.zip” which should start a download. The window should look like Figure 45.


XLoader
Arduino hex uploader

Go to comments Leave a comment

Xloader v1.00
Hex file: test.hex
Device: Mega(ATMEGA2560)
COM port: COM12
Baud rate: 115200
Upload About

I've made a small program that can be used to upload your own *.hex files to arduino boards using the bootloader. That means you don't need a flash programmer. I made it for my own use and found it pretty useful. So now I've made a more user friendly version..

To use it compile you're code in something like AvrStudio. Then simply start XLoader.exe, pick a hex file and press upload. That's it.

Good news it now also supports Arduino Uno.

Hope you find it useful... Please contact me about bugs or suggestions.

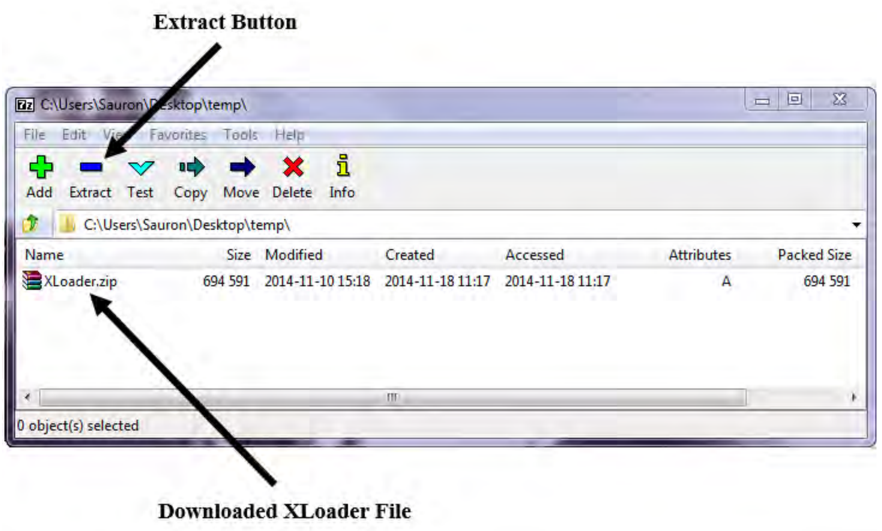
XLoader.zip v1.00 678kb

GPGSign

Leave a comment Comments (255)

Figure 45: XLoader Website Window

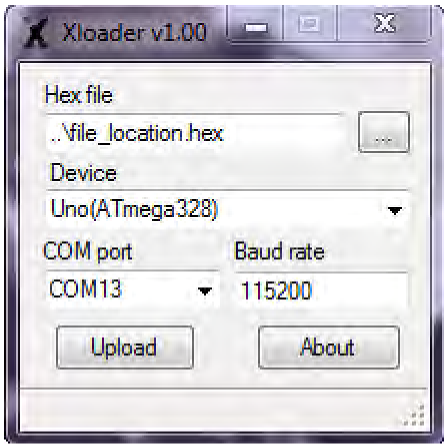
Go to the All Programs section of your start menu and open the program called ”7-zip File Manager” located inside of the Utilities folder in the same location as ChIDE. From inside the 7-zip window, navigate to the location of the XLoader.zip file you just downloaded. The window should look like Figure 46.



Name	Size	Modified	Created	Accessed	Attributes	Packed Size
XLoader.zip	694 591	2014-11-10 15:18	2014-11-18 11:17	2014-11-18 11:17	A	694 591

Figure 46: 7-zip Window

Select the file and click the ”Extract button” on the 7-zip tool bar and hit the Okay button, this should create folder called XLoader. Within that newly created folder should be an item called XLoader.exe. Double click on this program and follow all the prompts, this should install XLoader. Once XLoader is installed, open it up. It should look like Figure 47.



Hex file: ..\file_location.hex
Device: Uno(ATmega328)
COM port: COM13
Baud rate: 115200

Figure 47: XLoader Program Window

There should be a ”Hex file” window, there enter the location of the arduino.hex file downloaded earlier or use the browse button to the left of the window. Next, expand the drop down menu under ”Device” and select the Arduino board you are using. This will typically be the Uno. Next, connect the Arduino to the computer. You will need to find out which COM port it is connected to. To do this, use the search bar on the start menu to search for ”Device Manager” and open it up. Expand the section that called Ports and the Arduino should be listed next to a COM port, called ”COM#”, where ’#’ is a number, it will look like Figure 48.

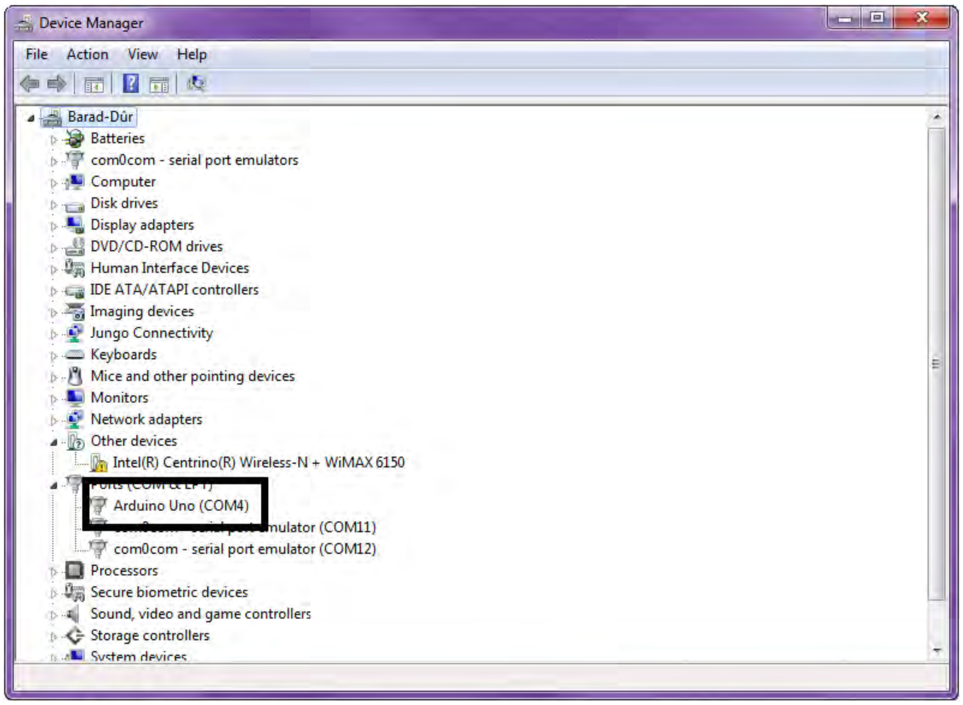


Figure 48: Device Manager Window

Remember the number and go back to XLoader. Expand the drop down menu under COM Port and select the port that the Arduino is connected to. The Baud rate setting does not have to be worried about. Finally, simply press upload and wait for completion. The Arduino should be ready to interact with Ch and ChIDE.