The open access version with the first 7 sections

Learning Physical Computing with Arduino for the Absolute Beginner

Version 2.0

Harry H. Cheng

UC Davis Center for Integrated Computing and STEM Education (C-STEM)
University of California-Davis
http://c-stem.ucdavis.edu

September 5, 2018

Copyright © 2018 by Harry H. Cheng, All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of the author, including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.
## Contents

### Part I: Introduction to Physical Computing

1 Introduction

1.1 The Board ........................................ 9
1.2 The External Circuitry .......................... 10
1.3 The Breadboard ................................. 13
1.4 The Pins ........................................ 14
1.5 Circuit Schematic Diagrams .................. 15
1.6 Exercises ....................................... 16

2 Setting Up and Testing Circuitry with Arduino

2.1 Project: Blink through ChDuino ............... 17
2.2 New Concepts .................................... 17
2.3 Required Components and Materials .......... 17
2.4 Getting Started ................................. 17
2.5 Building the Circuit ............................ 18
2.6 Opening CSTEM Studio on the Arduino ....... 19
2.7 Using the ChDuino GUI .......................... 20
2.7.1 Find and Manage Arduino Boards .......... 20
2.8 Scan Arduino Board via USB .................... 21
2.9 Scan Arduino Board via Bluetooth ............. 22
2.10 Connect and Control an Arduino Board ..... 26
2.11 Updating Firmware for the Arduino ........ 26
2.12 Using ChDuino to Test Hardware Setup .... 28
2.13 Exercises ....................................... 28

3 Programming Arduino in RoboBlockly and Ch

3.1 Project: Blink through RoboBlockly and Ch Programs ......................... 29
3.2 New Concepts .................................... 29
3.3 Required Components and Materials .......... 30
3.4 Getting Started ................................. 30
3.5 Exercises ....................................... 34
3.6 Accessing the Code in C-STEM Studio ........ 34
3.7 Using ChIDE to Run Programs ................. 37
3.8 Writing the Code ................................ 37
3.9 Using ChIDE to Understand and Troubleshoot Programs ..................... 39
3.10 Exercises ....................................... 43

4 More Blinking

4.1 The Direct Method of Repetition ............... 44
4.2 SOS .............................................. 44
4.3 Alternate Repetition Method Using the While-Loop ............ 47
4.4 Using an Infinite While-Loop .................. 48
4.5 Debugging Revisited ............................ 49
4.6 Multiple LEDs .................................. 55
4.7 Exercises ....................................... 57

5 Controlling a Traffic Light with a Push-Button

5.1 Project: Traffic Light ........................... 58
5.2 New Concepts .................................... 58
5.3 Required Components and Materials .......... 58
5.4 Building the Circuit ............................ 58
5.5 Writing the Code Without Input .............. 60
5.6 Building the Circuit (Part 2) ................. 61
5.7 ChDuino Basic Test ............................. 62
5.8 Writing the Code for Input ........................................................................ 63
5.9 Exercises ..................................................................................................... 65

6 Using a Potentiometer to Dim an LED .......................................................... 66
  6.1 Project: Dimmer ......................................................................................... 66
  6.2 New Concepts ........................................................................................... 66
  6.3 Required Components and Materials ......................................................... 66
  6.4 Building the Circuit ................................................................................. 66
  6.5 ChDuino Basic Test ................................................................................ 68
  6.6 Writing the Code ...................................................................................... 69
  6.7 Exercises .................................................................................................. 70

7 Data Acquisition and Plotting Using a Photo-resistor .................................... 71
  7.1 Project: Graphing Light ........................................................................... 71
  7.2 New Concepts .......................................................................................... 71
  7.3 Required Components and Materials ....................................................... 71
  7.4 Learning How to Plot Using plotArray.ch .............................................. 71
  7.5 Building the Circuit ............................................................................... 74
  7.6 Writing the Code ..................................................................................... 75
  7.7 Exercises ................................................................................................ 78

8 Using Photo-Resistors and Potentiometers to Change the Brightness and Color of an RGB LED ................................................................. 80
  8.1 Project: Color Mixing Lamp .................................................................... 80
  8.2 New Concepts .......................................................................................... 80
  8.3 Required Components and Materials ....................................................... 80
  8.4 Building the Circuit ............................................................................... 80
  8.5 Writing the Code ..................................................................................... 84
  8.6 Exercises ................................................................................................ 86

Part II: Physical Computing with Robots ........................................................... 87

9 Using the Arduino to Control Linkbot or Mindstorms .................................. 88
  9.1 Project: Direction Bot ............................................................................. 88
  9.2 New Concepts .......................................................................................... 88
  9.3 Required Components and Materials ....................................................... 88
  9.4 Building the Circuit ............................................................................... 89
  9.5 Writing the Code to Control the Linkbot ............................................... 90
  9.6 Stopping a Linkbot or Mindstorms Robot .............................................. 93
  9.7 Writing the Code to Control the Mindstorms ......................................... 94
  9.8 Writing the Code to Control Both Robots ............................................. 95
  9.9 Exercises ................................................................................................ 98

10 Control the Speed and Spin of a Mindstorms or Linkbot with a Potentiometer ................................................................. 99
  10.1 Project: Potentiometer Bot .................................................................... 99
  10.2 New Concepts ........................................................................................ 99
  10.3 Required Components and Materials ....................................................... 99
  10.4 Building the Circuit ............................................................................... 100
  10.5 Writing the Code for Linkbot Speed ...................................................... 101
  10.6 Writing the Code for Linkbot Spin .......................................................... 103
  10.7 Exercises ................................................................................................ 105

11 Adding Sensors to Linkbot Using Linkbot Arduino Pack ............................. 106
  11.1 Project: Linkbot Siren .......................................................................... 106
  11.2 New Concepts ........................................................................................ 106
  11.3 Connecting to Power and Ground on the Breadboard ......................... 106
  11.4 Connecting to Bluetooth ...................................................................... 106
  11.5 Required Components and Materials ................................................... 107
  11.6 Building the Circuit .............................................................................. 108
Part III: Advanced Physical Computing
Part I:

Introduction to Physical Computing
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.

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 provide the user with enough knowledge to express their creativity through their own personal projects. This book assumes that the user has the Arduino Uno Starter Kit and Arduino Uno and Pi Sensor Kit, which are available for purchase from Barobo, Inc. (http://www.barobo.com).
1.1 The 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 are both digital and analog pins, and a number of the digital pins are PWM capable, which will be discussed later in Section 6. The controllers also need to communicate with other computers so that user instructions can be received or programs uploaded.

Ch code currently supports a variety of Arduino boards, including the Arduino Uno board. 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 could be wired directly to the Arduino but that would be very messy. That is why a breadboard is typically used for building temporary circuits. A breadboard contains 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 pins into certain rows will connect those wires. Breadboards allow for relatively complex circuits to be built and modified easily for testing and troubleshooting. A more complex description will be given in Section 1.4: The Breadboard.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Hardware</th>
</tr>
</thead>
</table>

Resistors

As the simplest electrical component, resistors impede 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 25.1 in the Appendix for the resistor color-coding system. They have two leads, or little wires sticking out, that connect to the circuit. Resistors are bidirectional, meaning the leads do not have a positive or negative.

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 13 and 22 to smooth out the voltage changes across a servo. Many capacitors have a positive and negative side that must be plugged in correctly. *The side with the stripe on your capacitor is always the negative or ground!*

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.
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 allows 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 produce a wide range of resistances which in turn creates various of voltages that can be detected by the computer. While switches can only be on or off, a potentiometer gives the computer a continuous range of feedback values. A classic implementation would be a volume knob on a speaker or stereo. Because their basic component is a resistor, potentiometers are bidirectional.

Diodes

A diode simply allows electrical current to flow in one direction. They must be set up in the correct orientation to work properly. A diode will be used later in Section 19 to prevent the current generated from a DC motor from damaging the circuit. Think of the symbol like an arrow that shows the direction of current toward the end with a marking.

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 that create light when current is passed through. They are used widely as indicator lights on electrical devices, and share the property of controlling current direction like a standard diode. They have a longer positive and shorter negative lead.

RGB LEDs

RGB (Red Green Blue) LEDs are a type of light emitting diode that can create any combination of colors by combining 3 LEDs into 1. They have 4 pins: red, green, blue, and ground. Applying more voltage to the red pin gives a more red hue, and the same goes for green and blue. Like normal LEDs and diodes, current only flows in one direction from the color pins to ground.

Photo-resistors

A photo-resistor is a special type of resistor that changes its resistance when it is exposed to light. Photo-resistors act as variable resistors, 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.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Symbol" /></td>
<td><img src="image2.png" alt="Hardware" /></td>
</tr>
<tr>
<td><img src="image3.png" alt="Symbol" /></td>
<td><img src="image4.png" alt="Hardware" /></td>
</tr>
<tr>
<td><img src="image5.png" alt="Symbol" /></td>
<td><img src="image6.png" alt="Hardware" /></td>
</tr>
</tbody>
</table>

**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 single board computer 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 22 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 single board computer. 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 single board computer sends to the servo.

**BJTs (Bipolar Junction Transistor)**
While a BJT may resemble a temperature sensor, they behave quite differently. BJTs are a type of transistor that have a base, collector, and emitter pin. 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 applications and are the basis of modern computing.

**MOSFETs (Metal Oxide Semiconductor Field Effect Transistor)**
MOSFETs are a second type of transistor that work in essentially the same way as a BJT. Even though they are slower at switching than BJTs due to internal parasitic capacitances, they are much more tolerant to heat and thermal noises, and thus are better suited for high-powered applications. Instead of a base, collector, and emitter, MOSFETs have a gate, drain, and source.
H-Bridges
An H-bridge is an integrated chip, or IC, that controls the polarity, positive-to-negative or the negative-to-positive, of the voltage across a component. This is particularly useful in Section 20 to control the direction a DC motor will spin.

DC Motor
A DC motor rotates a shaft when a voltage is applied to it. The direction of the shaft's 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. Thus, 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. The breakout board plugs 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 it does the same for a '-' column and the pin marked 'GND'. Other common names for these columns are rails or buses, so do not be confused when these words are used interchangeably.

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.
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 in Figure 1.4 explains the purpose of each of the pins on the Arduino.
### 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, continuous 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 Schematic 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 schematic diagram. The standard circuit schematic 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 1.5 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 1.5 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 schematic 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 1.5: Example Circuit Schematic Diagram

1.6 Exercises

1. Which component(s) have a positive and negative lead?

2. Which sensor would be best to make a circuit that detects whether it fell over? Why?

3. Which sensor would be best to determine what time of day it is? Why?

4. What is another name for the (+) and (-) columns of the breadboard?
2 Setting Up and Testing Circuitry with Arduino

2.1 Project: Blink through ChDuino

**Project Description:** This first project will have an LED turn on, blink a couple of times, then 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 or ground 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. For this project you will learn how to set up and test circuitry with Arduino.

2.3 Required Components and Materials

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

2.4 Getting Started

At the most fundamental level, a digital output is like a light switch, as shown in Figure 2.1. If the switch is open, the light is in the off state. If the switch is closed, the light is in the on state.

![Figure 2.1: Simple Light Switch Circuit](image)

However, the hardware of a simple switch is inflexible and cannot be changed without altering the physical circuit. That is why microcontrollers are used for many devices, as they can perform a wide range of tasks through software changes, without changing the circuit itself.
2.5 Building the Circuit

First, plug one end of a wire into one of the breadboard columns marked ‘-', and the other end of the wire to the port marked ‘GND’ in the Arduino as shown in Figure 2.4.

Plug one of the wires coming out of the LED, called leads, into one row of the breadboard, and plug the other lead into a different row to obtain the setup in Figure 2.4. 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.
Plug one end of one Red-Red-Brown, or 220 Ohm, resistor (see Appendix Section 25.1) 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, place a wire into the row with the longer end of the LED, the one that should be bent. Plug the other end of the wire into digital pin ‘3’ on the Arduino.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Code</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3</td>
<td>3</td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>

Table 2: Blink Project Pin Assignment Chart

2.6 Opening CSTEM Studio on the Arduino

Downloading and installing C-STEM Studio is required before continuing. Open C-STEM Studio on your computer and find “ChDuino” on the left menu, as shown in Figure 2.5. Double-click on it to launch the ChDuino program.
2.7 Using the ChDuino GUI

The ChDuino GUI (Graphical User Interface) 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.

2.7.1 Find and Manage Arduino Boards

First, plug the Arduino into your computer and open the ChDuino program. Once it opens, the window in Figure 2.6 should pop up.
On the left of Figure 2.6 is a section containing a list of Arduino boards that are currently plugged in to your 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 of each board is a dot indicating its connection status. If the dot is green, then the board is currently connected. If the dot is red, then the board is not connected. To scan new devices, select the device type you want to scan in the “Device type” box as shown in Figure 2.7.

2.8 Scan Arduino Board via USB

First, plug the Arduino into your computer by a USB cable and open the ChDuino program. Check the “USB” option in the Device Type Box. Click the “Scan” button to search for available boards. The devices added to the list contains the type of the board and the port number associated with the computer as shown in Figure 2.8.
2.9 Scan Arduino Board via Bluetooth

Note: Be sure to enable Bluetooth on your computer before proceeding further.

1. On Windows 10, type “Settings” in your search bar and click on the Windows Settings application.
2. Go to “Devices” and you will see the window in Figure 2.9.
3. If “Bluetooth” is off, click it once to turn it on.

Note: Refer to Appendix D, Section 26 if you are using other operating systems other than Windows 10.

An Arduino board can be found wirelessly through Bluetooth communication. A Bluetooth module, such as the one in Figure 2.10, is required to connect with the Arduino board.
Wiring a Bluetooth module with an Arduino board: A Bluetooth module has four pins: GND, VCC, TXD, and RXD. Establish the following connections to obtain the physical setup shown in Figure 2.12:

- Connect the VCC pin on the module to the 5V pin on the microcontroller.
- Connect the GND pin on the module to one of the GND pins on the microcontroller.
- Connect the TXD pin on the module to the RX pin on the microcontroller.
- Connect the RXD pin on the module to the TX pin on the microcontroller.

Note: Do not swap the VCC and GND pins because that might damage the microcontroller. Once powered, the Bluetooth module should show a flashing red LED.

Once the setup is completed, the Bluetooth module will allow you to upload data to the Arduino remotely without the need of a USB cable. The flashing red light, as shown in Figure 2.11, indicates that the Bluetooth module is powered properly.

Warning: Remove the USB cable from the Arduino if you have not already. Do not power the Arduino with the USB cable and the battery at the same time because doing so causes some stability issues within the circuit.
Pairing a Bluetooth module with your computer: Select “Bluetooth” in “Device Type” box in ChDuino. Click the “Pair” button to open the “Pair Bluetooth Device” window as shown in Figure 2.13. To pair a Bluetooth Device, you can select “Manual add a Bluetooth Device” option on the first page of the window if you know the device’s address. Click the “Next” button to jump to the next page on which you can type in the address.

If you do not know the device’s address, select “Scan available Bluetooth devices” and click “Next”. In the following window, ChDuino will scan nearby Bluetooth devices and lists all devices found. Select a device in the list shown in the following figure and click “Pair”. After a device is selected, ChDuino will pair to the device with a default pairing code such as “1234”.

Note: The TX pin on the Arduino does NOT connect to the TXD pin on the Bluetooth module, but instead to the RXD pin. The same is true for the RX pin on the Arduino, which connects to the TXD pin on the Bluetooth module.
If the default codes fail, the program will ask you to type in the code manually as shown in Figure 2.16. You should be able to find the pairing code for your Bluetooth module in the user’s manual of the module or from the manufacturer’s website.

Finding paired Bluetooth module in ChDuino: After assembling and pairing the Bluetooth module, check the “Bluetooth” option in the “Device Type” box and click “Scan”. Since ChDuino is unable to tell the board type via Bluetooth, all devices found are listed as Arduino with the Bluetooth address.
2.10 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 indicates that the Arduino is properly connected. When testing a project, you should only focus on pins that have components attached.

![Arduino Interface](image)

2.11 Updating Firmware for the Arduino

Upload the CSTEM firmware to your Arduino before you start programming the microcontroller. Just as you click “Connect”, if you see a highlighted box that says “Update Firmware” just like in Figure 2.19, click on it, and then click on “Upload Firmware”. You should see the ‘Firmware Uploading Screen’ shown in Figure 2.20.
Figure 2.19: Updating the Firmware of the Arduino

Figure 2.20: Uploading Firmware

**Analog Pins**

Figure 2.21: ChDuino Analogpin Value

**Digital Pins**

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

Figure 2.22: 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 modulation, 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.
2.12 Using ChDuino to Test Hardware Setup

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.

- In ChDuino, set Digital Pin 3 to “Output” as shown in Figure 2.24. 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.

2.13 Exercises

1. In ChDuino, set 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 5V to the LED. "LOW" tells the Arduino to send 0V to the LED.

2. Plug the light into different digital pins and practice turning them on and off. Remember to include the resistor.
3 Programming Arduino in RoboBlockly and Ch

3.1 Project: Blink through RoboBlockly and Ch Programs

**Project Description:** This project will have an LED turn on, blink a couple of times, then turn off using a RoboBlockly program and a Ch program.

3.2 New Concepts

RoboBlockly is a graphical programming environment which features a simple puzzle-piece interface that makes coding visual and intuitive. When using RoboBlockly online, RoboBlockly can automatically generate a Ch file of the assembled code blocks, which can be further edited or run using the ChIDE on your computer. Alternatively, when launched from C-STEM Studio, RoboBlockly can automatically control an Arduino board.

To open RoboBlockly from C-STEM Studio, as shown in Figure 3.1, open C-STEM Studio and select RoboBlockly from the left hand selection bar, then click on launch to open the program.

![Figure 3.1: Opening RoboBlockly from C-STEM Studio](image)

**New Functions**

- `void pinMode(int pin, int mode)`
- `void digitalWrite(int pin, int value)`
- `void delay(int milliSeconds)`
3.3 Required Components and Materials

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

3.4 Getting Started

To demonstrate how RoboBlockly works, we will write a simple blink program. First, select the Arduino tab in the blocks tab, and drag a pinMode block onto the workspace as shown in Figure 3.2. The pinMode block is how we will determine which pins on the Arduino will act as inputs, or outputs. Notice that there are two arguments that can be changed on the block, a number from 2-13, and between INPUT and OUTPUT. The number corresponds to the pins labeled on the Arduino and the second choice corresponds to what we would like the pin to do. In this case, we have our LED connected on pin 3, and we would like to OUTPUT to the LED. Once you have tested your circuitry, you can write a program to control the Arduino board to perform various tasks.

![Figure 3.2: Adding Arduino Blocks](image)

Next, drag a digitalWrite block into the workspace, and place it under the pinMode block. RoboBlockly blocks run from top to bottom, just like reading a book. The digitalWrite block looks similar to the pinMode block with two arguments. The first dropdown menu is the same, with numbers that correspond to the pins labeled on the Arduino. The second choice now reads “HIGH” and “LOW”. Setting this to HIGH will provide power to the component on the selected pin, while setting it LOW will provide no power. In this case, we want to turn on the LED, so we set the first choice to “3” and the second choice to “HIGH”.

Next, drag and place a delay block under the previous blocks. The delay block will pause the program for a given number of milliseconds (1 second = 1000 milliseconds). The length of the delay can be adjusted by changing the number in the number block.

Finally, drag and place another digitalWrite block under the previous blocks. This time, set the digitalWrite block to LOW. This will turn off the LED after the one second delay.
To run this program, plug in the Arduino and make sure it is connected to ChArduino, then click the “Run” button below the grid.

RoboBlockly can also generate Ch code in real time. To see the Ch version of the blocks placed on the RoboBlockly workplace, click the “Show Ch” button on the top bar. You will notice the code appear on the right side of the screen.
Aside from the pinMode, digitalWrite, and delay blocks, there are several other Arduino related blocks that can be used. Table 3 lists the available Arduino blocks and briefly describes their function.

<table>
<thead>
<tr>
<th>Block</th>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pinMode</td>
<td>Pin Number and Pin Mode</td>
<td>Sets the specified pin number to the specified mode</td>
</tr>
<tr>
<td>digitalWrite</td>
<td>Pin Number and Value</td>
<td>Outputs value to the specified pin</td>
</tr>
<tr>
<td>digitalRead</td>
<td>Pin Number</td>
<td>Returns the value of the specified pin. Values are either 0 or 1.</td>
</tr>
<tr>
<td>analogWrite</td>
<td>Pin Number and Value</td>
<td>Writes an analog signal to the specified pin. Value can range from 0-255.</td>
</tr>
<tr>
<td>analogRead</td>
<td>Pin Number</td>
<td>Returns the analog value of the specified pin. Values can range from 0-1023.</td>
</tr>
<tr>
<td>delay</td>
<td>Duration</td>
<td>Delay the program by the given duration in milliseconds.</td>
</tr>
</tbody>
</table>

Table 3: Block Descriptions

RoboBlockly can also export the blocks on the workspace to a Ch file, which can then be further edited and run using the ChIDE, which will be described in detail in the next section. To save the block code in Ch, click the “Save Ch” button on the top bar. If you are running RoboBlockly from C-STEM Studio, the file will automatically open in ChIDE. If you are running RoboBlockly from a regular web browser such as Chrome or Edge, you will have to find the downloaded Ch file and open it using ChIDE. The next section will provide instructions for opening the C-STEM demonstration Ch files, and for writing your own programs using the ChIDE.
3.5 Exercises

1. Add another LED to the circuit, and use additional blocks to make both LED’s blink simultaneously.

2. Modify the blocks from the above exercise such that when one LED is on, the second LED is off.

3. Make the LED stay on for only a quarter of a second. Then, experiment with different amounts of time.

3.6 Accessing the Code in C-STEM Studio

First, navigate to the “Code in Curriculum” tab shown in Figure 3.9 and right-click on the “LearnArduino” folder. Click “Copy to Student Homework.”
Now, navigate to the “Student Homework” tab to view the folder you just copied. This folder is where any programs from the text should be executed from, and it is also where any edits or new versions of code can be created. Simply copy and paste the source folder from “Code in Curriculum” at any time to reset your code back to the original version. You should obtain the menu as shown in Figure 3.10.
Navigate to the “LearnArduino”, “program”, “Ch” folder and double-click on “blinkLED.ch” to launch the program.

Figure 3.10: Pasting Files to Student Homework

Figure 3.11: Launching blink.ch
3.7 Using ChIDE to Run Programs

ChIDE is an Integrated Development Environment for the Ch programming language. An IDE is an application that makes it easier to write, build, and run code. ChIDE is the IDE for Ch, but Arduino has their own IDE for their programming language. See Section 24 to learn more about Arduino IDE. 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 debug bar at the top. 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 3.12 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 output pane.

![ChIDE Window with Labels](image)

Figure 3.12: ChIDE Window with Labels

The last step before the computer and Arduino are ready to communicate with each other is that the ChArduino firmware needs to be uploaded to the Arduino. When a program is started, 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 upload the firmware 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 2.11.

3.8 Writing the Code

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

```c
// file: blinkLED.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
```
First it should be noted that the green text in the program 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. A function in programming resembles a function in math class because there are inputs and outputs. However, in programming the inputs and outputs can be different types of data. Also, the wording can be confusing, since inputs when writing the function are called “parameters” while inputs when the function is used are called “arguments”. This will become more clear as the book progresses.

#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 prototype, or 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. The int before pin and mode stands for “integer” (meaning whole numbers) to let the user know to pass in the correct data type. Remember that parameters are the variables listed in the function prototype, separated by commas, while the arguments are what get passed in to tell the function what to do or what to work on. The first parameter is an integer pin number, letting the function know what pin it is supposed to act on. The second parameter, 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 by the first argument an input or an output. This function is necessary for all 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.

```c
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.

```c
delay(1000);
```

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

```c
void delay(int milliSeconds);
```

There is only one argument for this function, an integer representing the number of milliseconds. The `delay()` function will pause the program for the specified number of seconds. In the previous segment of code, the argument was 1000 milliseconds so the `delay()` function will pause the program for 1 second. The purpose of this function 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.

```c
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.

### 3.9 Using ChIDE 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. 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 output pane, if it is not already open. Pressing the Step or Next buttons will enter the program into debug mode and highlight the first line, as shown in Figure 3.13. This will allow a programmer to proceed line by line through the code to verify each step, rather than the code speeding through as it normally does.
Figure 3.13: Step 1

The current line highlighted now should be the `pinMode(3, OUTPUT);`, which is being set to act as an output. Again, nothing should be happening on the breadboard yet. The program is still setting things up.
On the highlighted line in Figure 3.14, it seems that the LED should make a change to turn on now. This would make sense, as the line highlighted reads `digitalWrite(4, HIGH)`. However, the debug mode works by stopping just before the highlighted line. So in reality, the program only set the pin to an output so far, nothing else.
At this stage in Figure 3.15 the LED should turn on, as the program has stopped after the call to activate it.
In Figure 3.16, the delay of one second has been processed and the program is waiting to turn the LED off with `digitalWrite(4, LOW)`. Until the user continues the debugging process, the LED will stay on. However, only one more click of 'Next' will end the program and turn off the LED. Once completed, no lines will be highlighted anymore and the LED will be off.

### 3.10 Exercises

1. Change one line of the code to make the LED stay on for only a quarter of a second. Then, experiment with different amounts of time.

2. Add two lines to the code to turn the LED back on again after another second.
4 More Blinking

4.1 The Direct Method of Repetition

With only a few more lines of code, the same setup from Chapter 2 can be used to blink the LED a few more times. This lesson will be useful not only for LEDs, but for any output that must be repeated multiple times. The first way of causing an output to repeat is to manually write more lines of code to turn it on and off. Below is a continuation of the program from the previous chapter.

```c
// 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);

//Pause 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);
```

After the code from the previous section, new code is added that turns the LED on, waits, turns it off, waits, and does this again. Note that to make the LED blink only twice more required about eight more lines of code. Here, each blink occurs one second after the other. However, if we space the blinks out differently, we can send a message via Morse code.

4.2 SOS

This project again uses the setup from Chapter 2. The goal is to write out a message using Morse code, which is a language made up of a series of dots and dashes that can be translated into English letters and numerals. **Figure** 4.1 shows the full Morse Code alphabet.
International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

For this example, we will try to write out the letters “SOS,” which was an early use of the code by ships to signal that they needed help. It is a common phrase sent via Morse code, so it is useful to know. A dot will be represented by a one second flash of the LED, and a dash will be represented by a three second flash of the LED. The space between letters will also be three seconds, and the space between the dots and dashes will only be one second. The code to accomplish this is shown next.

```c
#include <arduino.h>

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

digitalWrite(3, HIGH);
```

Figure 4.1: Morse Code Alphabet
delay(1000);
digitalWrite(3, LOW);
delay(1000);
digitalWrite(3, HIGH);
delay(1000);
digitalWrite(3, LOW);
delay(1000);
digitalWrite(3, HIGH);
delay(1000);
digitalWrite(3, LOW);
// delay between letters 'S' and 'O'
delay(3000);

//Write Morse Code for 'O'
digitalWrite(3, HIGH);
delay(3000);
digitalWrite(3, LOW);
delay(1000);
digitalWrite(3, HIGH);
delay(3000);
digitalWrite(3, LOW);
delay(1000);
digitalWrite(3, HIGH);
delay(3000);
digitalWrite(3, LOW);
// delay between letters 'O' and 'S'
delay(3000);

//Write Morse Code for 'S'
digitalWrite(3, HIGH);
delay(1000);
digitalWrite(3, LOW);
delay(1000);
digitalWrite(3, HIGH);
delay(1000);
digitalWrite(3, LOW);
delay(1000);
digitalWrite(3, HIGH);
delay(1000);
digitalWrite(3, LOW);
// delay to finish the letter 'S'
delay(3000);

digitalWrite(3, HIGH);
delay(1000);
digitalWrite(3, LOW);
delay(1000);
digitalWrite(3, HIGH);
delay(1000);
digitalWrite(3, LOW);
delay(1000);
digitalWrite(3, HIGH);
delay(1000);
digitalWrite(3, LOW);
delay(3000);

To begin, use the same code setup as Chapter 2. Once this is done, the letter ‘S’ is sent.
The first eleven lines of the code simply turn the LED on and off three times. The general procedure is
turn on with `digitalWrite()`, leave it on for a dot or dash using `delay()`, turn it off with `digitalWrite()`
again, and finally wait a second for the next dot or dash using `delay()`. This is done however many times
is needed to create the letter or number desired. Recall that the `delay()` function accepts an argument of
‘1000’ to represent only one second. Because the ‘S’ is made of only three dots, the code repeats itself three
times. The ‘O’ is similar.

```c
digitalWrite(3, HIGH);
delay(3000);
digitalWrite(3, LOW);
delay(1000);
digitalWrite(3, HIGH);
delay(3000);
digitalWrite(3, LOW);
delay(1000);
digitalWrite(3, HIGH);
delay(3000);
digitalWrite(3, LOW);
delay(3000);
```

The only difference with the letter ‘O’ is that the delay while the the LED is on increases to three seconds
using an argument of ‘3000’ milliseconds. After this code, the same code for the ‘S’ appears one more time
to display the last letter. When run, the LED should turn on for a total of nine times in total, spelling out
“SOS” in Morse code.

While this direct line method of blinking an LED works perfectly fine, it is inefficient for scaling. For
example, what if the LED was to blink one hundred times? That would require over a hundred lines of code!
Instead, there is another way to continuously blink the LED.

### 4.3 Alternate Repetition 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 a 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. Note that there is another similar form of loop called a for-loop
that is not used in this book.

```c
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 < 3) {.....} The program will loop through the code inside
the while-loop until ‘i’ is greater than or equal to 3 and the condition becomes false. Although not used in
this text, a for-loop can perform the same actions with a more compact syntax. A while-loop can always be
used instead, though, so for-loops will no longer be mentioned.

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 3.8 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);

    //Pause 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, or iterator, 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 line has two operators, = and +, which both do exactly as expected. The equals operator sets the value on the left equal to the value on the right, and the plus operator returns the addition of the two numbers on either side of it. The < in the while-loop condition is also an operator. Looking at the iteration line again, 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.

4.4 Using an Infinite While-Loop

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.
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);

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

4.5 Debugging Revisited

Now that variables and loops have been introduced, it is important to track what is happening in ChIDE. For example, the programmer can look to see how a certain variable is changing or if a loop is acting the way it should. This section will go through the second version of the Blink project, with the (i < 3) while-loop condition, in debug mode to illustrate how it can be beneficial. As done previously, click Step or Next to begin debugging. The first line is simply a setup function, so click Step one more time to reach the second line.
At this point, `int i = 0;` should be highlighted as shown in Figure 4.2. 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 variables and their current values. This is very helpful for troubleshooting because it is easy to see how the variable are changing and pinpoint where something is going wrong. Press the Next button again and the program should move on to the next line.

![Debug Pane](image)

Figure 4.3

The `pinMode()` function should now be highlighted just like Figure 4.3. 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 program should move on to the creation of the while-loop. Press next one more time to highlight the first `digitalWrite()` function. The window should look like Figure 4.4.
Press the Next button to run the highlighted `digitalWrite()` function, causing the LED to 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 4.5.
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. The `i = i+1;` line should be highlighted. Run this line by pressing the Next button and obtain the setup in Figure 4.6.
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, let’s 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 while-loop or pressing Next over and over again. This can be accomplished using a breakpoint and the Continue button, found next to the Abort button. The Continue button will run the program like normal until it hits a breakpoint and then it stops. To insert a breakpoint, 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 breakpoint on that line. Insert a breakpoint on the line with $i = i + 1$ and the window should look like Figure 4.7.
Now hit the Continue button and the program should run until it hits the breakpoint. Because the program was left highlighted on the line with the `digitalWrite()` function, it will only run the few lines before hitting the breakpoint 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 breakpoint. The window should look like Figure 4.8.
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. 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. A value of 3 for ‘i’ no longer satisfies the while-loop condition that \( (i < 3) \). So, the while-loop finishes and the program ends.

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.

### 4.6 Multiple LEDs

So far in this chapter, only one LED has been used at a time by the program. Often it is useful to have multiple LEDs that can be turned on independently by the code. To do this, the physical setup from Chapter 2 must be modified slightly. Take a yellow LED this time and plug it into the breadboard in two separate and unused rows. Then, take another Red-Red-Brown (220 Ohm) resistor and plug one lead into the negative, or ground, rail marked (-). Remember that a rail refers to any one of the (+) or (-) columns on the sides. Take the other resistor lead and plug it into the row of the short, negative LED lead. Finally, take a wire and connect Digital Pin 4 to the row with the positive, or longer, lead of the LED. The final result is included
below along with the schematic.

With this new setup, new code can be written as shown to control each LED separately.

```c
// file: blinkTwoLED.ch  
// Alternate blinking 2 LEDs  
#include <arduino.h>  

//Set the LED pins to output mode  
pinMode(3, OUTPUT);  
pinMode(4, OUTPUT);  
```
while(1) {
  // Turn on one LED
  digitalWrite(3, LOW);
  digitalWrite(4, HIGH);
  delay(250);

  // Turn on the other
  digitalWrite(3, HIGH);
  digitalWrite(4, LOW);
  delay(250);
}

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

Using pinMode() twice, both LEDs are declared as outputs by using them as the first arguments in the function with “OUTPUT” as the second argument. Then, an infinite while loop is entered.

while(1) {
  digitalWrite(3, LOW);
  digitalWrite(4, HIGH);
  delay(250);

  digitalWrite(3, HIGH);
  digitalWrite(4, LOW);
  delay(250);
}

The first few lines turn the red LED (pin 3) off while switching the yellow LED (pin 4) on. This is done as before using digitalWrite() with the first argument being the pin and the second argument being either “LOW” or “HIGH”. Before continuing, a delay of 250 milliseconds is set using delay(). This quarter second delay ensures the blink is long enough for the user to see.

digitalWrite(3, HIGH);
digitalWrite(4, LOW);
delay(250);

The while loops ends by switching which LED is on. This is accomplished by reversing the function calls made in the previous lines, causing the active yellow LED to turn off while the inactive red LED turns on. Again, a quarter second delay is set. Then, the loop restarts, repeating this cycle infinitely until the user cancels the program.

Whether you are making one LED blink many times, or making many LEDs blink once, this chapter has covered the necessary functions to create the LED setups used in this book. The next step is to introduce user input to control LEDs and other devices attached to the Arduino.

4.7 Exercises

1. Modify “blink2.ch” so the LED will blink 10 times.

2. Modify the “blink3.ch” 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. Modify the “sos.ch” code so that the LED will spell out CSTEM.

4. Starting with the two LED setup, add another LED by wiring it to pin 5.
   (a) Modify “blinkTwoLED.ch” such that this third LED is on while the yellow LED is on.
   (b) Make the LEDs blink slower.
5 Controlling a Traffic Light with a Push-Button

5.1 Project: Traffic Light

**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, the traffic light will stay red when no input is given, and when the button is pressed, it will force the light to cycle through green, yellow, and back to red.

5.2 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 or switch, the computer will read a 1 when it is active or will read a 0 if it is not. To accomplish these tasks, this project will introduce if-else statements and a function to read digital inputs.

- **New Functions**
  - `int digitalRead(int pin)`

5.3 Required Components and Materials

- Breadboard
- 1 Push-Button Switch
- 1 10kΩ (Br-Bl-O) Resistor
- 3 220Ω (R-R-Br) Resistors
- 3 LEDs (Red, Yellow, and Green)

5.4 Building the Circuit

![Figure 5.1: Traffic Light Schematic Diagram](image-url)
Like the last project, the first step here is to 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.

Now it is time to plug the LEDs into the breadboard. Plug a red LED and yellow LED 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 yellow and red LEDs into the ‘4’ and ‘5’ pins, respectively, on the same side of the board to obtain the setup in Figure 5.3.
5.5 Writing the Code Without Input

Before moving on to the button, it would be a good idea to check if the LEDs are working properly without input. Below is the code for the Traffic Light project when left alone without a user input.

```cpp
// file: trafficLight.ch
// Simulate traffic light with red, yellow, and green LEDs.
// Display green light for 3s, yellow for 1s, then red for 3s.
#include <arduino.h>

// Set the LED pins to output mode
pinMode(5, OUTPUT); // red
pinMode(4, OUTPUT); // yellow
pinMode(3, OUTPUT); // green

// Traffic turns green for 3 seconds, then yellow for 1 second, then red for 3 seconds
while(1){
    digitalWrite(3, HIGH); // green light
    delay(3000);
    digitalWrite(3, LOW);
    digitalWrite(4, HIGH); // yellow light
    delay(1000);
    digitalWrite(4, LOW);
    digitalWrite(5, HIGH); // red light
    delay(3000);
    digitalWrite(5, LOW);
}
```

The first step, as in all of the code, is to include `<arduino.h>` header file, followed by declaration of the output pins.

```cpp
pinMode(5, OUTPUT);
```
```c
pinMode(4, OUTPUT);
pinMode(3, OUTPUT);
```

The previous code segment assigns each LED pin as an output using the `pinMode()` function. Pins D5, D4, and D3 are all set to ‘OUTPUT’ and represent 3 different LED colors. Pin D5 is red, D4 is yellow, and D3 is green.

```c
while(1){
    digitalWrite(3, HIGH);
    delay(3000);
}
```

After the setup, the program enters into an infinite while-loop in which it turns pin D3 to ‘HIGH’. The code inside of the while-loop, which is inside of the braces ..., will run forever until the user stops the program. Pin D3 is associated with the green light, and thus turns the LED on. A `delay()` follows this line, which lasts for 3000 milliseconds, or 3 seconds. Once the traffic light has been green for three seconds, it enters a yellow light.

```c
digitalWrite(3, LOW);
digitalWrite(4, HIGH);
delay(1000);
```  
Remember that the digital pins will not stop giving output until explicitly stated. In this code, the first `digitalWrite()` statement is used to turn the green LED off before moving on. The next line turns the yellow LED, connected to D4, to the active ‘HIGH’ state. Since yellow lights at intersections are generally more brief than green or red, the delay here is only 1000 milliseconds, or 1 second, using the `delay()` function. After a yellow light, the traffic light now needs to turn red.

```c
digitalWrite(4, LOW);
digitalWrite(5, HIGH);
delay(3000);
digitalWrite(5, LOW);
}
```  
As done previously with the green LED, the first line turns off the yellow LED by writing a value of ‘LOW’ to that pin. Next, the red LED, attached to pin D5, is set active using a ‘HIGH’ value. Like the green light, the red light LED remains active for 3 seconds using a `delay()` call for 3000 milliseconds. Finally, the red LED is turned off with a `digitalWrite()` function before closing the while-loop.

If written correctly, the while-loop should continue indefinitely. It will cycle between green, yellow, and red LEDs, staying on green and red for 3 seconds and on yellow for 1 second. At this point, it is time to control the lights with user input via a push-button.

### 5.6 Building the Circuit (Part 2)

Now, 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 to obtain the setup in Figure 5.4.

![Figure 5.4: Step 3](image)

<table>
<thead>
<tr>
<th>Pin</th>
<th>Code</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2</td>
<td>2</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td>3</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>D4</td>
<td>4</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>D5</td>
<td>5</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4: Traffic Light Project Pin Assignment Chart

5.7 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.

![Figure 5.5: Setting D2 to “Input”](image)

- 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 the Blink project, you can turn the LED lights on and off using the “HIGH” and “LOW” buttons.
5.8 Writing the Code for Input

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

```c
// file: trafficControl.ch
// Control traffic light with a button push
// When the button is pushed, only the green light is on.
#include <arduino.h>

int switchState = 0;

//Set the LED pins to output mode
pinMode(5, OUTPUT); //red
pinMode(4, OUTPUT); //yellow
pinMode(3, OUTPUT); //green
//Set the switch pin to input mode
pinMode(2, INPUT);

while(1) {
    //Read the state of the switch
    switchState = digitalRead(2);

    //If the button is not pushed, turn the red LED ON and everything else OFF
    if (switchState == 0){
        digitalWrite(5, HIGH); //red light
        digitalWrite(4, LOW);
        digitalWrite(3, LOW);
    }

    //If the button is pushed, traffic turns green for 3 seconds,
    //then turns yellow for 1 second, then red
    else {
        digitalWrite(5, LOW); //red OFF
        digitalWrite(3, HIGH); //green light
        delay(3000);
        digitalWrite(3, LOW);
        digitalWrite(4, HIGH); //yellow light
        delay(1000);
        digitalWrite(4, LOW);
    }
}
```

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

```c
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.

```c
pinMode(5, OUTPUT);
pinMode(4, OUTPUT);
pinMode(3, 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 by passing the pins in as arguments.

```
while(1){

Now that the pin modes are set, the program goes into the infinite while-loop. The code inside of the loop will run continuously forever.

``` switchState = digitalRead(2);

Now to start the code inside of the while-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 18. 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 single board computer 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(5, HIGH);

digitalWrite(4, LOW);

digitalWrite(3, LOW);
}
```

The program now needs to tell the Arduino 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 == 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. These three lines inside of the if-statement above will turn pin 5 high and turn pins 3 and 4 low, meaning that the red LED will be turned on and the yellow and green LEDs will be turned off. The LEDs will remain in this state indefinitely until a button is pressed, making the 'switchState' flag equal to 1.

```
else {

digitalWrite(5, LOW);

digitalWrite(3, HIGH);
}
```
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 only run the if-statement, completely ignoring the else-statement. So, in the case when the button is pressed (when 'switchState' is equal to 1) the program runs the else-statement. If 'switchState' 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. Notice that the code here is almost exactly the same as the no-input section of this chapter. The code explanation can be found there.

When the program is run, the traffic light should remain red until the button is pressed, at which point the light will turn green, then yellow, and back to red until input is given again. 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. 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 laughable mistakes, like forgetting a semi-colon at the end of a line or misspelling a function. This quickly teaches the programmer all the little syntax quirks of a programming language, so the more someone debugs, the better programmer they become.

5.9 Exercises

1. Try building the circuit with different colored resistors. How does this change the brightness of the lights?

2. Change the “if” parameter to check for “1” instead of “0”. How does the button function now?

3. Adjust the “else” code such that the green and yellow light blink back and forth every 100 milliseconds until the button is released.

4. Add in two more LEDs, one red and one green, that represent the crosswalk signal in the direction of traffic. The new green LED should be on whenever it is a green light, and the new red LED should be on whenever it is a yellow or red light. Use pins 6 and 7 to accomplish this.
6 Using a Potentiometer to Dim an LED

6.1 Project: Dimmer

**Project Description:** For this project, an LED’s brightness will be controlled by turning a potentiometer.

6.2 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 LED’s brightness can be done by applying more or less voltage, known as analog output. 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 imitate a range of voltages, thus changing the brightness of a connected LED.

**New Functions**

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

6.3 Required Components and Materials

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

6.4 Building the Circuit

![Dimmer Schematic Diagram](image)

Figure 6.1: Dimmer Schematic Diagram

Remember to supply 5V and GND to the vertical rails on the breadboard.
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. You should then have the setup in Figure 6.2.

![Figure 6.2: Step 1](image)

The next step is to setup the potentiometer. There are three pins in a straight line on the bottom of the potentiometer. Plug the potentiometer into the breadboard with each pin on its own row. Connect one of the two outer pin to the ‘+’ column of the breadboard and the other outer pin to the ‘-’ column, it does not matter which. Finally, use a wire to connect the middle pin of the potentiometer to the ‘A0’ pin of the Arduino.

Note that some potentiometers have two leads coming out of one side and one lead coming out of the other. These potentiometers need to be placed so that the leads are spanning the trench, meaning that the side with two leads is on one side of the trench and the side with one lead is on the other. The outer two pins are still connected to ‘+’ and ‘-’, and the middle pin is connected to A0 of the Arduino.
6.5 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 modulation (PWM). Pins that are capable of PWM will have a ‘˜’ symbol next to them, as you can see in ChDuino and Figure 6.4.

- 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.

This project also introduces the concept of analog input. Analog values read voltages and convert that value to an integer between 0 and 1023. The number 1023 may seem random, but it actually comes from how computers represent numbers. Computers use binary, the language of bits, to represent everything. With 10 bits for each analog pin, we can get $2^{10} = 1024$ different numbers since each bit can hold 2 values.

### Table 5: Dimmer Project Pin Assignment Chart

<table>
<thead>
<tr>
<th>Pin</th>
<th>Code</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A0</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td>3</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.3: Step 2

Figure 6.4: PWM
Because we include 0, this gives us the 0-1023 range. In the GUI, you can see both the raw voltage and this 10 bit value. Using the code in the next section, the Arduino can convert these analog values to real-world temperatures.

![Figure 6.5: Analog Pin](image)

### 6.6 Writing the Code

The following is the code for this project:

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

#include <arduino.h>

// Declare variables for pin assignments
// to hold values from the potentiometer
// and hold values to be written to the LED
int LEDpin = 3,
    potPin = A0,
    potVal = 0,
    LEDval = 0;

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

while(1){
    // 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/1023.0*255;
    printf("LEDval: %d\n", LEDval);

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

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

```c
int LEDpin = 3,
    potPin = A0,
    potVal = 0,
    LEDval = 0;

pinMode(LEDpin, OUTPUT);
```

The program begins by creating four integers. ‘LEDpin’ and ‘potPin’ are used to hold the pin numbers connected to the LED and the potentiometer, respectively. Another, called ‘potVal’, will hold the input value that will be read from the potentiometer. The last variable, ‘LEDval’, will hold the PWM value to written to the LED.
The setup uses `pinMode()` to set the mode of the LED pin to output mode.

```c
while(1){
    potVal = analogRead(potPin);
    LEDval = potVal/1023.0*255;
    printf("LEDval: %d\n", LEDval);
    analogWrite(LEDpin, LEDval);
}
```

The infinite while-loop 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 1023, the max size of values that can possibly be read, giving a ratio. This ratio is then multiplied by 255, the max size of the 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 PWM, the `analogWrite()` function is utilized. The prototype is shown below.

```c
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.

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

### 6.7 Exercises

1. Pick a random number between 0 and 1023. Try to produce that analog input number using the potentiometer. How close can you get?

2. Add a green LED that operates in reverse of the red LED. In other words, when one is dim, the other should be bright. Both should be controlled with the same potentiometer.

3. Use the temperature sensor from the previous project to make a hot/cold warning light that scales with brightness and temperature.
7 Data Acquisition and Plotting Using a Photo-resistor

7.1 Project: Graphing Light

**Project Description:** This project will measure and plot input values from a photo-resistor.

7.2 New Concepts

Arrays are a powerful tool that are useful in almost every application, including graphing. This project will detail how to use arrays to collect data from a photo-resistor sensor and 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)`

7.3 Required Components and Materials

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

7.4 Learning How to Plot Using plotArray.ch

Before jumping into this project, it would be best to understand how Ch can be used to plot graphs. Below is an example program that will clearly illustrate how it works by plotting the function at the top of the program:

```c
/* File: plotArray.ch
   Plot the polynomial y = 2x^2-20x+55 for x from 0 to 10.*/
#include <chplot.h>
int i = 0, num = 100;
array double x[num], y[num];
CPlot plot;

while(i < num) {
    x[i] = 10.0*i/(num-1);
    y[i] = 2*x[i]*x[i] - 20*x[i] + 55;
    i = i + 1;
}

plot.title("y=2x^2-20x+55");
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
plot.data2DCurve (x, y, num);
plot.plotting();
```
Note that *arduino.h* is *not* included as a header file in this program because it works without the use of any digital pins.

```c
#include <chplot.h>
```

Instead, it is necessary to include *chplot.h* as it contains all of the functions that will be used to create plots using given data.

```c
int i = 0, num = 100;
array double x[num], y[num];
```

First, let’s review what an array is. An array is a variable that contains multiple constant values. You can declare the size of an array and store values at different places in the array, starting with 0 as the index. For example, an array containing 100 words starts at index [0] and ends at index [99]. In this project, you will be using an array of integers.

Variables are declared. First is an integer ‘i’ which begins at 0 and will be used as a counter. Second, a variable ‘num’ is set to 100 to represent the size of the arrays, which are the next two variables. ‘x’ and ‘y’ are arrays of type *double* that will store the data points on their respective axes. In other words, to get an ordered pair or point on the graph like (x, y), take the number stored at the same index in both arrays. So, at the same *index* value, if ‘x’ gives 7 and ‘y’ gives 10, this is the ordered pair (7, 10).

```c
CPlot plot;
```

Before adding data to a graph, a ‘plot’ variable must be created of type *CPlot*. *CPlot* is actually a class, which changes the way we use functions with it. You will see more applications of classes in the following projects with the *Servo* class, and it will be used again for LCDs. Finally, it is time to write the code to calculate the data.

```c
while(i < num) {
    x[i] = 10.0*i/(num-1);
    y[i] = 2*x[i]*x[i] - 20*x[i] + 55;
    i = i + 1;
}
```

This short while-loop does 3 things for every spot in the array. First, it sets a data point for the x-axis by multiplying by 10 and dividing it by one less than the total number of elements. This creates a number that goes from 0 to 10 in increments of 1/num. Second, the y-axis value corresponding to that x-axis value is generated using the polynomial given in the beginning. Third, the value of i is incremented, or iterated, by 1 to ensure the while-loop eventually reaches the end and exits. Now it is time to plot these arrays.

```c
plot.title("y=2x^2-20x+55");
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
```

A good idea before putting the data in the plot is to set it up properly. A title is given using the *title()* function, which accepts a string as an argument. Next, the x-axis label is set using the *label()* function. Two arguments are necessary for this. This first is the name of the axis, in this case ‘‘PLOT_AXIS_X’’ to refers to the x-axis, and the second is the string which is used to label that axis on the image of the graph. To keep things simple, it is named “x”, but it could be anything.

Notice that because ‘plot’ is a CPlot class variable, we call functions on it using *plot.function()*.
Classes are incredibly powerful data structures because they hold many types of data and function.

```c
plot.data2DCurve (x, y, num);
plot.plotting();
```
Finally, the data is applied and plotted. `data2DCurve()` is used to add in the data. The general form is shown below.

```c
void CPlot.data2DCurve(array double x[], array double y[], int n)
```

In this program, we feed the ‘x’ and ‘y’ arrays for those axes, and use ‘num’ amount of points, n. Now that the data is in, the program must be told to compute the graph and display the data. This is done with `plotting()`. If everything is done correctly, the following graph should appear:

![Graph](image-url)

Figure 7.1
7.5 Building the Circuit

The circuit for this project be identical to the one used in Section 14 aside from the piezo.

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 to obtain the circuit shown in Figure 7.3.
Table 6: Graphing Light Project Pin Assignment Chart

<table>
<thead>
<tr>
<th>Pin</th>
<th>Code</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A0</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

7.6 Writing the Code

The following is the code for this project:

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

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

// Declare variable for pin assignment and to keep track of location
// inside of the arrays
int sensorPin = A0,
    element = 0,
    size = 100;

// Declare variable for sampling interval
double interval = 0.1*1000; //converts 0.1 second to milliseconds

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

while(element < size){
```
// 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;

// Print out the data from this pass through the while-loop
printf("Sensor Value: %lf Time: %lf\n", light[element], timeSec[element]);

// Delay the program for 0.1 seconds before repeating the loop
delay(interval);
    element = element + 1;
}

// 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();

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.

int sensorPin = A0,
    element = 0,
    interval = 100,
    size = 100;

First, an integer 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
The idea with this while-loop 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.

Now for the code inside of the while-loop. 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 while-loop, is multiplied by ‘interval’, giving the time in seconds since the program started repeating the while-loop. The reason the number is ‘interval’ is because later in the code, at the end of the while-loop, the program is paused for ‘interval’ number of seconds. Thus in the pre-set case, the program goes through the while-loop roughly every 0.1 seconds. So, multiplying the current ‘element’ value by 0.1 gives the total number of seconds since the while-loop started. Finally, the number is assigned to the current element within the ‘timeSec’ array after being divided by 1000 to convert to milliseconds.

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 the 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 function prototype is shown below.

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

The title() function only has one argument, a string, 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 prototype is shown below.

```c
CPlot.plot.title("Light vs. Time");
CPlot.plot.label(PLOT_AXIS_X, "Time(s)");
```
void CPlot.label(int axis, string_t label)

The first argument tells 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 prototype is shown below.

void CPlot.data2DCurve(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 prototype 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 its 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 7.4 on the next page. 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?

![Graph 7.4: Graphing Light Project Example Graph](image)

7.7 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 are the potential 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 are multiple ways to accomplish this.
8 Using Photo-Resistors and Potentiometers to Change the Brightness and Color of an RGB LED

8.1 Project: Color Mixing Lamp

**Project Description:** For this project, an 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 each photo-resistors.

8.2 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.

8.3 Required Components and Materials

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

8.4 Building the Circuit

![Color Mixing Lamp Schematic Diagram](image)

Figure 8.1: Color Mixing Lamp Schematic Diagram

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. Then, plug the other end of the wire into the ground rail, 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. Refer to Figure 8.4 when assembling the physical setup.

**Note:** Know the type of RGB LED that you are using beforehand. There are two types of RGB LED: common anode and common cathode. For the purpose of this example, the common anode RGB LED will be used. The “common” pin of the LED will be wired to either 5V or Ground depending on whether the LED is common anode or common cathode. Refer to Figure 8.2 and Figure 8.3 once you have determined the type of RGB LED that you will be using.

The leads on the RGB LED, as indicated in Figure 8.2, consist of three corresponding RGB pins that will determine the amount of red, green and blue colors that will be lit up, and one common pin placed between the red and the green pin. The common pin will be wired to ground for a common cathode LED, and to 5V for a common anode LED. 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.

Figure 8.2: RGB LED (Common Anode) Pinout Diagram

Figure 8.3: RGB LED (Common Cathode) Pinout Diagram

---

81
The last part of the build is to set up the photo-resistor and potentiometers. The photo-resistor looks like a flat disk with leads coming out of one side and a squiggle on the other. Take the photo-resistor and plug it into the breadboard so that all leads are plugged into their own row, like in Figure 8.6.

Take a wire and plug one end into the same row as one of the leads of the photo-resistor. Plug the other end of the wire into the ‘5V’ power column.

Take another wire and plug one end into the row with the photo-resistor and Brown-Black-Orange resistor. The other end of this wire should be plugged into the analog pin such that the photo-resistor’s wire is connected to analog pin A0.

Finally, plug two potentiometers into the breadboard. For each potentiometer, connect the two outer pins to power and ground, it does not matter which lead is plugged to power or ground, and connect the middle pin to an analog input pin with one potentiometer connected to A1 and the second connected to A2.

Please note that the wiring will be different depending on the type of RGB LED that you are using. If you are using a common anode LED, wire the common pin to 5V. Refer to Figure 8.7 for more detail.
Figure 8.5: Step 2 Common Anode

Figure 8.6: Step 2 Common Cathode

Figure 8.7: Common Anode Versus Common Cathode Pin Guide
<table>
<thead>
<tr>
<th>Pin</th>
<th>Code</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>A0</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>A1</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>A2</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>D9</td>
<td>9</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>D10</td>
<td>10</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>D11</td>
<td>11</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Color Mixing Lamp Project Pin Assignment Chart

8.5 Writing the Code

The following is the code for this project:

```c
// file: colorMixingLamp.ch
// control the brightness of the three colors of an RGB LED based on a
// photoresistors and two potentiometers

#include <arduino.h>

//Declare variables for pin assignment and to hold the raw sensor and
//transformed values for each LED
int blueLEDpin = 9,
    greenLEDpin = 10,
    redLEDpin = 11,
    greenSensorPin = A0,
    blueSensorPin = A1,
    redSensorPin = A2,
    greenVal,
    greenSensorVal,
    redVal,
    redSensorVal,
    blueVal,
    blueSensorVal;

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

while(1){
  //Read the input values from each sensor and print them out
  greenSensorVal = analogRead(greenSensorPin);
  delay(5);
  redSensorVal = analogRead(redSensorPin);
  delay(5);
  blueSensorVal = analogRead(blueSensorPin);
  printf("Raw Sensor Values \t Red: %d\t Green: %d\t Blue: %d\n", redSensorVal,
             greenSensorVal, 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;
```

84
redVal = redSensorVal/4;
blueVal = blueSensorVal/4;

printf("Mapped Sensor Values \t Red: %d\t Green: %d\t Blue: %d\n", redVal,
        greenVal, blueVal);

//Write the appropriate voltage based on the sensor input to each LED
analogWrite(greenLEDpin, greenVal);
delay(50);
analogWrite(redLEDpin, redVal);
delay(50);
analogWrite(blueLEDpin, blueVal);

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

int greenLEDpin = 9,
    blueLEDpin = 10,
    redLEDpin = 11,
    greenSensorPin = A0,
    blueSensorPin = A1,
    redSensorPin = A2,
    greenVal,
    greenSensorVal,
    redVal,
    redSensorVal,
    blueVal,
    blueSensorVal;

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 variables keep track of the pins and pin numbers much
easier.

while(1){
    greenSensorVal = analogRead(greenSensorPin);
delay(5);
    redSensorVal = analogRead(redSensorPin);
delay(5);
    blueSensorVal = analogRead(blueSensorPin);

    printf("Raw Sensor Values \t Red: %d\t Green: %d\t Blue: %d\n",
            redSensorVal,
            greenSensorVal, 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 while-loop. 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 with the help of a photo-resistor and potentiometer. The values read from the sensors are stored in the three `..SensorVal` variables created earlier. In between reading the sensors, the program pauses the program for a small amount of time, 5 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 `printf()` and 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 there is a `\t` inside the print statement. This is a special character that tells the `printf()` function to insert a tab space.

```c
    greenVal = greenSensorVal/4;
    redVal = redSensorVal/4;
    blueVal = blueSensorVal/4;
    printf("Mapped Sensor Values \t Red: %d\t Green: %d\t Blue: %d\n", redVal, greenVal, blueVal);
```

The values taken from the sensors 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 and divides it by four (because 1024 divided by 256 is 4), and then 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.

```c
    analogWrite(greenLEDpin, greenVal);
    delay(50);
    analogWrite(redLEDpin, redVal);
    delay(50);
    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. The `delay(50);` is added to avoid erratic behavior.

If everything is correct, the RGB LED should change color depending on which photo-resistor is covered and on the position of the potentiometer. 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.

**8.6 Exercises**

1. Set the digital pins (D9-11) connected to the RGB LED to “PWM,” and practice changing the color of the LED by making different combinations of digital outputs.

2. Instead of the RGB LED, use individual yellow, green, and red LEDs.

3. Replace one photo-resistor with a temperature sensor to control one of the RGB LED colors.

4. Use three buttons to control each of the RGB LED colors.

5. Use three photoresistors or potentiometers to control each color on the RGB LED. Does the code need to be modified? Why or why not?
Part II:

Physical Computing with Robots