S.I.K. BINDER

Table of Contents

Table of Contents //

Page

1

Getting Started

Sparkfun Inventor's Kit Teacher's Helper

These worksheets and handouts are supplemental material intended to make the educator's job a little easier by providing easily editable content. You can use these files however you see fit. Add a question here, delete a question there and definitely add some graphics if you like. The worksheets are intended for use after completion of the SparkFun Inventor's Kit or as you go along. There is no particular order so you can use whichever worksheets you wish, whenever you think is best.

The SparkFun Inventor's Kit is a great introductory tool to get people interested in electronics and physical computing. This is the first collection of worksheets that pertain to the SparkFun Iventors Kit. We would appreciate any feedback you feel would be useful. Topics we missed, projects you put together using the SparkFun Inventor's Kit, typos, gripes, material you have put together about this type of technology that you would like to share or stuff that was really, really useful. Basically we want you and your students to eventually be able to build robots that can sing, dance and take over the world... or at least your imagination.

Included material:

- Worksheets and handouts for S.I.K. circuits 1 – 14
- Answers to worksheets (creative answers left blank, Ohm's Law answers may vary)
- Expansion code for use in Arduino
- Images to create your own schematics
- Surveys for teachers and students to aid in development of material
- Fritzing (virtual prototyping software)

This material is a work in progress - feel free to contribute if you are so inclined.

Send feedback, worksheets or completed surveys to:

EdMaterials@Sparkfun.com

Attention Lindsay Craig
Department of Education
6175 Longbow Drive
Boulder, Colorado 80301

Material by: Lindsay Craig, Jim "The Engineer" Lindblom, and Ben Leduc-Mills
Design and Layout by: Nic Bingham
Images by: Nic Bingham, Dave Stadler and Lindsay Craig
Proofed by: Ben Leduc-Mills, Lindsay Craig, Jim "The Engineer" Lindblom, Chris "Cmac" McGrady, Michelle Shorter, Toni Klopfenstein, SparkFun's IT department, David Stadler, Jeff Branson, Lindsay Levkoff, Amanda Clark, and other equally awesome people.

To attribute this work please copy and paste the following into your credits section:

Created from SparkFun Electronic's SIK Binder

Material by: Lindsay Craig, Jim "The Engineer" Lindblom, and Ben Leduc-Mills
Design and Layout by: Nic Bingham
Images by: Nic Bingham, Dave Stadler and Lindsay Craig
Proofed by: Ben Leduc-Mills, Lindsay Craig, Jim "The Engineer" Lindblom, Chris "Cmac" McGrady, Michelle Shorter, Toni Klopfenstein, SparkFun's IT department, David Stadler, Jeff Branson, Lindsay Levkoff, Amanda Clark, and other equally awesome people.

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

To view a copy of this license visit:
http://creativecommons.org/licenses/by-nc-sa/3.0/

Or send a letter to:
Creative Commons, 171 Second Street, Suite 300, San Francisco, California

// Installing Arduino

Mac platform

1. Double click the file **arduino-0022.dmg** inside the folder \SIK Applications\Mac\

2. Go to "Arduino" in the devices section of the finder and move the "Arduino" application to the "Applications" folder.

3. Go to the "Arduino" device, double click and install:
"FTDI drivers for Intel Macs 0022.pkg"
or
"FTDI drivers for PPC Macs 0022.pkg"
then Restart your computer.

4. Plug your Arduino board into a free USB port using the USB cord provided.

PC platform

1. Unzip the file **arduino-0022** inside the folder \SIK Applications\PC\. We recommend unzipping to your c:\Program Files\ directory.

2. Open the folder containing your unzipped Arduino files and create a shortcut to Arduino.exe. Place this on your desktop for easy access.

3. Plug your Arduino board into a free USB port using the USB cord provided. Wait for a pop up box about installing drivers.

4. Skip "searching the internet." Click "Install from a list or specific location" in the advanced section. Choose the location c:\program files\arduino-0022\drivers\Arduino Uno\

(You may have to do this last step more than once) (If you are using the Duemilanove you will have to choose the sub-directory, FTDI USB Drivers and you will have to do this twice)

If you are having issues with Java make sure you have the latest version of Java installed. If not you're ready to open the Arduino programming environment.

(For Linux info go to www.arduino.cc/playground/learning/linux)

// Installing Fritzing

Mac platform

1. Move the Fritzing folder from \SIK Applications\Mac\ to somewhere convenient on your computer.

2. Double click the file: **fritzing.2010.09.30.mac**

PC platform

1. Move the Fritzing file from \SIK Applications\PC\ to somewhere convenient on your computer.

2. Double click the file: **fritzing.2010.09.30.pc**

You're ready to start using Fritzing for virtual prototyping.

// A few notes about setup

A few more tidbits that will help to know
There are seven buttons at the top of your Arduino window, and these are their functions:

Compile
This checks your code for errors.

Stop
This stops the program.

New
This creates a new sketch.

Open
This opens an existing sketch.

Save
This saves the open sketch.

Upload
This uploads the sketch to your Arduino.

Serial Monitor
Used to display Serial Communication.

Selecting Your Board

You are using the SparkFun RedBoard with an ATmega 328 microcontroller. This means you will need to select "Arduino Uno" as your board. To do this you click on the "Tools" menu tab, then click the "Board" tab and select "Arduino Uno". If you are using a different board you will need to select the correct model in order to properly upload to your board.

Selecting Your Com Port

Another option that is necessary to change occasionaly is your "Serial Port". This can also be found under the "Tools" menu tab. When you click on this tab you should be presented with at least one com port labeled "COM1" (or "COM2," etc....) This indicates which USB port your board is plugged into. Sometimes you will need to make sure you are using the correct com port. Here is some information on your com ports depending on which platform you are using:

Mac Platform

The Mac version of the Arduino IDE refreshes your com port list every time you plug in a device. For this reason all you really need to do is select the com port called "/dev/cu.usbserial-XXXX" where XXXX will be a value that changes.

PC Platform

The PC version of the Arduino IDE creates a new com port for every distinct board you plug into your computer. You will need to find out which com port is the board you are currently trying to use. This is likely to be COM3 or higher (COM1 and COM2 are usually reserved for hardware serial ports). To find out, you can disconnect your Arduino board and re-open the menu; the entry that disappears should be the Arduino board. Reconnect the board and select that serial port.

Installation

Arduino: http://www.arduino.cc/en/Guide/HomePage
Fritzing: http://fritzing.org/download/

Support

Arduino: http://www.arduino.cc, http://www.freeduino.org
Fritzing: http://www.fritzing.org/learning/

Forums

Arduino: http://forum.sparkfun.com/viewforum.php?f=32
Fritzing: http://fritzing.org/forum/

Basic Arduino Code Definitions

setup( ): A function present in every Arduino sketch. Run once before the loop( ) function. Often used to set pinmode to input or output. The setup( ) function looks like:

```
void setup( ){
        //code goes here
        }
```

loop( ): A function present in every single Arduino sketch. This code happens over and over again. The loop( ) is where (almost) everything happens. The one exception to this is setup( ) and variable declaration. ModKit uses another type of loop called "forever( )" which executes over Serial. The loop( ) function looks like:

```
void loop( ) {
        //code goes here
        }
```

input: A pin mode that intakes information.

output: A pin mode that sends information.

HIGH: Electrical signal present (5V for RedBoard). Also ON or True in boolean logic.

LOW: No electrical signal present (0V). Also OFF or False in boolean logic.

digitalRead: Get a HIGH or LOW reading from a pin already declared as an input.

digitalWrite: Assign a HIGH or LOW value to a pin already declared as an output.

analogRead: Get a value between or including 0 (LOW) and 1023 (HIGH). This allows you to get readings from analog sensors or interfaces that have more than two states.

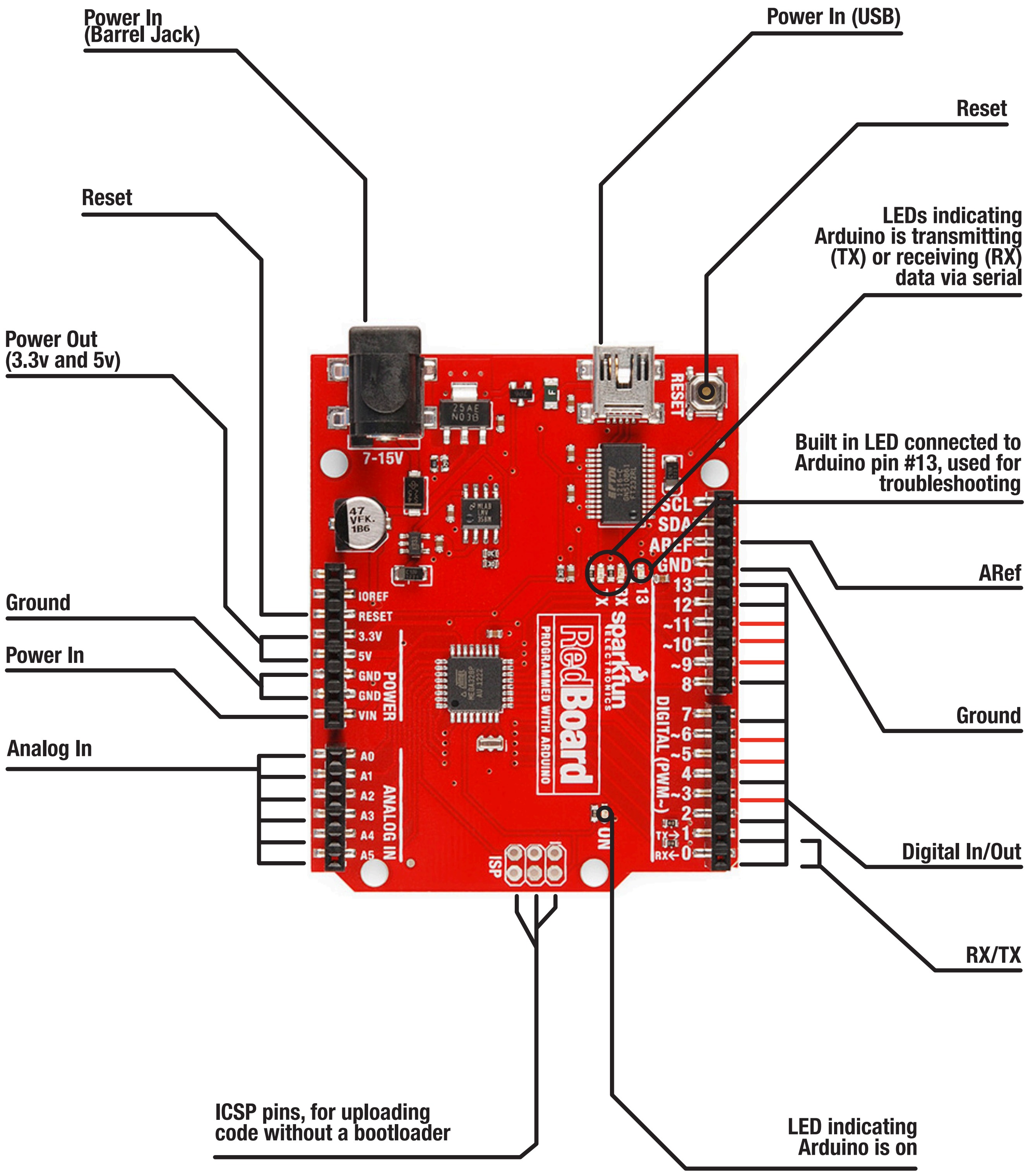
analogWrite: Assign a value between or including 0 (LOW) and 255 (HIGH). This allows you to set output to a PWM value instead of just HIGH or LOW.

PWM: Stands for Pulse-Width Modulation, a method of emulating an analog signal through a digital pin. A value between or including 0 and 255. Used with analogWrite.

RedBoard Pin Type Definitions: (Take a look at your board)

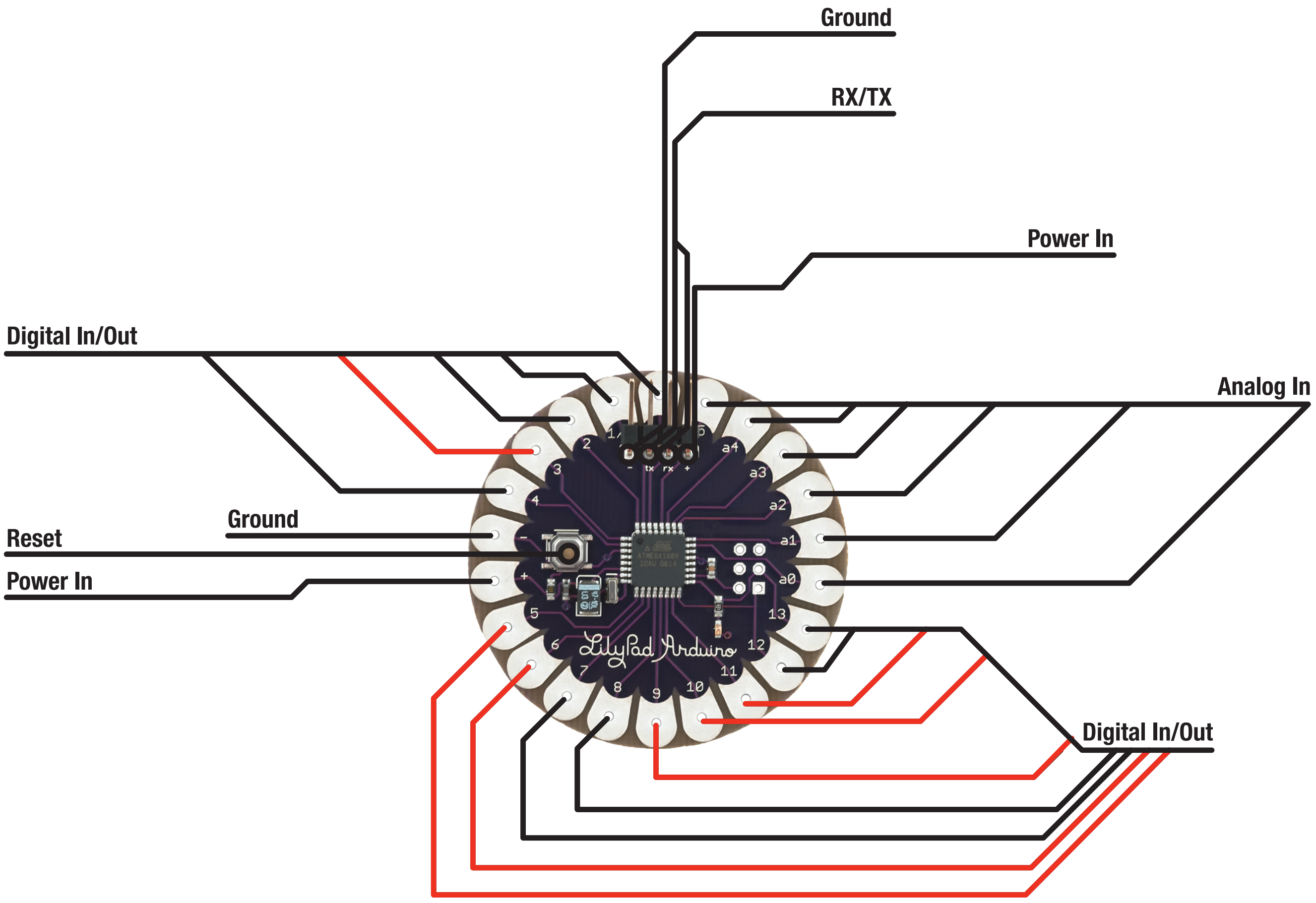
Reset	3v3	5v	Gnd	Vin	Analog In	RX/TX	Digital	PWM(~)	AREF
Resets Arduino sketch on board	3.3 volts in and out	5 volts in and out	Ground	Voltage in for sources over 7V (9V - 12V)	Analog inputs, can also be used as Digital	Serial comm. Receive and Transmit	Input or output, HIGH or LOW	Digital pins with output option of PWM	External reference voltage used for analog

Basic Arduino Pin Reference Sheet



Red lines indicate which pins are PWM compatible.

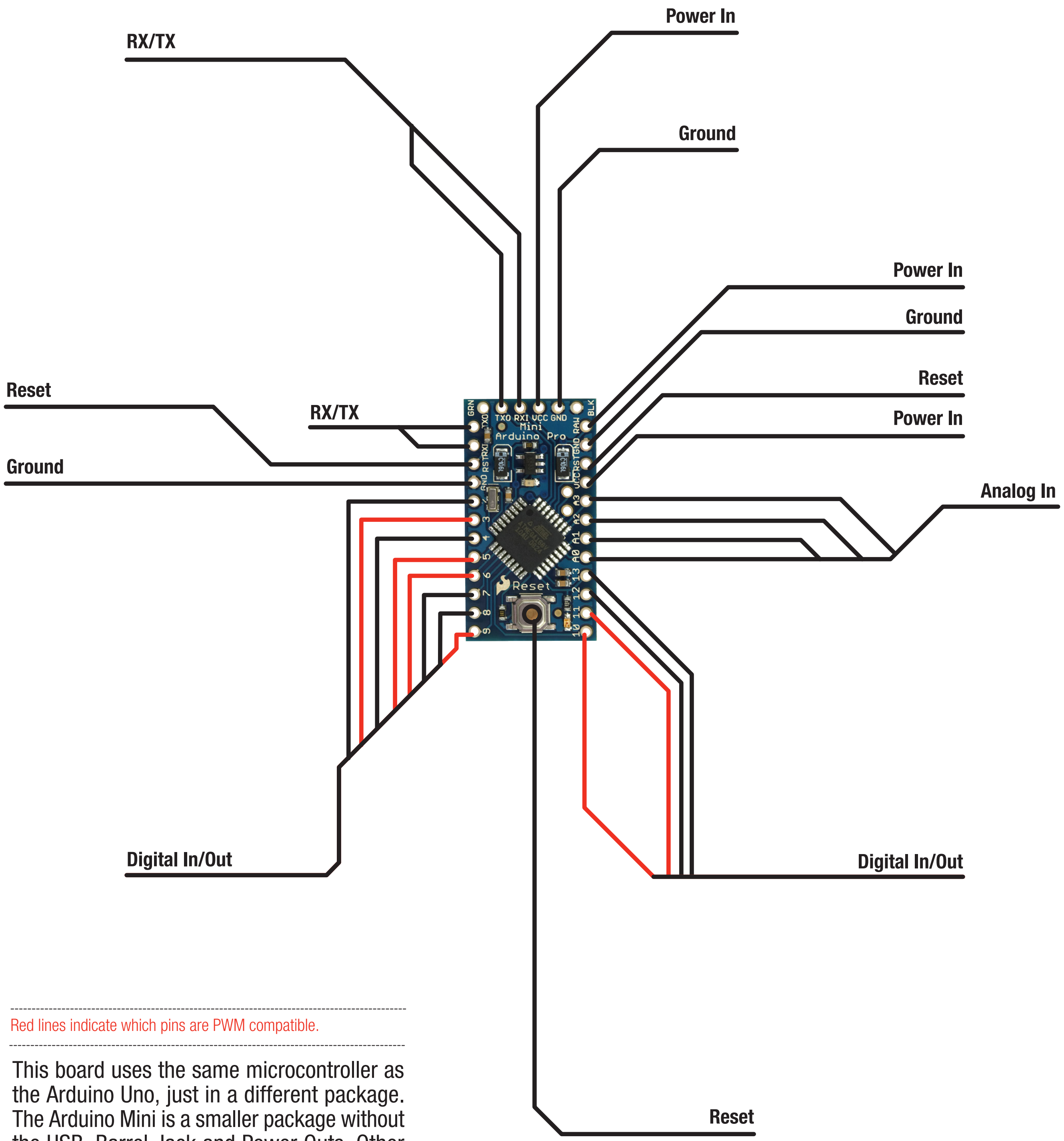
SparkFun RedBoard



Red lines indicate which pins are PWM compatible.

This board uses the same microcontroller as the Arduino Uno, just in a different package. The Lilypad is designed for use with conductive thread instead of wire. Other boards in the Arduino family can be found at http://arduino.cc/en/Main/Hardware

Arduino Lilypad



Red lines indicate which pins are PWM compatible.

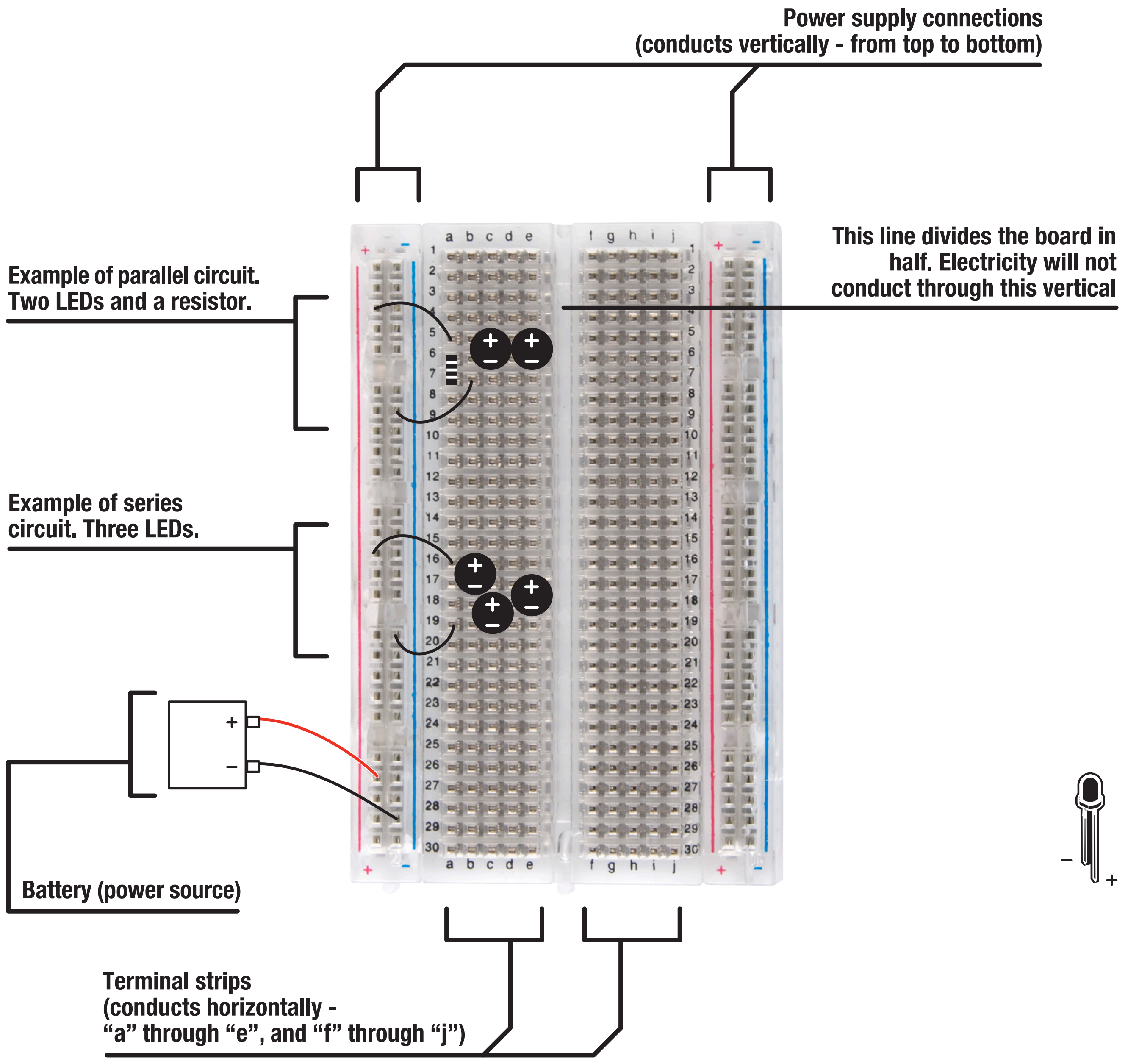
This board uses the same microcontroller as the Arduino Uno, just in a different package. The Arduino Mini is a smaller package without the USB, Barrel Jack and Power Outs. Other boards in the Arduino family can be found at http://arduino.cc/en/Main/Hardware

Arduino Mini

2

Electrical

// How it works


Power supply connections
(conducts vertically - from top to bottom)
This line divides the board in half. Electricity will not conduct through this vertical
Example of parallel circuit. Two LEDs and a resistor.
Example of series circuit. Three LEDs.
Battery (power source)
Terminal strips
(conducts horizontally - "a" through "e", and "f" through "j")
a b c d e
f g h i j
+
–

One of the most important tools for electrical prototyping and invention is the breadboard. It's not a piece of bread that you stick electronics into, it's a piece of plastic with holes to place wires into and copper connecting the holes so electricity can get to all the pieces you are working with. But not all the holes are connected! Above is a diagram and explanation of how a breadboard works as well as examples of parallel and series circuits. Not sure what parallel and series circuits are? Don't worry! The important thing is learning how to use the breadboard so you can play around with some electronics.

The labels on the picture of this breadboard show you which holes are connected and allow electricity to flow between them without anything else connecting them. This is made possible by strips of copper on the underside of the board. The power supply connections have a + and – indicating how to hook up your power source. The connections for the power supply run up and down. The terminal strips are labeled "a" through "j", these connections run across the board, but are broken down the middle. This cuts the connection across the entire terminal area in half, giving you two unconnected sections to work with.

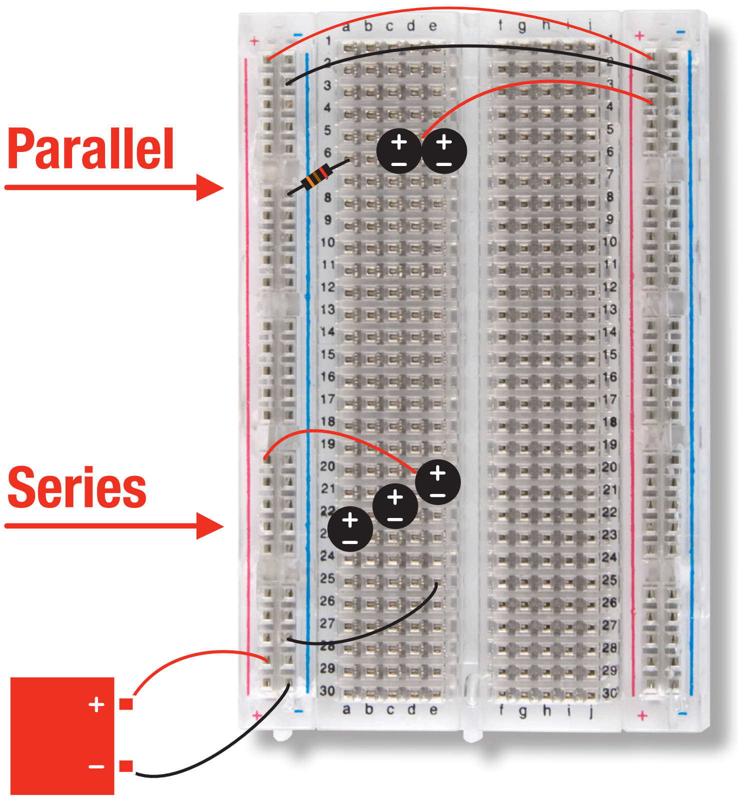
Name:
Date:

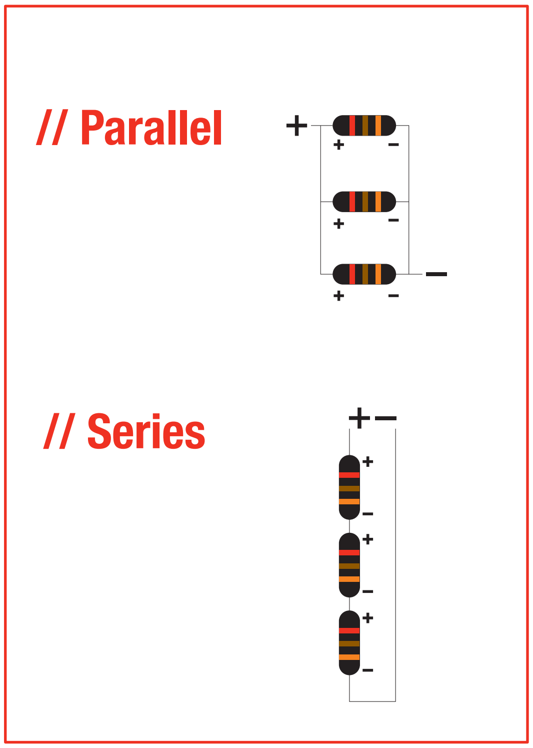
Because the copper plating below the **power** supply connections and the **terminal** connections conduct electricity there are many different ways to hook up the same circuit and make it work. All that matters is that the electricity can flow through the entire circuit from power (+) to ground (-).

This is an example of the same two circuits from the previous page hooked up in different ways that still work the same. There are many differences between this picture and the previous one. First of all, at the very top there is a wire connecting the **positive (+) power** terminal on the left with the **positive (+) power** terminal on the right. Now it is possible to supply power to your circuits from either side of the board. That's why this example of a parallel circuit has a red wire stretching all the way to the **positive (+) power** terminal on the right side. What would you do if you wanted to use the **ground (-) power** terminal on the right side of the board?

Another thing that has changed in the parallel circuit example is the position of the resistor. In the previous image, the resistor went from the row with the negative LED connections to the row below with a wire connecting that final row to the **ground (-) power** terminal. This is an example of tossing out a wire because you can use the wires coming out of components to plug directly into the **power** terminals. You can do the same thing with your LEDs if you like, try it out.

Lastly, the positions of the LEDs have changed in both examples. This is because it doesn't matter where the LEDs are positioned to the left or right (columns "a" through "e"), as long as they are plugged into the correct rows (up and down).





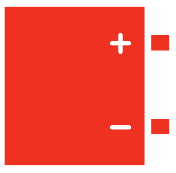
Name:
Date:

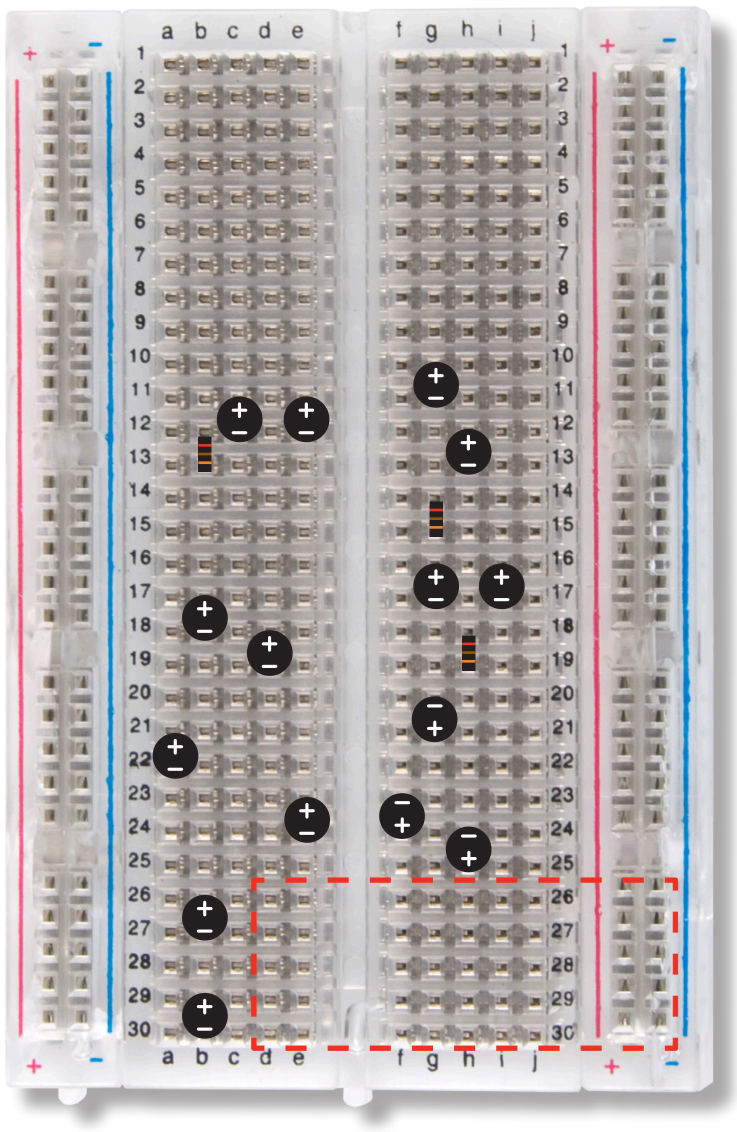
Here are some quick questions to make sure you understand the breadboard:

1. Circle the power terminals below, make sure you get all of them.

2. Draw wires to complete six LED circuits that will work. Each circuit needs either a total of three LEDs or two LEDs and a resistor. Use all power terminals at least once and don't forget to hook up your battery.

3. Inside the dotted lines draw lines to show where electricity will conduct without plugging anything else in.





Name:
Date:

All of the electrical signals that the RedBoard works with are either Analog or Digital. It is extremely important to understand the difference between these two types of signal and how to manipulate the information these signals represent.

// Analog

A continuous stream of information with values between and including 0% and 100%.

Humans perceive the world in analog. Everything we see and hear is a continuous transmission of information to our senses. The temperatures we perceive are never 100% hot or 100% cold, they are constantly changing between our ranges of acceptable temperatures. This continuous stream is what defines analog data. Digital information, the complementary concept to Analog, estimates analog data using only ones and zeros.

In the world of Arduino an analog signal is simply a signal that can be HIGH (on), LOW (off) or anything in between these two states. This means an Analog signal has a voltage value that can be anything between 0V and 5V (unless you mess with the Analog Reference pin). Analog allows you to send output or receive input about devices that run at percentages as well as on and off. The RedBoard does this by sampling the voltage signal sent to these pins and comparing it to a voltage reference signal (5V). Depending on the voltage of the Analog signal when compared to the Analog Reference signal the RedBoard then assigns a numerical value to the signal somewhere between 0 (0%) and 1023 (100%). The digital system of the RedBoard can then use this number in calculations and sketches.

To receive Analog Input the Arduino uses Analog pins # 0 - # 5. These pins are designed for use with components that output Analog information and can be used for Analog Input. There is no setup necessary and to read them use the command:

```
analogRead(pinNumber);
```

where pinNumber is the Analog In pin to which the the Analog component is connected. The analogRead command will return a number including or between 0 and 1023.

The RedBoard also has the capability to output a digital signal that acts as an Analog signal, this signal is called Pulse Width Modulation (PWM). Digital Pins # 3, # 5, # 6, # 9, # 10 and #11 have PWM capabilities. To output a PWM signal use the command:

```
analogWrite(pinNumber, value);
```

where pinNumber is a Digital Pin with PWM capabilities and value is a number between 0 (0%) and 255 (100%). On the Arduino UNO PWM pins are signified by a ~ sign. For more information on PWM see the PWM worksheets or S.I.K. circuit 12.

Examples of Analog:

Values: Temperature, volume level, speed, time, light, tide level, the list goes on....
Sensors: Temperature sensor, Photoresistor, Microphone, Turntable, Speedometer, etc....

Things to Remember about Analog:

Analog Input uses the Analog In pins, Analog Output uses the PWM pins
To receive an Analog signal use:

```
analogRead(pinNumber);
```

To be able to send a PWM signal use:

```
analogWrite(pinNumber, value);
```

Analog Input values range from 0 to 1023 (1024 values because it uses 10 bits, 2^{10})
PWM Output values range from 0 to 255 (256 values because it uses 8 bits, 2^8)

Analog

Name:
Date:

All of the electrical signals that the RedBoard works with are either Analog or Digital. It is extremely important to understand the difference between these two types of signal and how to manipulate the information these signals represent.

// Digital

An electronic signal transmitted as binary code that can be either the presence or absence of current, high and low voltages or short pulses at a particular frequency.

Humans perceive the world in analog, but robots, computers and circuits use Digital. A digital signal is a signal that has only two states. These states can vary depending on the signal, but simply defined the states are ON or OFF, never in between.

Digital signals are used for everything with the exception of Analog Input. Depending on the voltage of the Arduino the ON or HIGH of the Digital signal will be equal to the system voltage, while the OFF or LOW signal will always equal 0V. This is a fancy way of saying that on a 5V RedBoard the HIGH signals will be a little under 5V and on a 3.3V RedBoard the HIGH signals will be a little under 3.3V.

To receive or send Digital signals the Arduino uses Digital pins # 0 - # 13. You may also setup your Analog In pins to act as Digital pins. To set up Analog In pins as Digital pins use the command:

```
pinMode(pinNumber, value);
```

where pinNumber is an Analog pin (A0 – A5) and value is either INPUT or OUTPUT. To setup Digital pins use the same command but reference a Digital pin for pinNumber instead of an Analog In pin. Digital pins default as input, so really you only need to set them to OUTPUT in pinMode. To read these pins use the command:

```
digitalRead(pinNumber);
```

where pinNumber is the Digital pin to which the Digital component is connected. The digitalRead command will return either a HIGH or a LOW signal. To send a Digital signal to a pin use the command:

```
digitalWrite(pinNumber, value);
```

where pinNumber is the number of the pin sending the signal and value is either HIGH or LOW.

The RedBoard also has the capability to output a Digital signal that acts as an Analog signal, this signal is called Pulse Width Modulation (PWM). Digital Pins # 3, # 5, # 6, # 9, # 10 and #11 have PWM capabilities. To output a PWM signal use the command:

```
analogWrite(pinNumber, value);
```

where pinNumber is a Digital Pin with PWM capabilities and value is a number between 0 (0%) and 255 (100%). For more information on PWM see the PWM worksheets or S.I.K. circuit 12.

Examples of Digital:

Values: On/Off, Men's room/Women's room, pregnancy, consciousness, the list goes on....
Sensors/Interfaces: Buttons, Switches, Relays, CDs, etc....

Things to Remember about Digital:

Digital Input/Output uses the Digital pins, but Analog In pins can be used as Digital
To receive a Digital signal use:

```
digitalRead(pinNumber);
```

To be able to send a Digital signal use:

```
digitalWrite(pinNumber, value);
```

Digital Input and Output are always either HIGH or LOW / ON or OFF.

Digital

Name:
Date:

All of the electrical signals that the RedBoard works with are either input or output. It is extremely important to understand the difference between these two types of signal and how to manipulate the information these signals represent.

// Input Signals

A signal entering an electrical system, in this case a micro-controller. Input to the RedBoard pins can come in one of two forms; Analog Input or Digital Input.

Analog Input enters your RedBoard through the Analog In pins # 0 - # 5. These signals originate from analog sensors and interface devices. These analog sensors and devices use voltage levels to communicate their information instead of a simple yes (HIGH) or no (LOW). For this reason you cannot use a digital pin as an input pin for these devices. Analog Input pins are used only for receiving Analog signals. It is only possible to read the Analog Input pins so there is no command necessary in the setup( ) function to prepare these pins for input. To read the Analog Input pins use the command:

```
analogRead(pinNumber);
```

where pinNumber is the Analog Input pin number. This function will return an Analog Input reading between 0 and 1023. A reading of zero corresponds to 0 Volts and a reading of 1023 corresponds to 5 Volts. These voltage values are emitted by the analog sensors and interfaces. If you have an Analog Input that could exceed Vcc + .5V you may change the voltage that 1023 corresponds to by using the Aref pin. This pin sets the maximum voltage parameter your Analog Input pins can read. The Aref pin's preset value is 5V.

Digital Input can enter your RedBoard through any of the Digital Pins # 0 - # 13. Digital Input signals are either HIGH (On, 5V) or LOW (Off, 0V). Because the Digital pins can be used either as input or output you will need to prepare the RedBoard to use these pins as inputs in your setup( )function. To do this type the command:

```
pinMode(pinNumber, INPUT);
```

inside the curly brackets of the setup( ) function where pinNumber is the Digital pin number you wish to declare as an input. You can change the pinMode in the loop( )function if you need to switch a pin back and forth between input and output, but it is usually set in the setup( )function and left untouched in the loop( )function. To read the Digital pins set as inputs use the command:

```
digitalRead(pinNumber);
```

where pinNumber is the Digital Input pin number.

Input can come from many different devices, but each device's signal will be either Analog or Digital, it is up to the user to figure out which kind of input is needed. Hook up the hardware and then type the correct code to properly use these signals.

Things to Remember about Input:

Input is either Analog or Digital, make sure to use the correct pins depending on type.
To take an Input reading use analogRead(pinNumber); (for analog)
Or digitalRead(pinNumber); (for digital)
Digital Input needs a pinMode command such as pinMode(pinNumber, INPUT);
Analog Input varies from 0 to 1023
Digital Input is always either HIGH or LOW

Examples of Input:

Push Buttons, Potentiometers, Photoresistors, Flex Sensors

Input

Name:
Date:

All of the electrical signals that the Arduino works with are either input or output. It is extremely important to understand the difference between these two types of signal and how to manipulate the information these signals represent.

// Output Signals

A signal exiting an electrical system, in this case a micro-controller.

Output to the RedBoard pins is always Digital, however there are two different types of Digital Output; regular Digital Output and Pulse Width Modulation Output (PWM). Output is only possible with Digital pins # 0 - # 13. The Digital pins are preset as Output pins, so unless the pin was used as an Input in the same sketch, there is no reason to use the pinMode command to set the pin as an Output. Should a situation arise where it is necessary to reset a Digital pin to Output from Input use the command:

```
pinMode(pinNumber, OUTPUT);
```

where pinNumber is the Digital pin number set as Output. To send a Digital Output signal use the command:

```
digitalWrite(pinNumber, value);
```

where pinNumber is the Digital pin that is outputting the signal and value is the signal. When outputting a Digital signal value can be either HIGH (On) or LOW (Off).

Digital Pins # 3, # 5, # 6, # 9, # 10 and #11 have PWM capabilities. This means you can Output the Digital equivalent of an Analog signal using these pins. To Output a PWM signal use the command:

```
analogWrite(pinNumber, value);
```

where pinNumber is a Digital Pin with PWM capabilities and value is a number between 0 (0%) and 255 (100%). For more information on PWM see the PWM worksheets or S.I.K. circuit 12.

Output can be sent to many different devices, but it is up to the user to figure out which kind of Output signal is needed, hook up the hardware and then type the correct code to properly use these signals.

Things to remember about Output:

Output is always Digital
There are two kinds of Output: regular Digital or PWM (Pulse Width Modulation)
To be able to send an Output signal use:
analogWrite(pinNumber, value); (for analog) or
digitalWrite(pinNumber, value); (for digital)
Output pin mode is set using the pinMode command:
pinMode(pinNumber, OUTPUT);
Regular Digital Output is always either HIGH or LOW
PWM Output varies from 0 to 255

Examples of Output:

Light Emitting Diodes (LED's), Piezoelectric Speakers, Servo Motors

Output

Purpose: Group activity teaching the concepts of input and output as used in Arduino Programming and Physical Computing. Text formatted like this denotes actual Arduino code.

Materials: Three to five different sized balls and a white/chalk board big enough so the whole room can see it.

Vocabulary to be explained prior to activity:

input: A pin mode that intakes information.
output: A pin mode that sends information.
digitalRead: Command used to get a HIGH or LOW value from a digital input pin.
analogRead: Command used to get a value between or including 0 and 1023 from an analog input pin.
digitalWrite: Command used to send a HIGH or LOW value to an output pin.
analogWrite: Command used to send a PWM value to an output pin simulating an analog output.
PWM: A value between 0 and 255 representing a digital signal simulating an analog output. Used with analogWrite.
HIGH: Electrical signal present (5V for Uno). Also ON or True in boolean logic.
LOW: No electrical signal present (0V). Also OFF or False in boolean logic.

Preparation: Divide the class in quarters, assign each group the following names: Sensors, Input Pins, Output Pins, and Output Components. Arrange the groups in lines in this order about ten to twenty feet apart (or farther if the students are older). The students at the front of each line are Code, their job is to write the Arduino code corresponding to the signal received by their team on the chalk board or white board. Distribute the balls so that each student in the Sensor line has at least one of each size. The smallest balls (tennis or bouncy balls) represent the smallest signal a sensor can send to the RedBoard, LOW. The largest balls represent the largest signal a sensor can send, HIGH. The ball or balls of medium size represent PWM values depending on size.

Activity: To start the activity ask the Code student in the Sensor line to tell the class what kind of sensors the Sensor line represents (photoresistor, potentiometer, flex sensor, etc...) and write on the board what the sensor value is. The sensor value corresponds to the sensor type. For example, if the sensor type is photoresistor then “sunny day” might be written to signal a HIGH signal or “really cloudy” might be written to signal a smaller PWM value.

Each student in the Sensor line then throws the corresponding sized ball to a student in the Input Pin line. Once all the Input Pins have caught their “signals” the Code student in the Input Pin line writes the analogRead or digitalRead value they think corresponds to the signal.

Once the analogRead value has been written on the board the Input Pin line throws the balls to the Output Pin line. The Code student in that line writes the analogWrite or digitalWrite value they think corresponds to the signal on the board and the balls are thrown to the Output Components.

Once all the Output Components catch their balls the Code student tells the class what type of output component the Output Component line represents and the Output Components strike a pose depending on the signal they received. In the example below poses for LEDs and Servos are shown, but students should be encouraged to make up their own output poses or actions. For example, to represent a HIGH value with motor component outputs students might run in place as fast as they can.

Once the Output Component line has finished, the balls are thrown back to the Sensor line, the Code students are replaced by another student in their line and the process starts over. Once every student has had a chance to be the Code student the lines should switch so eventually everyone has a chance to play each part of the input/output process.

This version of the input/output activity is the simplest form of the activity. If students are comfortable with this version and want more of a challenge there are many ways to complicate the activity.

Give the Output line a set of balls as well as the Sensor line and place a piece of code between the Input and Output lines. The code should be a map command switching the Output signal. For example use:

```
map(signal, 0, 1023, 255, 0);
```

so that the Output line must throw a large ball (HIGH signal) when the Input line receives a small ball (LOW signal). You can then switch this code through out the game.

Get rid of the Code students and have the Sensor line choose which ball they will throw. Each student can yell out what their line's value equals depending on the size of the ball they catch. This version is a little more fun but will also be a little more chaotic.

Activity

Once all the Output Components catch their balls the Code student tells the class what type of output component the Output Component line represents and the Output Components strike a pose depending on the signal they received. In the example below poses for LEDs and Servos are shown, but students should be encouraged to make up their own output poses or actions. For example, to represent a HIGH value with motor component outputs students might run in place as fast as they can.

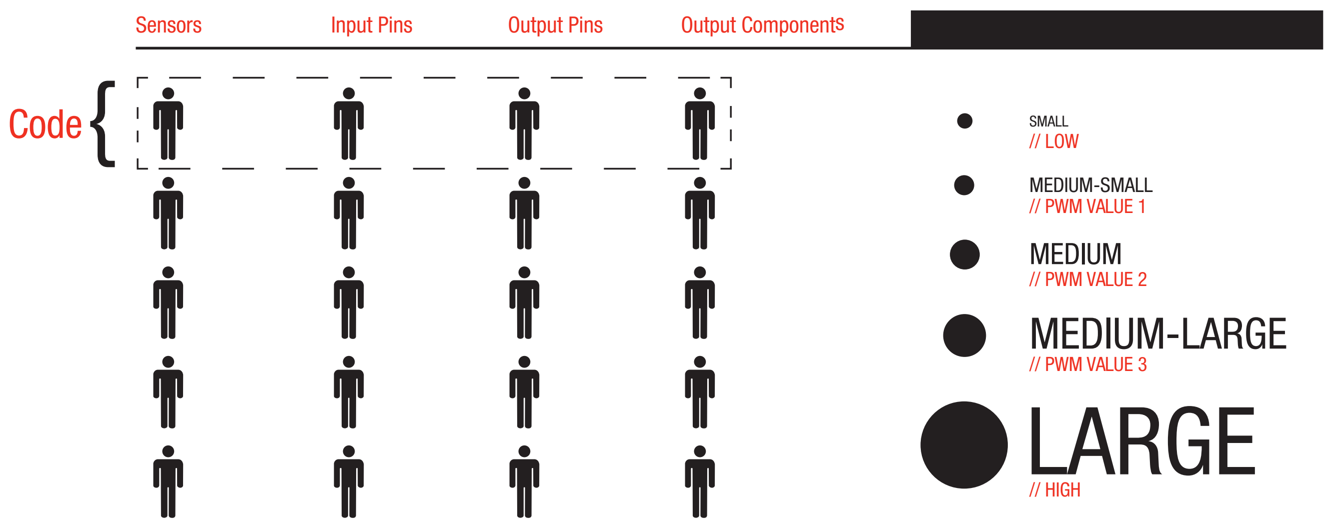
Once the Output Component line has finished, the balls are thrown back to the Sensor line, the Code students are replaced by another student in their line and the process starts over. Once every student has had a chance to be the Code student the lines should switch so eventually everyone has a chance to play each part of the input/output process.

This version of the input/output activity is the simplest form of the activity. If students are comfortable with this version and want more of a challenge there are many ways to complicate the activity.

Give the Output line a set of balls as well as the Sensor line and place a piece of code between the Input and Output lines. The code should be a map command switching the Output signal. For example use:

map(signal, 0, 1023, 255, 0);
so that the Output line must throw a large ball (HIGH signal) when the Input line receives a small ball (LOW signal). You can then switch this code through out the game.

Get rid of the Code students and have the Sensor line choose which ball they will throw. Each student can yell out what their line's value equals depending on the size of the ball they catch. This version is a little more fun but will also be a little more chaotic.

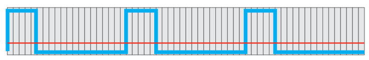


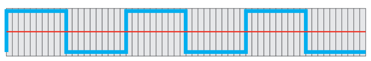
Additional thoughts: This is a great activity just prior to computer lab time. Instead of having kids bouncing off the monitors they will be calmer and ready to sit still applying the concepts they just solidified through physical activity. This is great for kinesthetic learners in particular.

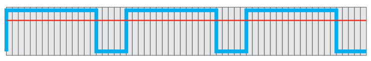
Name:
Date:

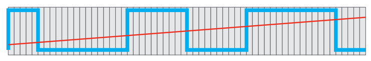
For the most part in computer language one means ON and zero means OFF. This keeps things nice and simple, but what if you want to turn something halfway ON so that it is not all the way ON and not all the way OFF? You can't just use a decimal because digital technology only understands ones and zeros. For this reason some of the pins on your Arduino are labeled PWM or Pulse Width Modulation pins. This means you can send a bunch of ones and zeros real quick and the Arduino board will read these ones and zeros as an average somewhere between one and zero. The red line in the diagrams represent the average. See tables to the right.

Luckily a lot of the work has been done for you so you don't have to figure out the actual patterns of ones and zeros. All you have to do is pick a number between 0 and 255 and type the command analogWrite. The number zero means the pin is set fully off, the number 255 means the pin is set fully on, and all other numbers set the pin to values between ON (100% or 255) and OFF (0% or 0). You can use PWM on any pin labeled PWM and do not need to set the pin mode before sending an analogWrite command.


PWM signal at 25%


PWM signal at 50%


PWM signal at 75%


PWM signal rising from 25% - 75%

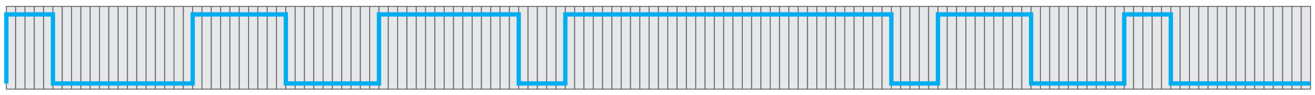
How do you think a PWM signal will affect each of these components compared to a 1 or a 0?

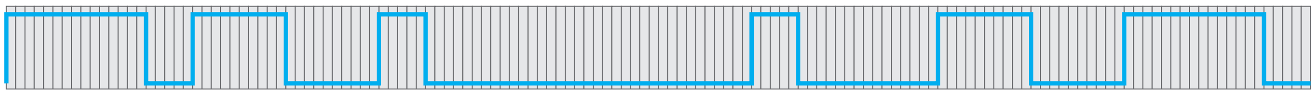
LED: ______________________

Motor: ______________________

Piezo: ______________________

Draw a line through these two charts to show where you believe the PWM value should be.





There are many concepts outside of electrical engineering that are similar to Pulse Width Modulation. Can you list at least three and explain what is being modulated?

____

____

____

____

____

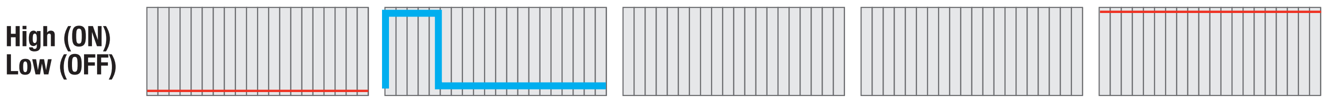
Name:
Date:

Computers and microprocessors only understand two things, ON and OFF. These are represented in a few different ways. There is ON and OFF, One and Zero, or HIGH and LOW. Ones and Zeros are used in the computer language Binary, HIGH and LOW are used with electricity, ON and OFF are plain old human speak.

But what if we want to turn something digital less than 100% ON? Then we use something called PWM, or Pulse Width Modulation. The way your Arduino microprocessor does this is by turning the electricity on a PWM pin ON and then OFF very quickly. The longer the electricity is ON the closer the PWM value is to 100%. This is very useful for controlling a bunch of stuff. For example: the brightness of a light bulb, volume of sound, or the speed of a motor. **These are very basic examples, what else might you need to control that is not only ON or OFF? Explain at least two examples.**

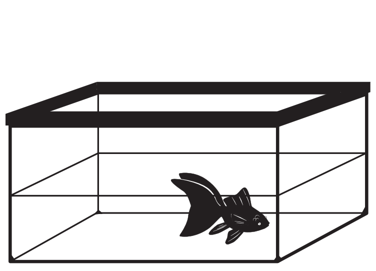
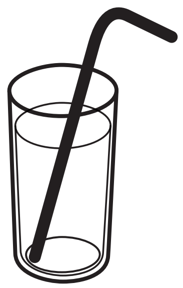
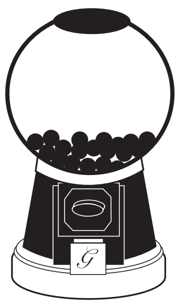
A microprocessor creates a PWM signal by using a built in clock. The microprocessor measures a certain amount of time (also called a window or a period) and turns the PWM pin ON (or HIGH) for the first part of this window and then OFF (or LOW) near the end of the window. The window is filled up with a different length ON (or HIGH) signal depending on the PWM value. If the PWM value is 50% then the PWM signal is ON (or HIGH) for half of the window. If the PWM value is 25% then the PWM signal is ON (or HIGH) for a quarter of the window. The only time the window will not have a LOW value is if the PWM signal is turned completely ON the whole time and therefore equal to 100% ON. The opposite is true as well, if the PWM signal is set to 0% or OFF, then there will not be any HIGH value at the beginning of the window. **Explain in your own words what a PWM window is.**

Below are five different PWM windows. A PWM signal is simply a bunch of PWM windows one after another. Some are missing labels and some are missing diagrams. Please fill in the blanks on the middle three.



PWM percent 0% (No wave) | PWM percentage ______ | PWM percent 50% (Draw in window) | PWM percent 75% (Draw in window) | PWM percent 100% (No wave)

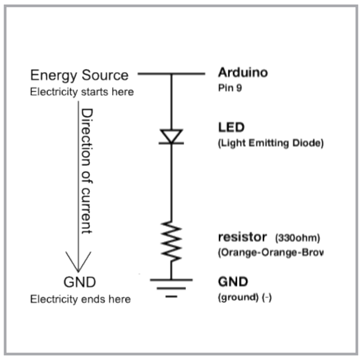
Below are three different metaphors for a PWM window and a PWM signal. Write the physical item that represents the window and the item or items that represents the signal. Then estimate the PWM percent.

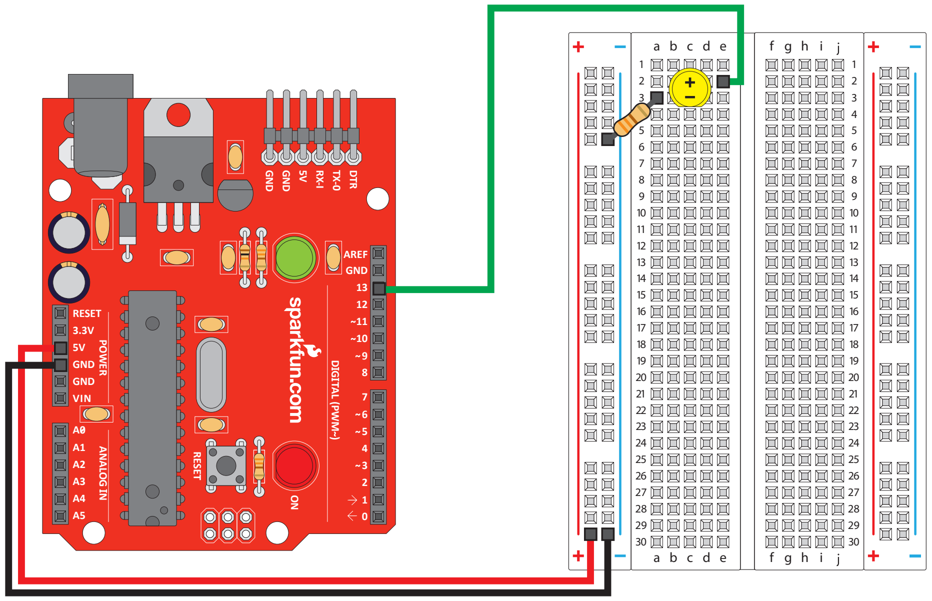




Window:	Window:	Window:
Signal percentage:	Signal percentage:	Signal percentage:

Circuit #1: Blinking an LED

Name:
Date:





Explanation:

This circuit takes electricity from digital Pin # 9 on the RedBoard. Pin # 9 on the RedBoard has Pulse Width Modulation capability allowing the user to change the brightness of the LED when using analogWrite. The LED is connected to the circuit so electricity enters through the anode (+, or longer wire) and exits through the cathode (-, or shorter wire). The resistor dissipates current so the LED does not draw current above the maximum rating and burn out. Finally the electricity reaches ground, closing the circuit and allowing electricity to flow from power source to ground.

Components:

Arduino Digital Pin # 9: Power source, PWM (if code uses analogWrite) or digital (if code uses digitalWrite) output from Arduino board.

LED: As in other diodes, current flows easily from the + side, or anode (longer wire), to the - side, or cathode (shorter wire), but not in the reverse direction. Also lights up!

330 Ohm Resistor: A resistor resists the current flowing through the circuit. In this circuit the resistor reduces the current so the LED does not burn out.

Gnd: Ground

Code:

```
int ledPin = 13;

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

 void loop() {
   digitalWrite(ledPin, HIGH); //LED on
   delay(1000); // wait second
   digitalWrite(ledPin, LOW); //LED off
   delay(1000); // wait second
}
```

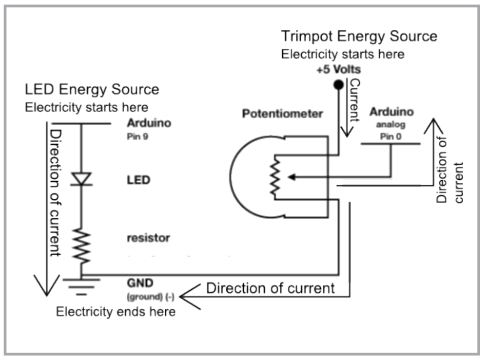
or for PWM the output loop could read :

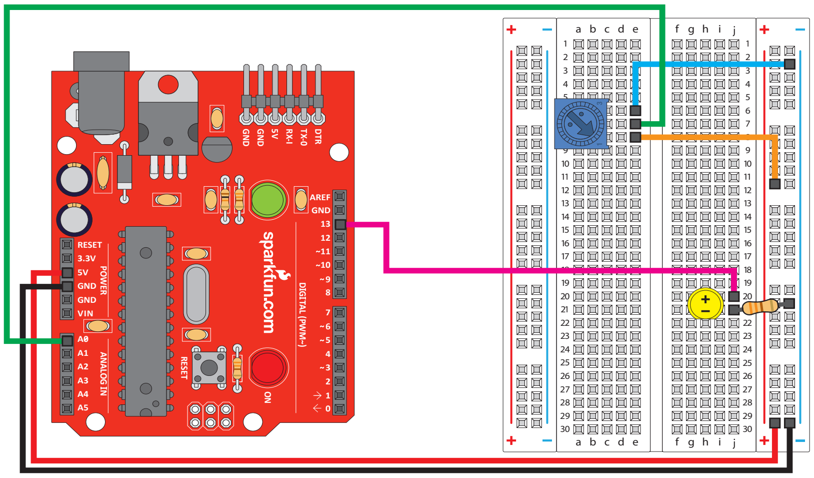
```
int ledPin = 11;

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

 void loop() {
   analogWrite(ledPin, 255); // LED on
   delay(1000); // wait second
   analogWrite(ledPin, 0); // LED off
   delay(1000); // wait second
}
```

CHAPTER 2
How the Circuits Work

Name:
Date:

Circuit #2: Potentiometer





Explanation:

This circuit is actually two different circuits. One circuit for the potentiometer and another for the LED. See 'How the Circuits Work' Circuit 1 for an explanation of the LED circuit. The potentiometer circuit gets electricity from the 5V on the Arduino. The electricity passes through the potentiometer and sends a signal to Analog Pin # 0 on the Arduino. The value of this signal changes depending on the setting of the dial on the potentiometer. This analog reading is then used in the code you load onto the Arduino and effects the power signal in the LED circuit. Finally the electricity reaches ground, closing the circuit and allowing electricity to flow from power source to ground.

Components:

Arduino Digital Pin # 13: Power source, PWM (if code uses analogWrite) or digital (if code uses digitalWrite) output from Arduino board.

Arduino Analog Pin # 0: Analog input to Arduino board.

330 Ohm Resistor: A resistor resists the current flowing through the circuit. In the LED circuit it reduces the current so the LED in the circuit does not burn out.

LED: As in other diodes, current flows easily from the + side, or anode (longer wire), to the - side, or cathode (shorter wire), but not in the reverse direction.

Potentiometer: A voltage divider which outputs an analog value.

+5V: Five Volt power source.

Gnd: Ground

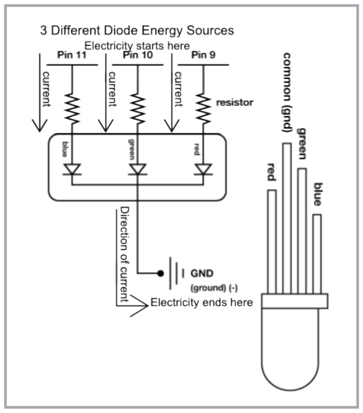
Code:

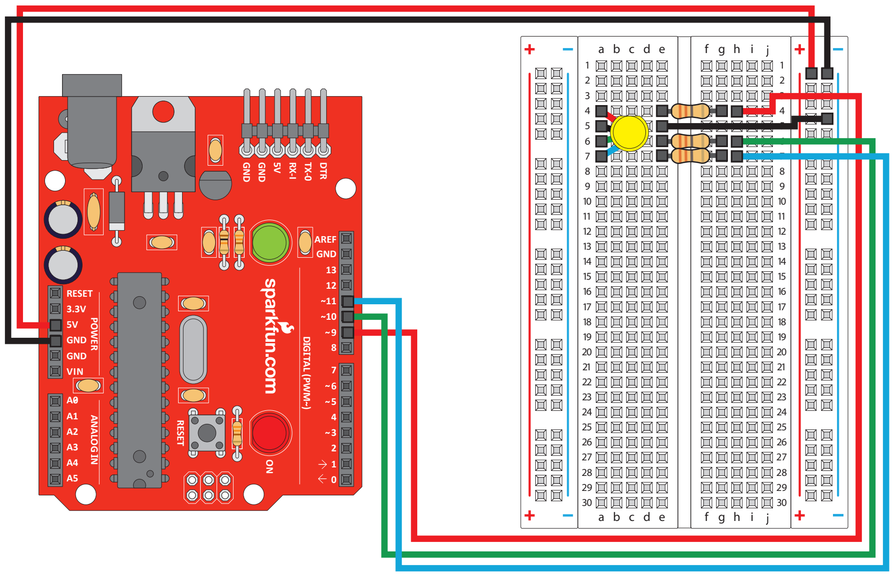
```
int sensorPin = 0;
int ledPin = 13;
int sensorValue = 0;

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

void loop() {
//this line assigns whatever the analog Pin 0 reads to sensorValue

 sensorValue = analogRead(sensorPin);

 digitalWrite(ledPin, HIGH);
 delay(sensorValue);
 digitalWrite(ledPin, LOW);
 delay(sensorValue);
}
```

Additional thoughts: This is another example of input, only this time it is Analog. Circuits 2 and 5 in the S.I.K. introduces you to the two kinds of input your board can receive: Digital and Analog. Not sure what a voltage divider is? Check out the Voltage Divider page towards the back of this section.

Name:
Date:

Circuit #3: RGB LEDs





Explanation:

This circuit is pretty straight forward. The Digital Arduino Pins # 9, # 10 and # 11 supply a PWM value to each of the three different LEDs within the Tri-Color LED (Red, Green, and Blue). The LEDs are connected to the circuit so electricity enters through the anode (+, or longer wire) and exits through the cathode (-, or shorter wire). The resistors dissipate current so the LEDs do not draw current above the maximum rating and burn out. Finally the electricity reaches ground, closing the circuit and allowing electricity to flow from power source to ground. By supplying different values to just these three Digital Pins you can mix 16,777,216 different colors!

Components:

Arduino Digital Pin # 9, # 10 and # 11: Power source, PWM output from Arduino board.

RGB LED: Unlike single color LEDs, on RGB (Also called ‘Tri-Color’) LEDs, the cathode (or ground wire) is the longest wire and each color (Red, Green, and Blue) gets its own lead. (See the schematic for details).

330 Ohm Resistor: A resistor resists the current flowing through the circuit. In this circuit the resistor reduces the current so the LEDs do not burn out.

Gnd: Ground

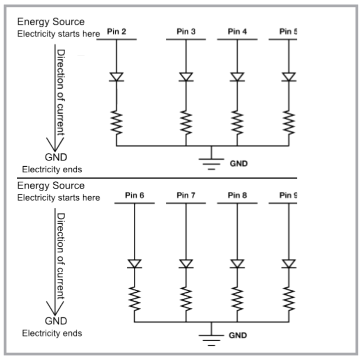
Code:

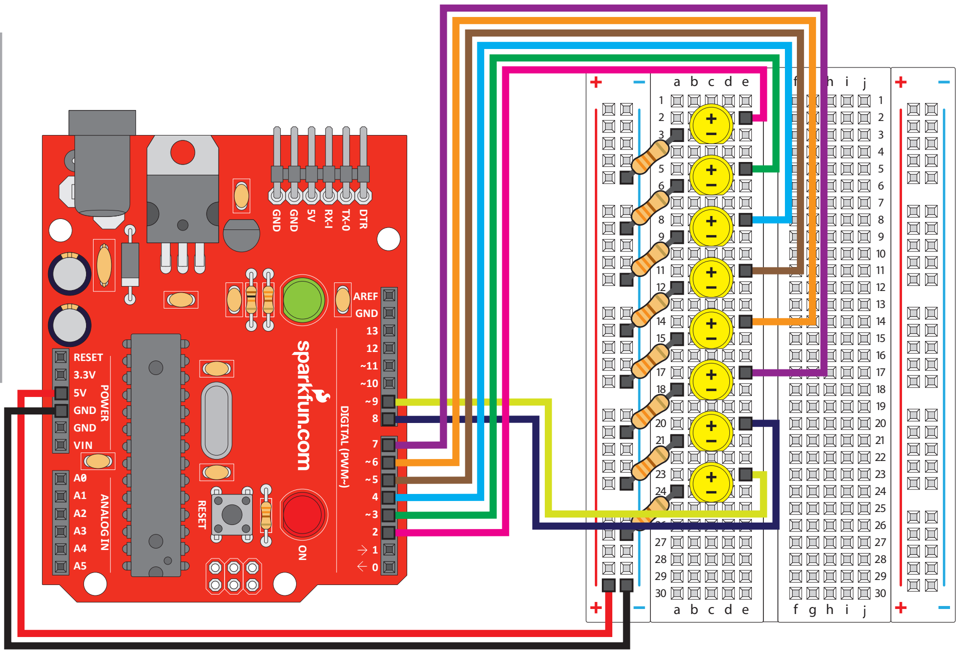
```
const int RED_LED_PIN = 9;
const int GREEN_LED_PIN = 10;
const int BLUE_LED_PIN = 11;
int redIntensity = 0;
int greenIntensity = 0;
int blueIntensity = 0;
const int DISPLAY_TIME = 100;

void setup() {
// No setup required but you still need it
}

void loop(){
  for (greenIntensity = 0; greenIntensity <= 255;
greenIntensity+=5) {
     redIntensity = 255-greenIntensity;
     analogWrite(GREEN_LED_PIN, greenIntensity);
     analogWrite(RED_LED_PIN, redIntensity);
     delay(DISPLAY_TIME);
  }
  for (blueIntensity = 0; blueIntensity <= 255;
blueIntensity+=5) {
     greenIntensity = 255-blueIntensity;
     analogWrite(BLUE_LED_PIN, blueIntensity);
     analogWrite(GREEN_LED_PIN, greenIntensity);
     delay(DISPLAY_TIME);
  }
  for (redIntensity = 0; redIntensity <= 255; redIntensity+=5)
{
     blueIntensity = 255-redIntensity;
     analogWrite(RED_LED_PIN, redIntensity);
     analogWrite(BLUE_LED_PIN, blueIntensity);
     delay(DISPLAY_TIME);
  }
}
```

Additional thoughts: See how combining just three simple outputs can create some amazing results?

Name:
Date:

Circuit #4: Multiple LEDs





Explanation:

This circuit takes electricity from Pin # 2 through Pin # 9 on the Arduino. The LEDs are connected to the circuit so electricity enters through the anode (+, or longer wire) and exits through the cathode (-, or shorter wire). The resistor dissipates current so the LEDs do not draw current above the maximum rating and burn out. Finally the electricity reaches ground, closing the circuit and allowing electricity to flow from power source to ground.

Components:

Arduino Digital Pins # 2 - # 9: Power source, analog (if code uses analogWrite, only possible on pins 3, 5, 6, & 9) or digital (if code uses digitalWrite) output from Arduino board.

LEDs: As in other diodes, current flows easily from the + side, or anode (longer wire), to the - side, or cathode (shorter wire), but not in the reverse direction. Also lights up!

330 Ohm Resistor: The resistors resist the current flowing through the circuit. In this circuit the resistors reduce the current so the LEDs do not burn out.

Gnd: Ground

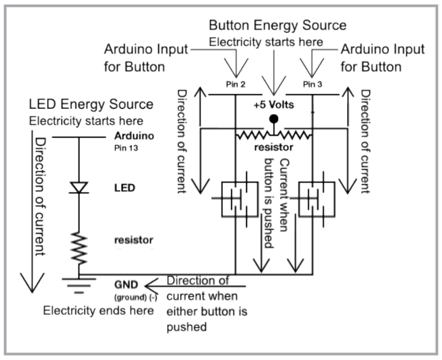
Code:

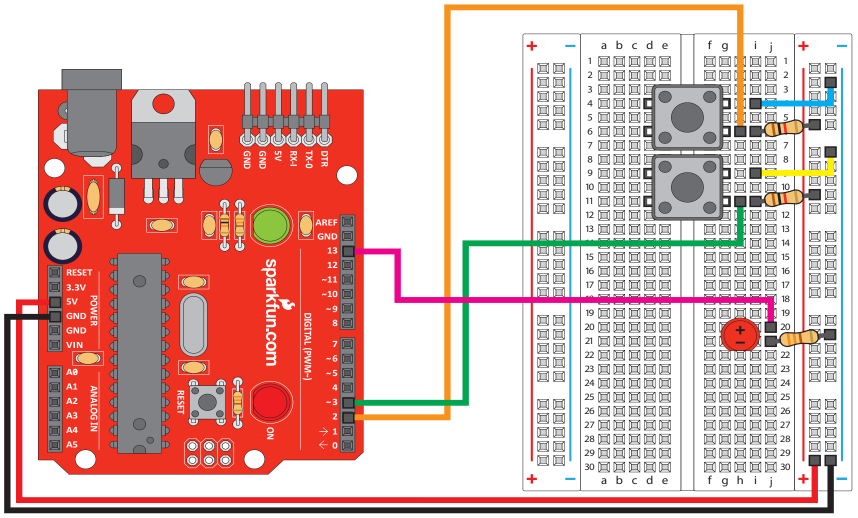
```
//this line below declares an array
int ledPins[ ] = {2,3,4,5,6,7,8,9};

void setup( ) {

//these two lines set Digital Pins # 0 – 8 to output
for(int i = 0; i < 8; i++){
pinMode(ledPins[i],OUTPUT);

}
```

```
void loop( ) {

//these lines turn the LEDs on and then off
for(int i = 0; i <= 7; i++){
  digitalWrite(ledPins[i], HIGH);
  delay(delayTime);
  digitalWrite(ledPins[i], LOW);
  }
}
```

Additional thoughts: The code examples in the S.I.K get a little complicated for the fourth circuit, but don't worry, it's just more outputs. Some of the code examples use "for" loops to do something a number of times, if you're not familiar with "for" see Loops in Programming Concepts.

Circuit #5: Push Buttons





Explanation:

This circuit is actually two different circuits. One circuit for the buttons and another for the LED. See 'How the Circuits Work' Circuit 1 for an explanation of the LED circuit. The button circuit gets electricity from the 5V on the Arduino. The electricity passes through a pull up resistor, causing the input on Arduino Pins # 2 and # 3 to read HIGH when the buttons are not being pushed. When a button is pushed it allows the current to flow to ground, causing a LOW reading on the input pin connected to it. This LOW reading is then used in the code you load onto the Arduino and effects the power signal in the LED circuit.

Components:

Arduino Digital Pin # 13: Power source, PWM (if code uses analogWrite) or digital (if code uses digitalWrite) output from Arduino board.

Arduino Digital Pin # 2 and # 3: Digital input to Arduino board.

330 & 10K Ohm Resistors: Resistors resist the current flowing through the circuit. In the LED circuit the 330 ohm resistor reduces the current so the LED in the circuit does not burn out. In the button circuits the 10Ks ensure that the buttons will read HIGH when they are not pressed.

LED: As in other diodes, current flows easily from the + side, or anode (longer wire), to the - side, or cathode (shorter wire), but not in the reverse direction. Lights up!

Button: A press button which is open (or disconnected) when not in use and closed (or connected) when pressed. This allows you to complete a circuit when you press a button.

+5V: Five volt power source.

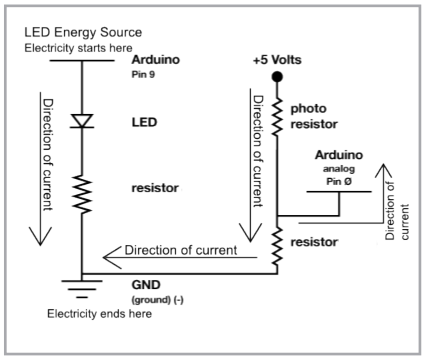
Gnd: Ground

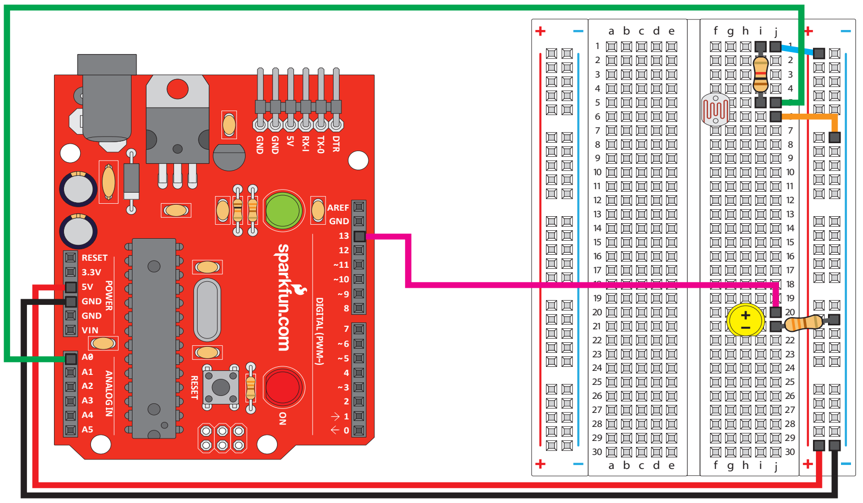
Code:

```
const int buttonPin = 2;
const int ledPin =  13;

int buttonState = 0;

void setup() {
  pinMode(ledPin, OUTPUT);
  //this line below declares the button pin as input
  pinMode(buttonPin, INPUT);
}

void loop(){
  //this line assigns whatever the Digital Pin 2 reads to
  //buttonState
  buttonState = digitalRead(buttonPin);

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

Additional thoughts: This circuit is the first circuit to use the input capabilities of the Arduino. Notice the difference in setup( ). You are still using a Digital Pin but you are using it as input rather than output. Buttons are sweet by the way, let the kids press these buttons instead of yours.

Circuit #6: Photo Resistor





Explanation:

This circuit is actually two different circuits. One circuit for the photoresistor and another for the LED. See ‘How the Circuits Work’ Circuit 1 for an explanation of the LED circuit. The photoresistor circuit gets electricity from the 5V on the Arduino. The electricity passes through the photoresistor and sends a signal to Analog Pin # 0 on the Arduino. The value of this signal changes depending on the amount of sunlight. This analog reading is then used in the code you load onto the Arduino and effects the power signal in the LED circuit. The resistor below the Analog Pin connection creates the voltage divider necessary to measure the resistance of the photoresistor. Finally the electricity reaches ground, closing the circuit and allowing electricity to flow from power source to ground.

Components:

Arduino Digital Pin # 13: Power source, PWM (if code uses analogWrite) or digital (if code uses digitalWrite) output from Arduino board.

Arduino Analog Pin # 0: Analog input to Arduino board.

330 Ohm Resistor: A resistor resists the current flowing through the circuit. In the LED circuit it reduces the current so the LED in the circuit does not burn out. In the photoresistor circuit the resistor completes the voltage divider.

LED: As in other diodes, current flows easily from the + side, or anode (longer wire), to the - side, or cathode (shorter wire), but not in the reverse direction. Lights up!

Photoresistor: A resistor with a resistance value that changes depending on the amount of light hitting the sensor.

+5V: Five Volt power source.

Gnd: Ground

Code:

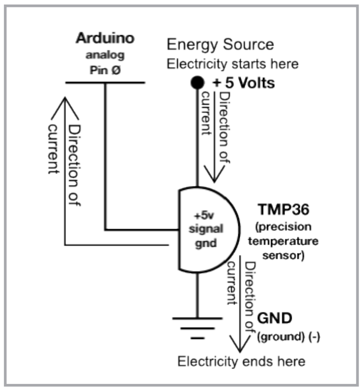
```
int lightPin = 0;
int ledPin = 13;

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

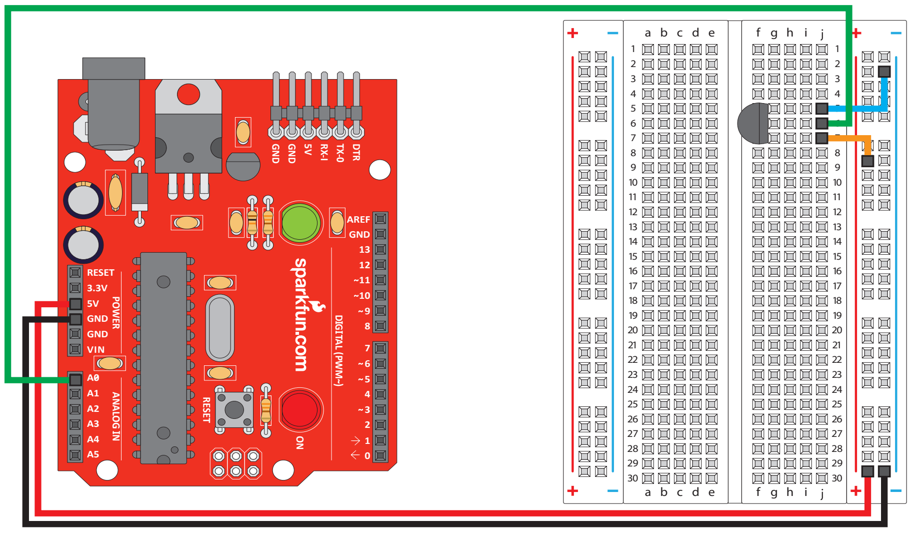
```
void loop() {
  int lightLevel = analogRead(lightPin);
  lightLevel = map(lightLevel, 0, 900, 0, 255);
  lightLevel = constrain(lightLevel, 0, 255);
  analogWrite(ledPin, lightLevel);
}
```

Additional thoughts: This circuit is another example of Analog input. It is also a perfect example of a voltage divider. Don’t worry about the “map” and “constrain” functions they are explained in the glossary. Unsure about the voltage divider? See the voltage divider page towards the back of this section.

Name:
Date:

Circuit #7: Temperature Sensor





Explanation:

This circuit takes electricity from the 5V on the Arduino. The temperature sensor sends an analog value to Arduino Analog Pin # 0. Then the electricity reaches ground, closing the circuit and allowing electricity to flow from power source through the sensor to ground. Finally Arduino uses its Serial monitor to display the temperature reading.

Components:

Arduino Analog Pin # 0: Analog input to Arduino board.

Temperature Sensor: Provides a voltage value depending on the temperature. Some math is then required to convert this value to Celsius or Fahrenheit.

+5V: Five Volt power source.

Gnd: Ground

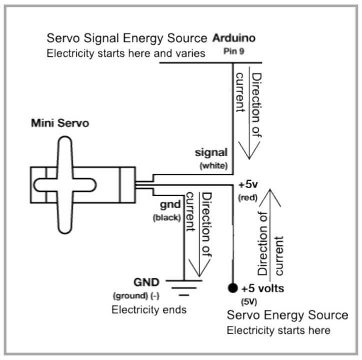
Code:

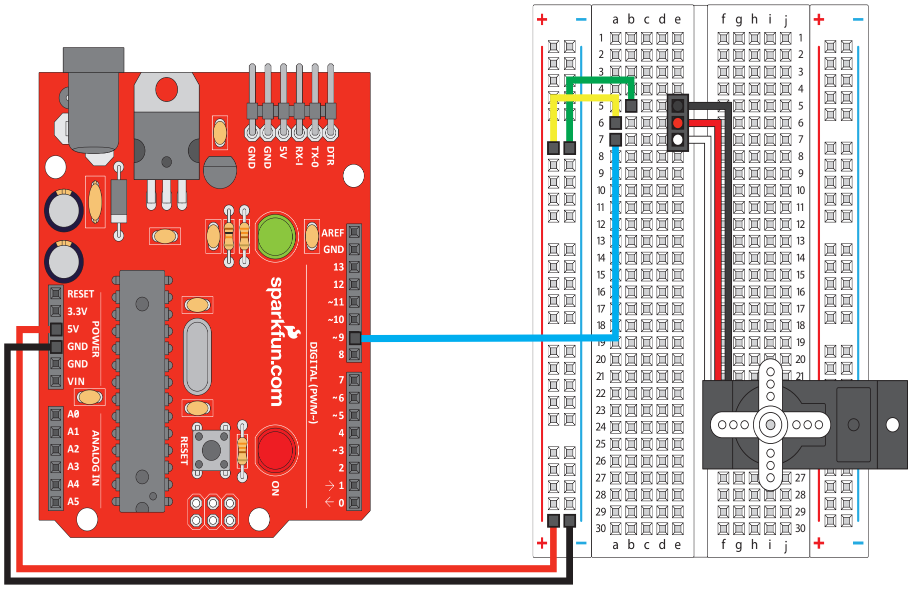
```
int temperaturePin = 0;

void setup() {

//Serial comm. at a Baud Rate of 9600
  Serial.begin(9600);
}

void loop() {

//Calls the function to read the sensor pin
  float temp = getVoltage(temperaturePin);

//Below is a line that compensates for an offset
//(see datasheet)
  temp = (temp - .5) * 100;

   //This line displays the variable temperature after all
  //the math
  Serial.println(temp);
  delay(1000);
}

//function that reads the Arduino pin and starts to convert
//it to degrees

float getVoltage(int pin) {
  return (analogRead(pin) * .004882814);
}
```

Additional thoughts: There is a lot of math involved in the code section of this circuit and it all has a reason. But how would you know you need to offset the temperature reading by .5 unless you had read the Datasheet? Also, pay attention to the code lines that enable Serial communication.

Name:
Date:

Circuit #8: A Single Servo





Explanation:

The servo in this circuit takes electricity from 5V on the Arduino. Pin # 9 on the Arduino supplies a PWM signal which sets the position of the servo. Each voltage value has a distinct correlating position. Finally the electricity reaches ground, closing the circuit and allowing electricity to flow from power source to ground.

Components:

Arduino Digital Pin #9: Signal power source for servo.

Servo: Sets the position of the servo arm depending on the voltage of the signal received.

+5V: Five volt power source.

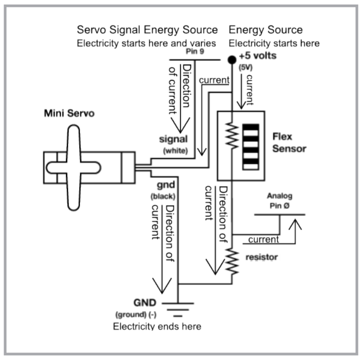
Gnd: Ground

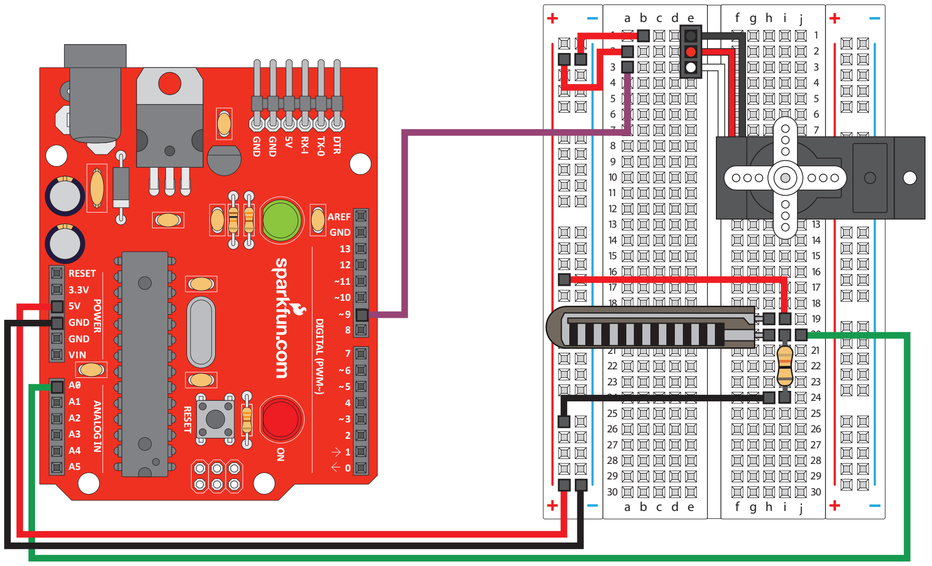
Code:

```
//include the servo library for use
#include <Servo.h>
Servo myservo;  //create servo object

int pos = 0;

void setup() {
  myservo.attach(9);
}
void loop() {
//moves servo from 0° to 180°
  for(pos = 0; pos < 180; pos += 1) {
    myservo.write(pos);
    delay(15);
    }
  // moves servo from 180° to 0°
  for(pos = 180; pos>=1; pos-=1)  {
    myservo.write(pos);
    delay(15);
    }
}
```

Additional thoughts: To some kids this is exciting stuff. There are all kinds of things kids can think to do with servos, you've just got to ask them. Throw out the word "robot" and see what comes back at you. Remember, this is just slightly more complicated output, same as the motor and LED.

Circuit #9: Flex Sensor





Explanation:

This circuit is actually two different circuits. One circuit for the flex sensor and another for the servo. See 'How the Circuits Work' Circuit 8 for an explanation of the servo circuit. The flex sensor circuit gets electricity from the 5V on the Arduino. The electricity passes through the flex sensor and sends a signal to Analog Pin # 0 on the Arduino. The value of this signal changes depending on the amount of bend in the flex sensor. This analog reading is then used in the code you load onto the Arduino and sets the position of the servo. The resistor and flex sensor create a voltage divider which is measured by Analog Pin # 0. Finally the electricity reaches ground, closing the circuit and allowing electricity to flow from power source to ground.

Components:

Arduino Digital Pin # 9: Power source, PWM output from Arduino board.

Arduino Analog Pin # 0: Analog input to Arduino board.

10K Ohm Resistor: A resistor resists the current flowing through the circuit.

Flex Sensor: A resistor with a value that varies depending on the amount of bend in the sensor.

Servo: Sets the position of the servo arm depending on the voltage of the signal received.

+5V: Five Volt power source.

Gnd: Ground

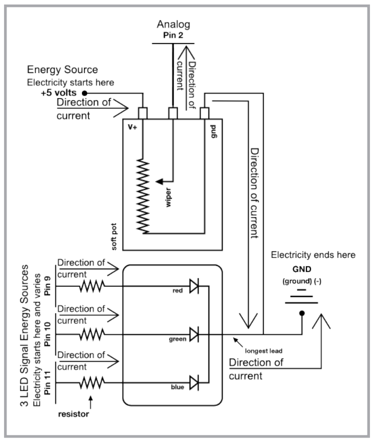
Code:

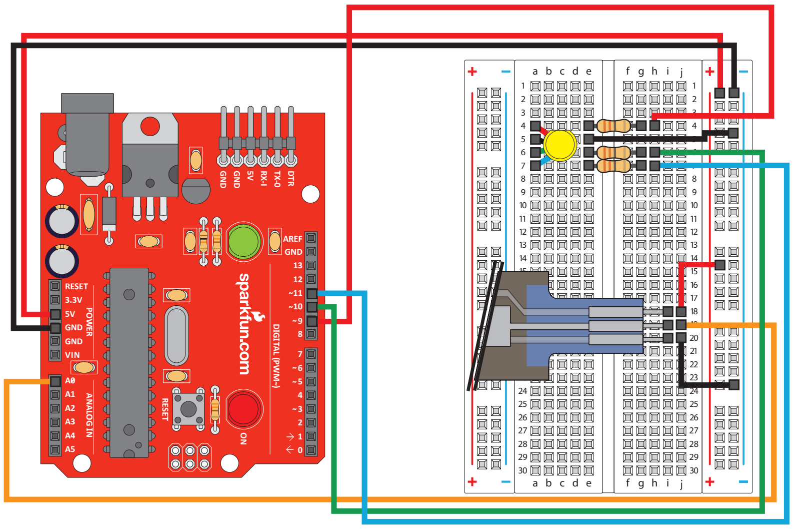
```
#include <Servo.h> //include the servo library
Servo myservo;

int potpin = 0; //sets pin 0 to read the flex sensor
int val;

void setup() {
  Serial.begin(9600);
  myservo.attach(9);
}

void loop() {
  val = analogRead(potpin); //get a reading from the flex sensor
  Serial.println(val);
  val = map(val, 50, 300, 0, 179);
  myservo.write(val);
  delay(15);
}
```

Additional thoughts: This analog input is definitely very different from any other input we have looked at so far but the concept is the same. We treat the sensor as a resistor in a voltage divider to get a reading and then change our output depending on that reading.

Name:
Date:

Circuit #10: Soft Potentiometer





Explanation:

This circuit is actually two different circuits. One circuit for the soft pot and another for the RGB LED. See 'How the Circuits Work' Circuit 3 for an explanation of the RGB LED circuit. The soft pot circuit gets electricity from the 5V on the Arduino. The electricity passes through the soft pot and sends a signal out the com line of the soft pot to Analog Pin # 0 on the Arduino. The value of this signal changes depending on where the wiper (any type of contact) touches the soft pot. This analog reading is then used in the code you load onto the Arduino and sets the color of the RGB LED. Notice that yet again our sensor and the input pin form a voltage divider, only this time the voltage divider is completely inside the sensor. The wiper divides the resistor into two different portions with values that depend on the position of the wiper. Finally the electricity reaches ground, closing the circuit and allowing electricity to flow from power source to ground.

Components:

Arduino Digital Pins # 9, 10, 11: Power source, PWM output from Arduino board.

Arduino Analog Pin # 0: Analog input to Arduino board.

330 Ohm Resistor: A resistor resists the current flowing through the circuit. In the RGB LED circuit it reduces the current so the LED it is attached to does not burn out.

Flex Sensor: A resistor with a value that varies depending on the amount of bend in the sensor.

RGB LED: A grouping of three LEDs, Red, Green and Blue. Power goes in three different anodes (+, the short wires) and out one common cathode (-, the long wire). Lights up!

+5V: Five Volt power source.

Gnd: Ground

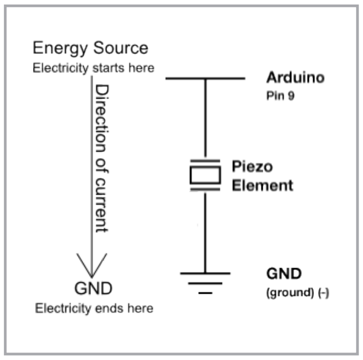
Code:

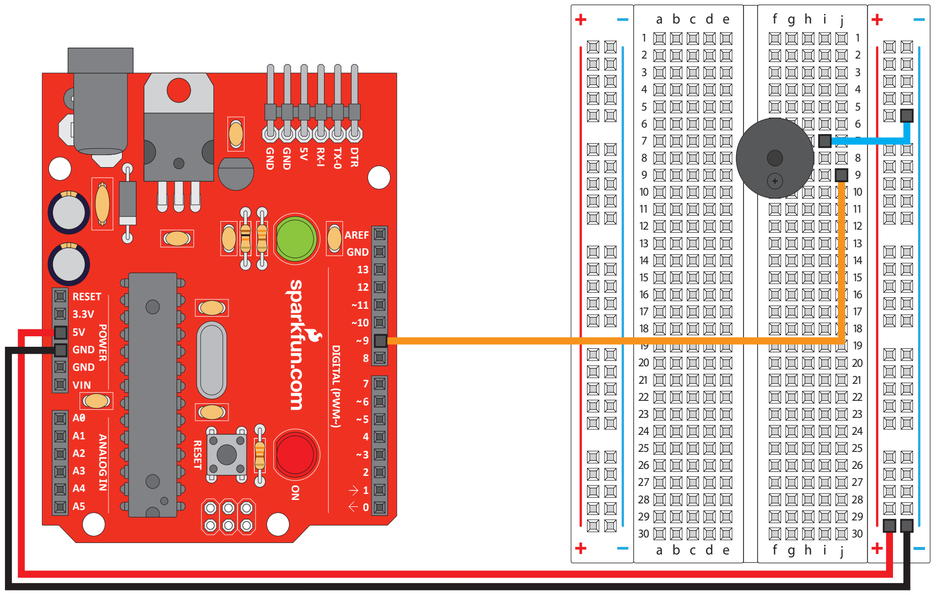
```
const int RED_LED_PIN = 9;
const int GREEN_LED_PIN = 10;
const int BLUE_LED_PIN = 11;

void setup() {
//No setup necessary but you still need it
}

void loop() {
  int sensorValue = analogRead(0);
  int redValue = constrain(map(sensorValue, 0, 512, 255, 
0),0,255);
  int greenValue = constrain(map(sensorValue, 0, 512, 0, 
255),0,255)-constrain(map(sensorValue, 512, 1023, 0, 
255),0,255);
  int blueValue = constrain(map(sensorValue, 512, 1023, 
0, 255),0,255);

  analogWrite(RED_LED_PIN, redValue);
  analogWrite(GREEN_LED_PIN, greenValue);
  analogWrite(BLUE_LED_PIN, blueValue);
}
```

Additional thoughts: This analog sensor is similar to the flex sensor. You will often see the basic concepts covered in the S.I.K. in different forms as you work with more complicated sensors and outputs. Most of these technologies are built using the same building blocks and some math.

Circuit #11: Piezo Elements

Name:
Date:





Explanation:

This circuit gets electricity from Arduino Pin # 9. The Piezo element plays different musical notes depending on the speed and duration of the electrical signal sent from Pin # 9. Finally the electricity reaches ground, closing the circuit and allowing electricity to flow from power source to ground.

Components:

Arduino Digital Pin # 9: Power source, digital output from Arduino board. (If changed to PWM output this creates distortion of note, not a change in volume.)

Piezo element: A tiny speaker with a magnetic coil that responds to electrical current by moving more or less depending on the current. The coil is attached to a diaphragm that moves air and causes the noise we hear.

Gnd: Ground

Note:

This section contains only the two functions needed to make the piezo play a note of a given duration. These functions are called in the loop ( ) function.

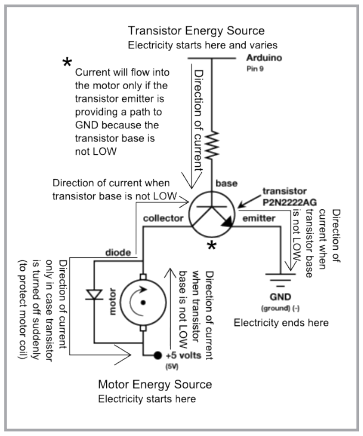
Code:

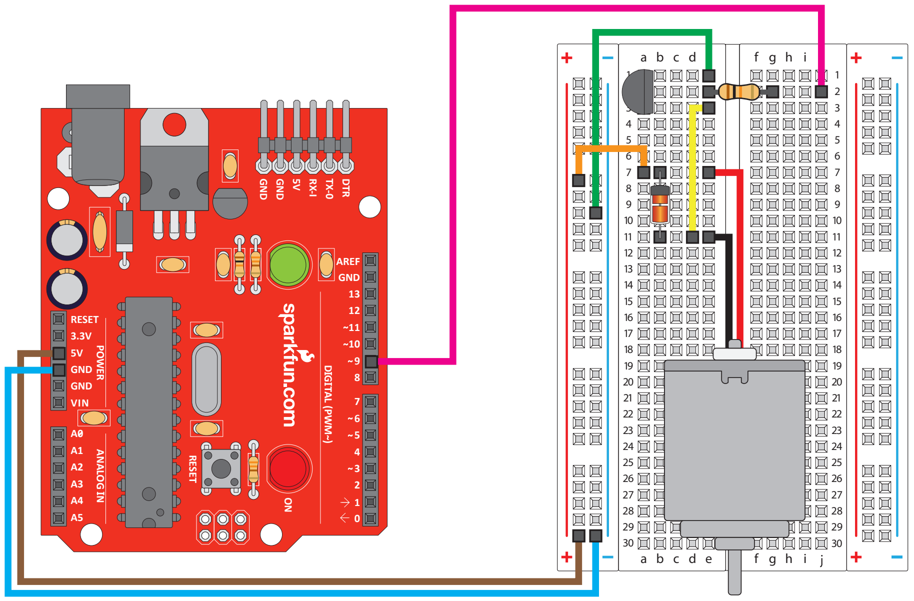
```
void playTone(int tone, int duration) {
  for (long i = 0; i < duration * 1000L; i += tone * 2) {
    digitalWrite(speakerPin, HIGH);
    delayMicroseconds(tone);
    digitalWrite(speakerPin, LOW);
delayMicroseconds(tone);
   }
}
void playNote(char note, int duration) {
  char names[ ] = { ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘a’, ‘b’, ‘C’ };
  int tones[ ] = { 1915, 1700, 1519, 1432, 1275, 1136,
1014, 956 };

  for (int i = 0; i < 8; i++) {
  if (names[i] == note) {
    playTone(tones[i], duration);
     }
   }
}
```

Additional thoughts: This code is fairly complicated. Don’t worry if you don’t understand some aspects of it. If you do understand it, congratulations! You are already ahead of this packet in regards to Arduino code. Just remember, this is another way to use digital pins to create analog output.

Name:
Date:

Circuit #12: Spinning a Motor


Transistor Energy Source
Electricity starts here and varies
Arduino
Pin 9
*
Current will flow into the motor only if the transistor emitter is providing a path to GND because the transistor base is not LOW
Direction of current
Direction of current when transistor base is not LOW
base
transistor
P2N2222AG
collector
emitter
*
Direction of current when transistor base is not LOW
diode
motor
Direction of current only in case transistor is turned off suddenly (to protect motor coil)
Direction of current when transistor base is not LOW
+5 volts
(5V)
GND
(ground) (-)
Electricity ends here
Motor Energy Source
Electricity starts here



Explanation:

The motor in this circuit takes electricity from 5V on the Arduino. The transistor takes electricity from Pin # 9 on the Arduino. The resistor before the transistor limits the voltage so the PWM output from the Arduino affects the motor rate properly. The higher the voltage supplied to the base of the transistor, the more electricity is allowed through the motor circuit to ground. If the transistor base is LOW no electricity is allowed through to ground and the motor will not run. Pin # 9 on the Arduino has PWM capability so it is possible to run the motor at any percentage. The flyback diode connected close to the motor is simply to protect the motor in the rare case that electricity flows from the transistor towards the motor. This only happens if the transistor is shut off suddenly. Finally, after turning the motor and traveling through the forward biased transistor, the electricity reaches ground, closing the circuit and allowing electricity to flow from power source to ground.

Components:

Arduino Digital Pin # 9: Signal power source, PWM output from Arduino board.

Motor: Electric motor, + and – connections, converts electricity to mechanical energy.

Transistor: A semiconductor which can be used as an amplifier or a switch. In this case the amount of electricity supplied to the base corresponds to the amount of electricity allowed through from the collector to the emitter.

Flyback Diode: As in other diodes, current flows easily from the + side, or anode (longer wire), to the - side, or cathode (shorter wire), but not in the reverse direction.

10K Ohm Resistor: A resistor resists the current flowing through the circuit. In this circuit the resistor acts as a ‘pull-down’ resistor to ground.

+5V: Five Volt power source.

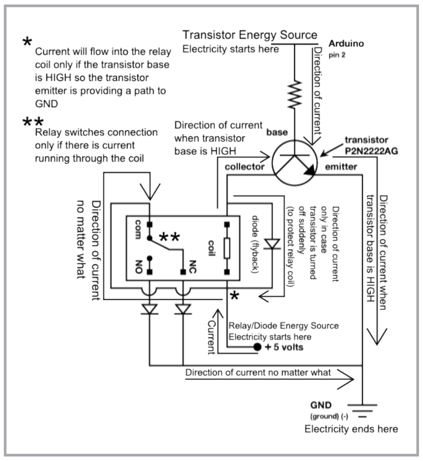
Gnd: Ground

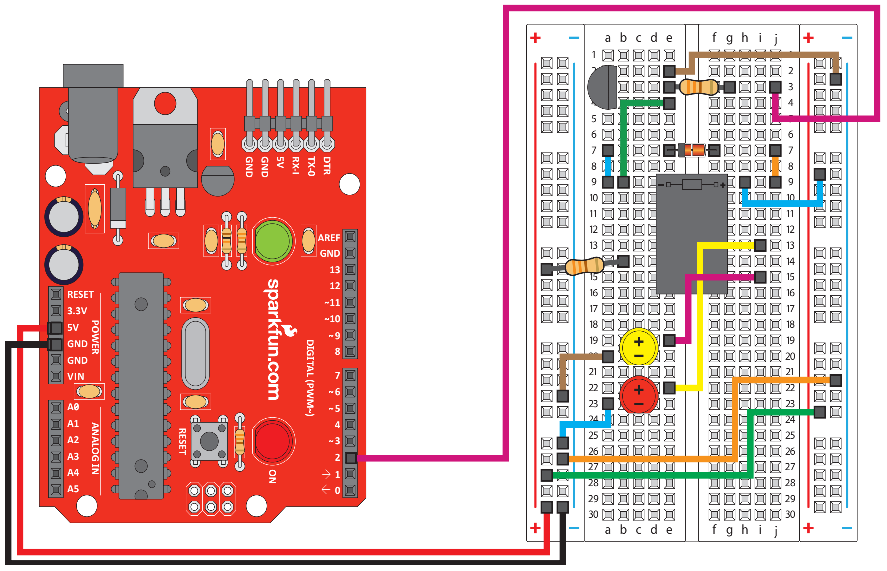
Code:

```
int motorPin = 9;

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

void loop() {
  for (int i = 0; i < 256; i++){
    analogWrite(motorPin, i);
    delay(50);
  }
}
```

Additional thoughts: If you are not familiar with electronics that’s a lot of information. This circuit is great because it teaches about transistors, one of the basic electronic building blocks.

Circuit #13: Relays


* Current will flow into the relay coil only if the transistor base is HIGH so the transistor emitter is providing a path to GND
Transistor Energy Source Electricity starts here
Arduino pin 2
Direction of current
** Relay switches connection only if there is current running through the coil
Direction of current when transistor base is HIGH
base
transistor P2N2222AG
collector
emitter
Direction of current when transistor base is HIGH
Direction of current only in case transistor is turned off suddenly (to protect relay coil)
diode (flyback)
coil
com
NO
NC
Direction of current no matter what
Current
Relay/Diode Energy Source Electricity starts here
+ 5 volts
Direction of current no matter what
GND (ground) (-)
Electricity ends here



Explanation:

The relay circuit gets electricity from the 5V on the Arduino. The electricity always passes through the relay communication line which is switched to either NO (Normally Open) or NC (Normally Closed), lighting up one of the two LEDs. The transistor gets electricity from Arduino Digital Pin # 2 with a resistor to prevent burn out. In this case the transistor receives a digital signal. The transistor closes the circuit when it is sent a HIGH value, allowing electricity to flow through the relay coil, into the collector, out the emitter and to ground, completing the circuit. The energized coil sets the relay switch to NO. The transistor opens or breaks the circuit when it is sent a LOW value, so no electricity passes through the coil and the relay switch is set to NC. The flyback diode connected close to the motor is simply to protect the motor in the rare case that electricity flows from the transistor towards the motor. This only happens if the transistor is shut off suddenly.

Components:

Arduino Digital Pin # 2: Power source, digital output from Arduino board.

Relay: The relay acts as an electrically operated switch between the two LED's.

Transistor: A semiconductor which can be used as an amplifier or a switch. In this case the amount of electricity supplied to the base corresponds to the amount of electricity allowed through from the collector to the emitter.

330 Ohm & 10K Resistors: A resistor resists the current flowing through the circuit. In the transistor circuit it reduces the current so the transistor in the circuit does not burn out.

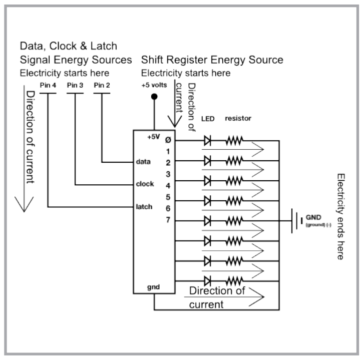
Flyback Diode: As in other diodes, current flows easily from the + side, or anode, to the - side, or cathode, but not in the reverse direction. In this case the diode is being used to prevent current from 'flying back' to the relay in case the transistor is suddenly turned off.

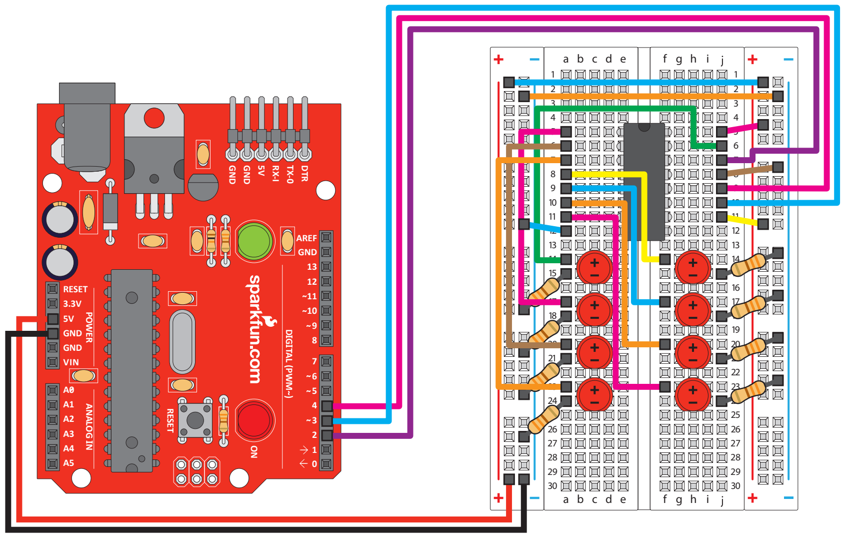
Code:

```
int ledPin =  2;

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

void loop() {
 //set the transistor on
 digitalWrite(ledPin, HIGH);
 // wait for a second
 delay(1000);
 // set the transistor off
 digitalWrite(ledPin, LOW);
// wait for a second
delay(1000);
}
```

Additional thoughts: By now you should be thinking that transistors seem pretty important in the world of electrical circuits. They can be used as switches or amplifiers and they are often called the most important invention of the 20th century. The relay is also a great control component, but it needs something to activate it, hence the transistor.

Circuit #14: Shift Register





Explanation:

The shift register in this circuit takes electricity from 5V on the Arduino. Pin # 2, # 3 and # 4 on the Arduino supply a digital value. The latch and clock pins are used to allow data into the shift register. The shift register sets the eight output pins to either HIGH or LOW depending on the values sent to it via the data pin. The LEDs are connected to the circuit so electricity enters through the anode (+, or longer wire) and exits through the cathode (-, or shorter wire) if the shift register pin is HIGH. The resistor dissipates current so the LEDs do not draw current above the maximum rating and burn out. Finally the electricity reaches ground, closing the circuit and allowing electricity to flow from power source to ground.

Components:

Arduino Digital Pin # 2, # 3 and # 4: Signal power source for data, clock and latch pins on shift register.

Shift register: Allows usage of eight output pins with three input pins, a power and a ground.
Link: http://www.sparkfun.com/datasheets/Components/General/sn74hc165.pdf

LED: As in other diodes, current flows easily from the + side, or anode (longer wire), to the - side, or cathode (shorter wire), but not in the reverse direction. Lights up!

330 Ohm Resistor: A resistor resists the current flowing through the circuit. In this circuit the resistor reduces the current so the LED does not burn out.

+5V: Five volt power source.

Gnd: Ground

Code:

```
int data = 2;
int clock = 3;
int latch = 4;

int ledState = 0;
const int ON = HIGH;
const int OFF = LOW;

void setup() {
  pinMode(data, OUTPUT);
  pinMode(clock, OUTPUT);
  pinMode(latch, OUTPUT);
}

void loop(){
  for(int i = 0; i < 256; i++) {
    updateLEDs(i);
    delay(25);
    }
}

void updateLEDs(int value) {
  digitalWrite(latch, LOW);
  shiftOut(data, clock, MSBFIRST, value);
  digitalWrite(latch, HIGH);
}
```

Additional thoughts: For more advanced components you will need to read documentation or datasheets to figure out how to use them. Any documentation is good as long as you can get the correct information out of it. Datasheets are your friends!

Name:
Date:

// Parts of a Multimeter

Often you will have to use a multimeter for troubleshooting a circuit, testing components, materials or the occasional worksheet. This section will cover how to use a digital multimeter, specifically a SparkFun VC830L. We will discuss how to use this multimeter to measure voltage, current, resistance and continuity on the circuits in the S.I.K.

Display: Where values are displayed.

Knob/Setting: Used to select what is being measured and the upper limit of how much is being measured.

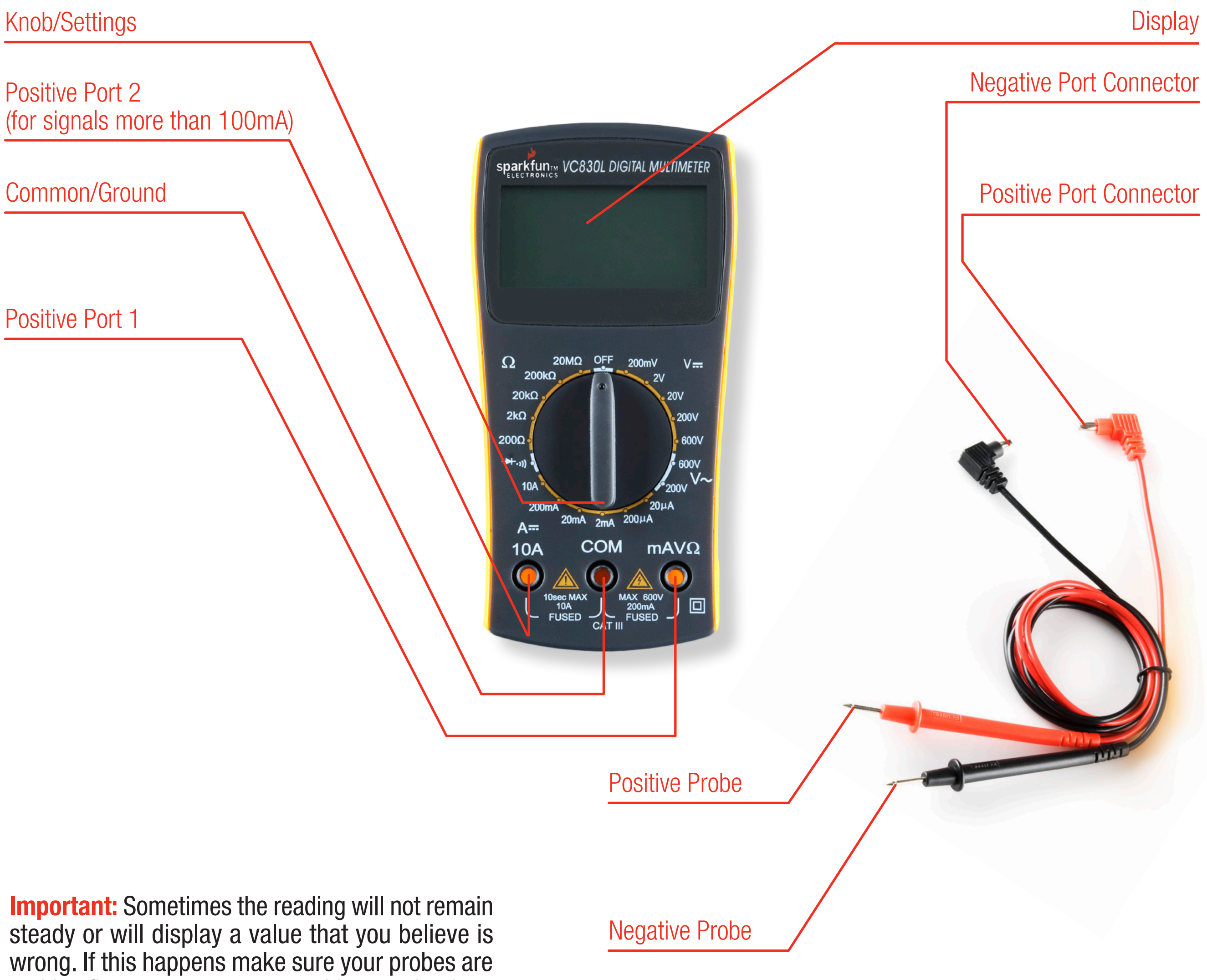
Positive Port 1: Where the positive port connector is plugged in if you are measuring less than 100mA of current.

Common/Ground: Where the negative port connector is plugged in no matter what.

Positive Port 2: Where the positive port connector is plugged in if you are measuring more than 100mA of current.

Probes: The points of contact for measuring electrical signals. Place the positive probe closer to the energy source and the negative probe closer to ground.

Port connectors: Plug them into multimeter.



Important: Sometimes the reading will not remain steady or will display a value that you believe is wrong. If this happens make sure your probes are making firm, constant contact with your circuit on a conductive material.

Name:
Date:

// Settings

There are many different settings depending on how much of a signal the multimeter is being used to measure. This is a good opportunity to talk about unit conversion.

Changing com ports:
Use the first positive com port if you are measuring a signal with less than 100mA of current. Switch to the second positive com port if you are using more. With the Arduino you will usually be using the first positive com port.

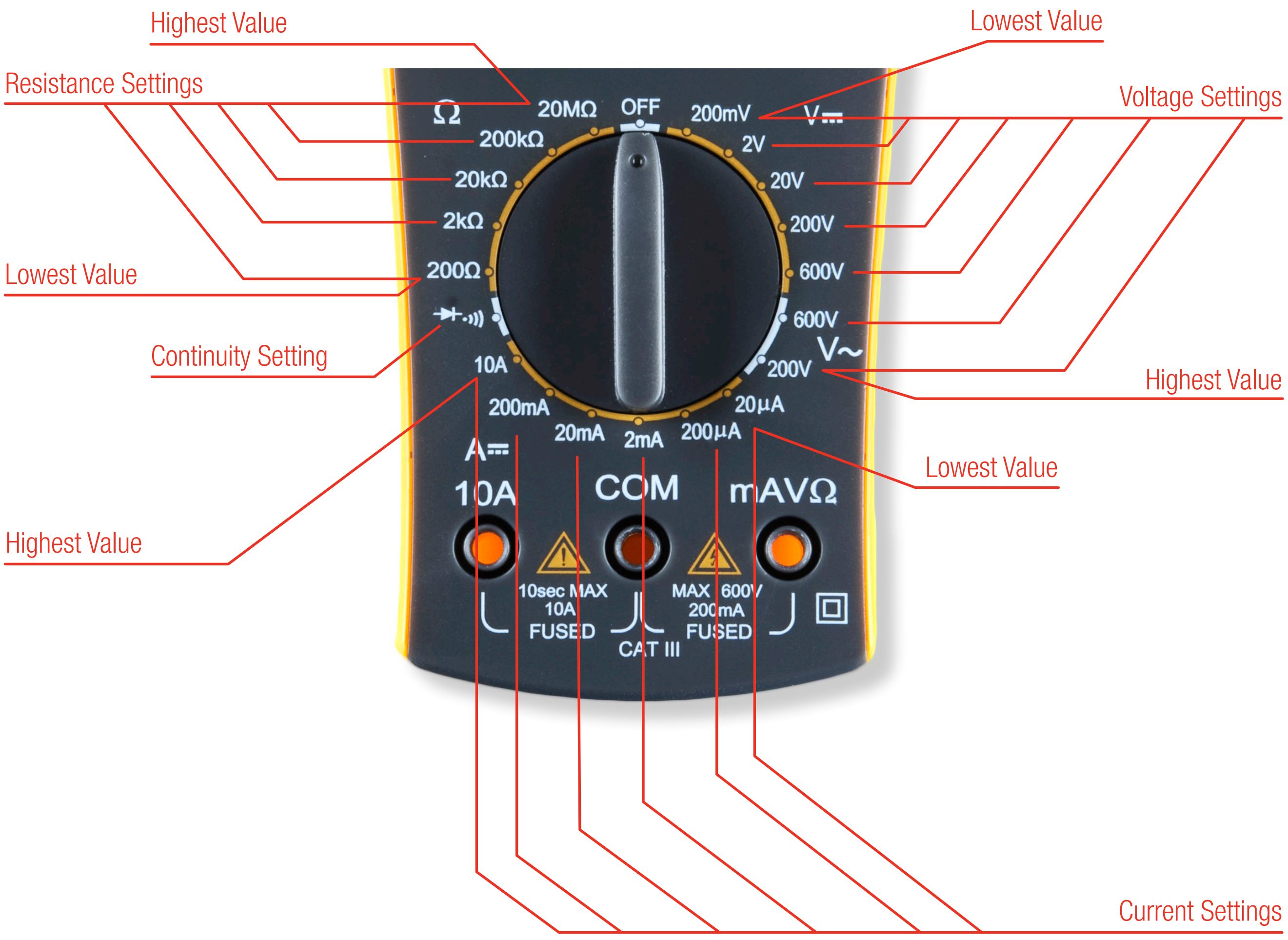
Replacing fuses:
If you try to measure more than 100mA of current through the first positive com port you will most likely blow the fuse in your multimeter. Don't worry, the multimeter isn't broken, it simply needs a new fuse. Replacing fuses is easy, this tutorial explains it: http://www.sparkfun.com/tutorials/202

Voltage: The options for measuring voltage range from 200mV all the way up to 600 Volts.

Current: The options for measuring current range from 20μA all the way up to 10 Amps.

Resistance: The options for measuring resistance range from 200Ω to 20MΩ.

Continuity: This option is for testing to see if there is an electrical connection between two points.



Name:
Date:

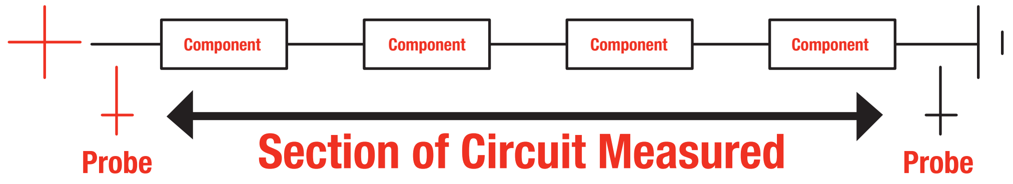
// Measuring Voltage

To start with something simple, let's measure voltage on an AA battery. Pull out your multimeter and plug the black probe into COM ('common') jack and the red probe into mAVΩ. Set the multimeter to "2V". Squeeze the probes with a little pressure against the positive and negative terminals of the AA battery. The black probe is customarily connected to ground or '-' and red goes to power or '+'. If you've got a fresh battery, you should see around 1.5V on the display!

What happens if you switch the red and black probes? Nothing bad happens! The reading on the multimeter is simply negative - so don't worry too much about getting the red or black probe in the right place.

For most RedBoard uses you will be measuring voltages that are 9V or less. Knowing this allows you to start your voltage measurement setting at 20V and workyour way down.

On a circuit use the multimeter to measure voltage from one point in the circuit to another point somewhere along the same circuit. The multimeter can be used to measure the voltage of the whole circuit (if it's going from 5V to GND this will usually read 4.8 to 5V) or just a portion. If you want to measure the voltage of just a portion of your circuit, you have to pay attention to where you place your probes. Find the portion of the circuit you want to measure, and place one probe on the edge of that portion nearest to the energy source. Place the other probe on the edge of that portion nearest to ground. Voila - you have found the voltage of just that section between your probes! Confused? See the schematic images below. Still confused? For more on this see voltage drop.



If your multimeter reads **1.** the multimeter voltage setting you are using is too low. Try a larger voltage setting, if you still encounter the same problem try an even higher setting.

If your multimeter reads **0** the multimeter voltage setting you are using is too high. Try a smaller voltage setting, if you still encounter the same problem try an even smaller setting.

Name:
Date:

// Measuring Resistance

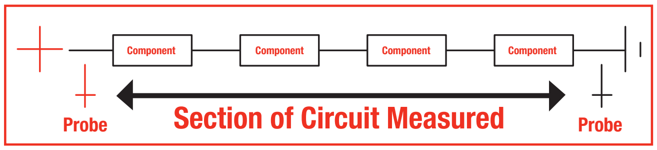
To start with something simple, let's measure the resistance of a resistor. Pull out your multimeter and plug the black probe into COM ('common') jack and the red probe into mAVΩ. Set the multimeter to "2kΩ". Squeeze the probes with a little pressure against the wires on either end of the resistor. The black probe is customarily connected to ground or '-' and red goes to power or '+'. The multimeter will measure the resistance of all the components between the two probes.

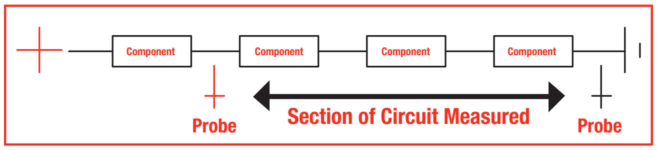
It is important to remember to turn off the power of a circuit before measuring resistance. Measuring resistance is one of the few times you will use a multimeter on a circuit with no power.

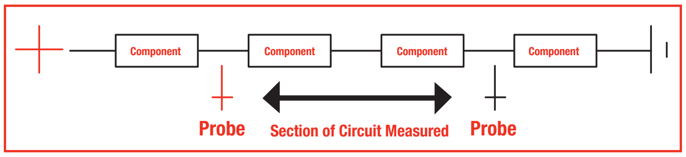
The example below is a 330Ω resistor. Notice the multimeter does not read exactly .330, often there is some margin of error.

When measuring resistance first make sure that the circuit or component(s) you are measuring do not have any electricity running through them.

On a circuit use the multimeter to measure resistance from one point in the circuit to another point somewhere along the same circuit. The multimeter can be used to measure the resistance of the whole circuit or just a portion. If you want to measure the resistance of just a portion of your circuit, you have to pay attention to where you place your probes. Find the portion of the circuit you want to measure, and place one probe on the edge of that portion nearest to the energy source. Place the other probe on the edge of that portion nearest to ground. Voila - you have found the resistance of just that section between your probes! Confused? See the schematic images below. Still confused? For more on this see **Resistance** (Page 48).







If your multimeter reads **1.** the multimeter resistance setting you are using is too low. Try a larger resistance setting, if you still encounter the same problem try an even higher setting.

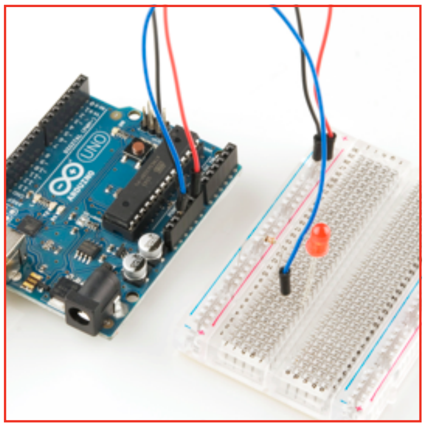
If your multimeter reads **0** the multimeter resistance setting you are using is too high. Try a smaller resistance setting, if you still encounter the same problem try an even smaller setting.

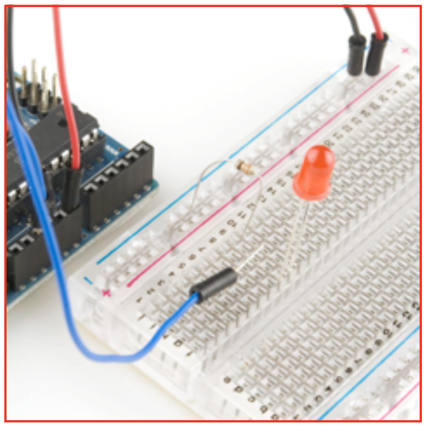
You can measure the resistance of any conductive material whether it is in a circuit or not. Depending on how conductive the material is you may need to change your resistance multimeter setting, or even use a multimeter with a larger range, but if the material is conductive you can measure the resistance of it. This is an easy way to get students to wander around getting comfortable with measuring resistance. Maybe start them off measuring the resistance of some of the S.I.K. circuits, then move to a penny and finally just set them loose to measure anything and everything.

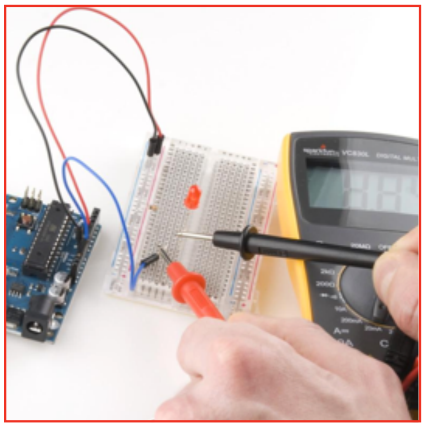
Name:
Date:

// Measuring Current

Ok, we're done with simple. Measuring current is a little more complicated than measuring voltage or resistance. In order to measure current you will need to "break" your circuit and insert the multimeter in series as if the multimeter and it's two probes were a wire. The pictures below are an example of how to measure the current of the first S.I.K. circuit.


Unbroken circuit


Circuit broken by unplugging wire connected to power


Multimeter probes touching wire connected to power and positive lead of LED, putting multimeter in series

It doesn't matter where in the circuit you insert your multimeter. The important thing is that the electricity has no choice but to travel through your multimeter in order to get through the rest of the circuit.

So, pull out your multimeter and plug the black probe into COM ('common') jack and the red probe into mAVΩ. Set the multimeter to "20mA". Squeeze the probes with a little pressure against the two wires you used to "break" your circuit. The black probe is customarily connected closer to ground or '-' and red goes closer to power or '+'. The multimeter will measure the total current running through the circuit.

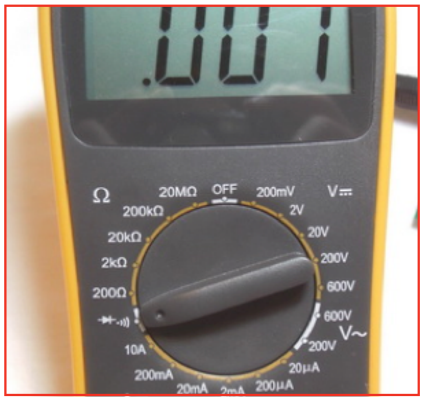
When you are measuring current the multimeter measures the current that is present at that very instant. If your circuit or RedBoard is changing the amount of current you will see that change happen instantly on your multimeter. In order to get a good reading make sure you keep the multimeter connected for at least a couple seconds. (You may also get two readings, a high and a low.)

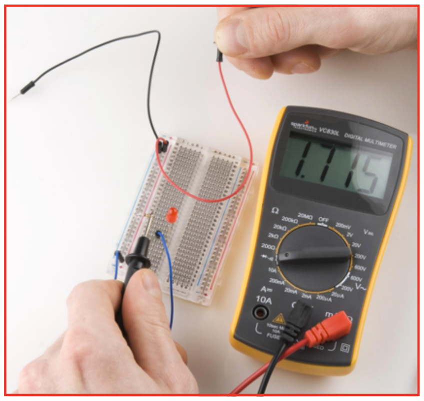
If your multimeter reads **1.** the multimeter resistance setting you are using is too low. Try a larger resistance setting, if you still encounter the same problem try an even higher setting.

If your multimeter reads **0** the multimeter resistance setting you are using is too high. Try a smaller resistance setting, if you still encounter the same problem try an even smaller setting.

Name:
Date:

// Measuring Continuity





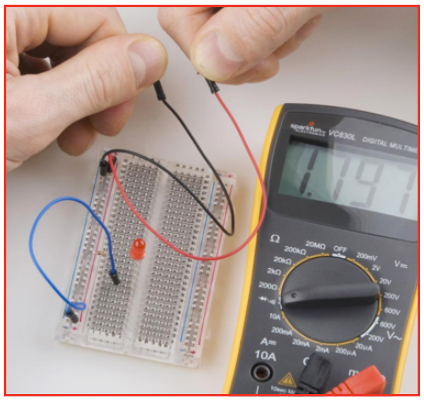
Continuity is how you check to see if two pieces of a circuit are actually connected. The multimeter does this by sending a very small current from the positive probe to the negative, when there is electricity present the multimeter beeps. This is useful when you have a circuit that you think should work but doesn't. Make sure to turn power off when checking continuity.

Set the multimeter to the continuity setting as shown to the right. Touch your probes together and you should hear a beep. This means that electricity is free to travel between the two probes without too much resistance.

If your circuit is plugged in incorrectly, or if it is broken somewhere (maybe your breadboard or a wire is broken) when you touch the probes to the wire providing power and the wire connected to ground the multimeter will not beep. If it were hooked up correctly you would hear a beep and you wouldn't be using the continuity setting!

In order to figure out where the circuit is broken move one of the probes along the circuit towards the other probe. Do this component by component. When the multimeter beeps you know that the probes now detect electricity passing between them so the break must be between where the probe you are moving is now, and where it was the last time the multimeter didn't beep.

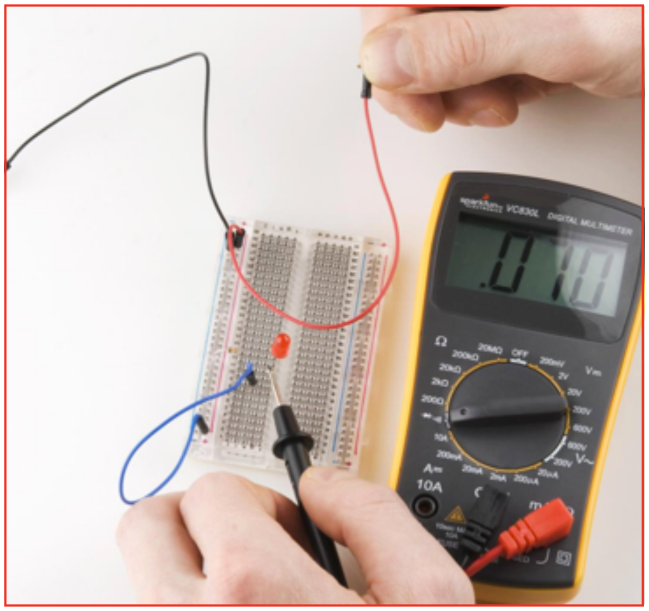
Measuring continuity of the circuit excluding wire connected to ground and resistor. Measured continuity includes LED and two wires connected to breadboard power rail and power. Probes are touching resistor wire and power.



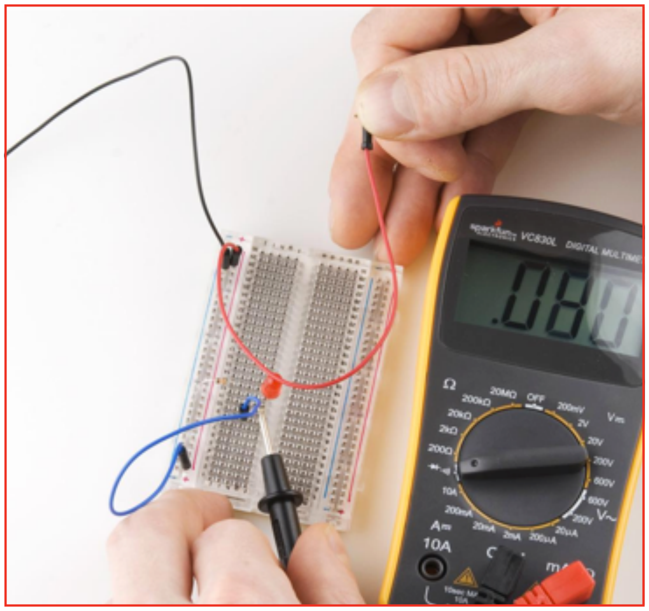
Measuring continuity of the whole circuit from power to ground. Probes are touching wires normally plugged into power and ground.

Name:
Date:

// Measuring Continuity

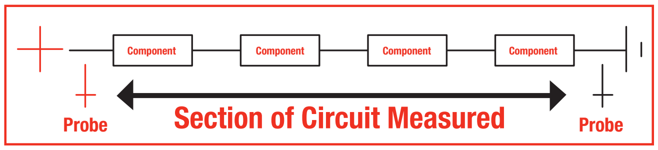


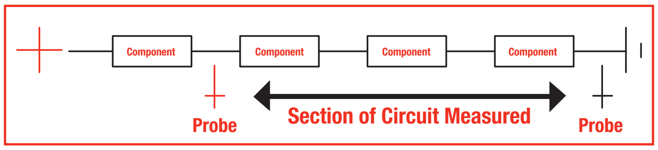
Measuring continuity of the circuit excluding wire connected to ground, resistor and LED. Measured continuity includes two wires connected to breadboard power rail and power. Probes are touching negative LED wire and power.

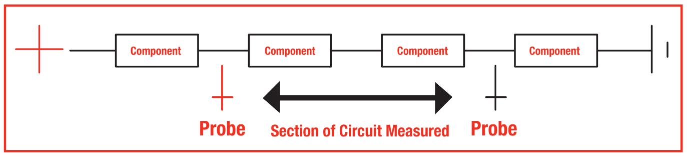


Measuring continuity of the circuit excluding wire connected to ground, resistor and LED. Measured continuity includes two wires connected to breadboard power rail and power. This image looks similar to the image on the left, but the black probe is touching the blue wire, not the negative LED wire. Probes are touching blue wire and power.

The multimeter can be used to measure the continuity of the whole circuit or just a portion. If you want to measure the continuity of just a portion of your circuit, you have to pay attention to where you place your probes. Find the portion of the circuit you want to measure, and place one probe on the edge of that portion nearest to the energy source. Place the other probe on the edge of that portion nearest to ground. Voila - you are testing the continuity of just that section between your probes! Confused? See the schematic images below.







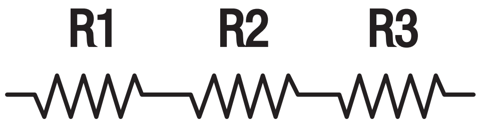
Continuity is one of the most useful settings on a multimeter and you will most likely use it constantly simply to check for connections that aren't quite connected. Breadboards sometimes break so if your multimeter tells you there is no continuity but you know everything is plugged in correctly try switching breadboards.

The beep of the multimeter only tells you that there is very little resistance between the two probes. If there is a resistor in the circuit you will not hear a beep but the display will show a number indicating there is continuity between the two probes.

Name:
Date:

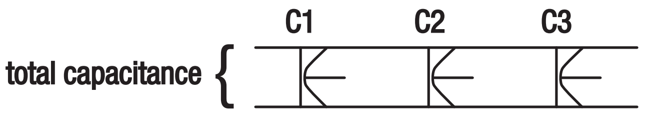
One of the most important concepts in circuit building is the difference between components in series and components in parallel. Basically you can think of components in series as being one after another, like in a chain, while parallel components are hooked up next to each other. It's important to know how certain components affect your circuit when hooked up in these two ways. There are a few things to remember, mostly that resistors and inductors work in the opposite way from capacitors:

Resistors and Inductors **in series** can simply be added together:



R1 + R2 + R3 = total resistance

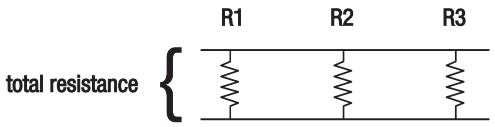
As can capacitors that are **in parallel**:



C1 + C2 + C3 = total capacitance

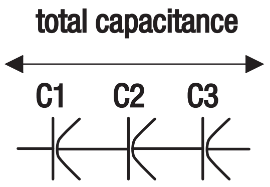
However, for resistors and inductors **in parallel**, as well as capacitors in series, the equation is a bit more complex. Basically the values between any two elements in these setups equal the product of the values divided by the sum of the values. For three elements or more, solve for two and repeat until done. For example (the // indicates that the elements are in parallel):

Resistors:



R1 // R2 =(R1 * R2) / (R1 + R2)
total resistance = (R1 // R2) * R3 / (R1 // R2) + R3

Capacitors:



C1 // C2 = (C1 * C2) / (C1 + C2)
total capacitance = (C1 // C2) * C3 / (C1 // C2) + C3

It seems quite dry, but you never know when basic knowledge like this will come in handy. (Hint: read the next section on powering your projects)

Name:
Date:

When dealing with electronics, it is always a good idea to know how much power you need and how you're going to get it. If you want your project to be portable, or run separately from a computer, you'll need an alternate power source. Plus, not all RedBoard's projects can be run off 5V from the USB port. Fortunately there are a lot of options, one or more of which should suit your purposes perfectly.

Understanding Battery Ratings

One popular way to get power to your project is through batteries. There are tons of different kinds of batteries (AA, AAA, C, D, Coin Cell, Lithium Polymer, etc). In fact, there are too many to go over here-however, they all have a few things in common which can help you choose which ones to use. Each battery has a positive (+) and negative terminal (-) that you can think of as your power and ground. Batteries also have ratings in volts and milliamp hours (written mAh). Given this info along with how much current your circuit will draw, you can figure out how long a battery will last. For example, if I have a battery rated at 1.2v for 2500 mAh, and my circuit requires 100mA (milliamps) current, my battery will last around 17.5 hours. Wait, what? Why not 25 hours you say? Well, you shouldn't drain your battery completely, and other factors such as temperature and humidity can affect battery life, so typically the equation for determining battery life is:

(Capacity rating of battery (in mAh) ÷ Current Consumption of Circuit) x 0.7

Note that we could still use our 2500 mAh battery in a 500mA circuit, but then our battery life would only be 3.5 hours. Make sense? There's a lot to understand about powering circuits, so don't worry if it's not all clicking. Just take an educated guess, be safe, use your multimeter, and make adjustments. It is also worth mentioning that batteries are not the only potential source of power for your project. If your project will be outside or near a window, consider using solar power. There's plenty of good documentation online, but basically, solar cells have the same kinds of voltage and current ratings that any power source might have; the only difference is that the percentage you get from your solar panel depends on how much sunlight it's getting. (Check out http://www.solarbotics.com/) for some good products and documentation using solar power.

So, what if your circuit needs 12v, and all you have are a bunch of 1.5v batteries? Or what if you need your project to be powered for longer, but you don't want to give it too much power? This is where your knowledge of series and parallel may actually come in handy.

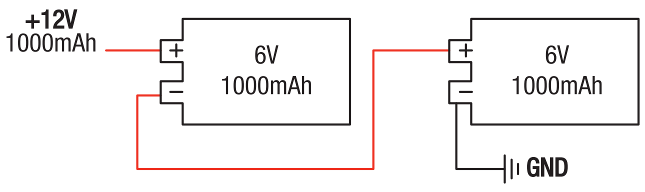
Here's the rule:

Connecting batteries in series increases the voltage but maintains the capacity (mAh) - this what you want to do if you need more power.

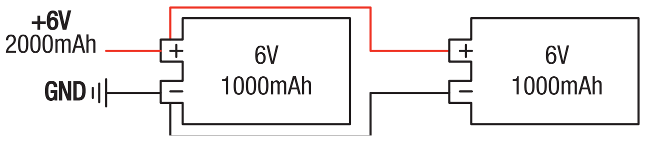
Connecting batteries in parallel maintains the voltage but increases the capacity. This is what you want to do if you need your power supply to last longer.

Here's how to hook them up:

Batteries in Series



Batteries in Parallel



As always, use caution. Batteries of the same kind (same voltage and capacitance) work best in these kinds of situations. Using different kinds of batteries may also work but it is not recommended, as the results are not as predictable.

CHAPTER 2 Resistance

Name:
Date:

Resistance is an important concept when you are creating circuits. Resistance is the difficulty a current encounters when it passes through a component. Everything that electricity passes through provides some measure of resistance: wires, motors, sensors, even the human body!

Measuring voltage, current and resistance are all done in different ways. **To measure resistance you disconnect (turn off) your circuit and place both multimeter leads on either side of the portion of the circuit you wish to measure.** For example: for measuring just a component you would place your leads on the power and ground leads of the component. To measure the resistance of multiple components you leave them connected and place the positive (red) multimeter lead closer to the disconnected power source and the negative (black) multimeter lead closer to the ground. Sometimes you will want to measure the resistance of input and output leads, but more often you will find yourself measuring resistance along the power to ground circuit. It is important to know how much resistance is present in components and circuits for many reasons. Too much resistance and the current will never travel through the whole circuit, too little and the current may fry some of your components! But most importantly you can use resistance to choose the path the current takes through your circuit.

Hook up the circuit below using red LEDs. (Don't hook up the power yet.)

Measure the resistance of each of the possible paths the current can take from power (5v) to ground. There are three possible paths. You will have to measure each component separately and then add the resistance up for the total. You will can add the components' resistance together because the components are in series, if they were parallel it would require more math. Record the total resistance for each circuit below. (Hint: you won't be able to measure the LED)

Circuit 1: ___Ω Circuit 2: ___Ω Circuit 3: ___Ω

Now connect the power and, one at a time, press the two buttons. Which circuit makes the LED the dimmest? Circuit # _______
If you press both buttons which path does the current take? Circuit # _______
If the voltage is staying at 5v in this circuit no matter which paths are closed, there is a way to calculate the current given the resistance. **Write the name of the law and the equation that solves for resistance below. Label all variables.**

____

Now measure the resistance of a potentiometer when it is dialed all the way up and down. Record the highest and lowest values you get.

Highest:________________________ Ω

Lowest:________________________ Ω

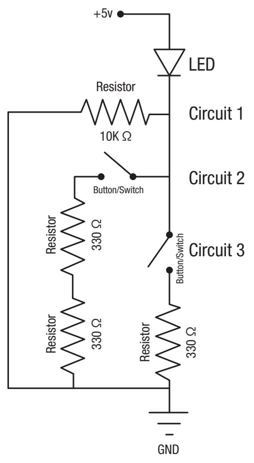
Redraw the schematic below, but use a potentiometer to control the LED brightness instead of the buttons and various resistors. Remember that you must have at least 330Ω of total resistance, otherwise you'll burn out your LED!

Since a circuit or component does not need a current running through it in order to measure the resistance you can take your multimeter and measure the resistance of anything you can think of. Wander around and measure the resistance of various objects. Start with a penny. **Record the most interesting things that have resistance and the value of their resistance below. List at least three.**

____

____

____



Draw Your Schematic

Name:
Date:

Voltage drop is an important concept when you are creating circuits. Voltage drop is the amount that the voltage drops when it passes through a component. The following exercises will show how to measure voltage drop in real life. This is essential when you are fixing your remote control car, electric guitar or even a cell phone.

Measuring voltage, current and resistance are all done in different ways. To measure voltage you connect your positive (red) multimeter lead to the side of the circuit that is closer to your power source and the negative (black) multimeter lead to the side of the circuit that is closer to the ground. It is important to know how much voltage is going through a circuit for many reasons. The most important reasons being that too much voltage can damage your components and too little voltage may not allow electricity to flow all the way through to ground.

Hook up the 5v circuit below using red LEDs.

Close the circuit so only one LED is grounded with the 300 resistor. Insert the end of the resistor not plugged into the ground into a hole on the same row as the first LED's negative lead. The other LEDs don't light up, why is this?

____

____

____

____

Measure the voltage drop across just the LED and record.
_______________v

Measure the voltage drop across the LED and the resistor.
_______________v

Close the circuit so two LEDs light up.

Voltage drop across one LED = _____v
Voltage drop across two LEDs = _____v

Measure the voltage drop across the whole circuit and record.
____________v

Close the circuit so three LEDs light up.

Voltage drop across one LED = _____v
Voltage drop across two LEDs = _____v

Voltage drop for three LEDs = _____v
Voltage drop for whole circuit = _____v

What happened to the LEDs with the last question?

____

____

Now hook up the same circuit to the 3.3V power source without the resistor.
Why don't you need the resistor?

____

____

____

____

Measure the voltage drop across all the LEDs and record.
____________v

Close the circuit so only two LEDs light up.

Voltage drop across one LED = _____v

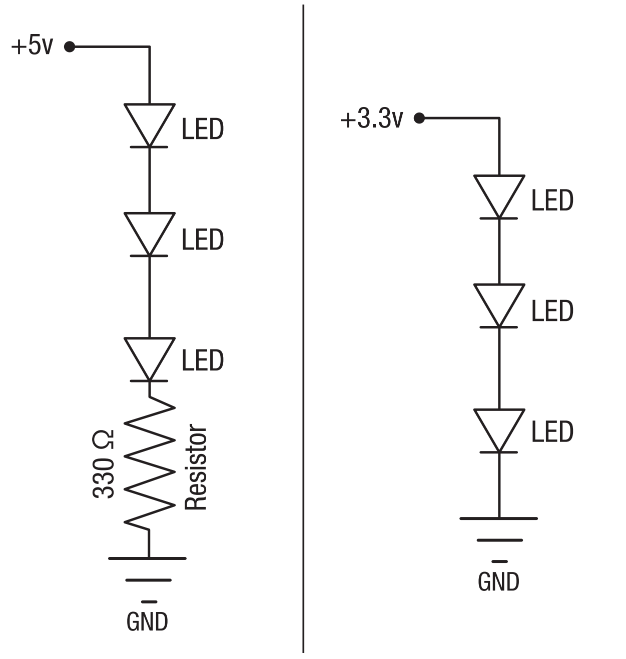
Voltage drop across two LEDs = _____v
Hook up the circuit above to the 5V power source but use the 3.3v as ground.
Wait a second! You can't use a power source as a ground! Or can you?

What is the voltage available and how many LEDs can you light up with it?

Voltage available = ________v

# of LEDs you can light up = __________

Many people think of Gnd as the ONLY place to connect a 'negative' pin, but all you need is a voltage drop from the beginning of a circuit to the end. This difference in voltage is what draws the current in the correct direction.



CHAPTER 2 Transistors

Name:

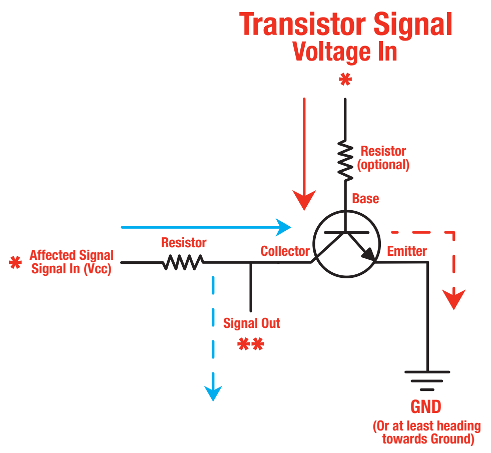
Date:

What is a transistor?
Transistors are semiconductors used to amplify an electrical signal or switch an electrical signal on and off.

Why is a transistor useful?
Often you will need more power to run a component than your Arduino can provide. A transistor allows you to control the higher power signal by breaking or closing a circuit to ground. Combining this higher power allows you to amplify the electrical signal in your circuit.

What is in a transistor?
A transistor circuit has four parts; a signal power source (connects to transistor base), an affected power source (connects to transistor collector), voltage out (connects to transistor collector), and ground (connected to transistor emitter).

How do you put together a transistor?
It's really pretty easy. Here is a schematic and explanation detailing how:



KEY:

✱	Power source somewhere further up the line.
✱✱	Output of transistor. Send this to the component of your choice.
- - →	Direction of current when Voltage In is greater than or equal to the necessary forward bias of the transistor
→	Direction of current
→	Direction of current when a path to ground
- - →	Direction of current when a path to ground is present

The transistor voltage in signal is the signal that is used to control the transistor's base.

Signal in is the power source for the signal out which is controlled by the transistor's action.

Signal out is the output of the signal originating from signal in, it is controlled by the collector.

The amount of electrical current allowed through the transistor and out of the emitter to ground is what closes the entire circuit, allowing electrical current to flow through signal out.

Ok, how is this transistor information used?

It depends on what you want to do with it really. There are two different purposes outlined above for the transistor, we will go over both.

If you wish to use the transistor as a switch the signal in and voltage in signal are connected to the same power source with a switch between them. When the switch is moved to the closed position an electrical signal is provided to the transistor base creating forward bias and allowing the **electrical signal** to travel from the **signal in** to the **transistor's collector** to the **emitter** and finally to **ground**. When the circuit is completed in this way the **signal out** is provided with an electrical current from **signal in**.

The signal amplifier use of the transistor works the same way only **Signal In** and **Voltage In** are not connected. This disconnection allows the user to send differing values to the base of the transistor. The closer the **voltage in** value is to the saturation voltage of the transistor the more electrical current that is allowed through the emitter to ground. By changing the amount of electrical current allowed through to ground you change the signal value of signal out. For examples of transistor uses see S.I.K. circuits # 12 and # 13.

CHAPTER 2 Voltage Dividers

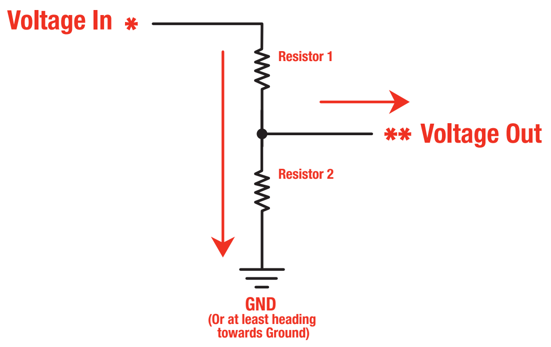
Name:
Date:

What is a voltage divider?
Voltage dividers are a way to produce a voltage that is a fraction of the original voltage.

Why is a voltage divider useful?
One of the ways a voltage divider is useful is when you want to take readings from a circuit that has a voltage beyond the limits of your input pins. By creating a voltage divider you can be sure that you are getting an accurate reading of a voltage from a circuit. Voltage dividers are also used to provide an analog reference signal.

What is in a voltage divider?
A voltage divider has three parts; two resistors and a way to read voltage between the two resistors.

How do you put together a voltage divider?
It's really pretty easy. Here is a schematic and explanation detailing how:



KEY:

*	Power source somewhere further up the line.
**	Output of voltage divider. Send this to input pins or a circuit that needs a lower voltage than the original voltage source.
→	Direction of current

Often resistor # 1 is a resistor with a value that changes, possibly a sensor or a potentiometer.

Resistor # 2 has whatever value is needed to create the ratio the user decides is acceptable for the voltage divider output.

The Voltage In and Ground portions are just there to establish which way the electrical current is heading, there can be any number of circuits before and after the voltage divider.

Here is the equation that represents how a voltage divider works:

$$V_{out} = V_{in} \frac{R_2}{(R_1 + R_2)}$$

If both resistors have the same value then Voltage Out is equal to ½ Voltage In.

Ok, how is this voltage divider information used?

It depends on what you want to do with it really. There are two different purposes outlined above for the voltage divider, we will go over both.

If you wish to use the voltage divider as a sensor reading device you first need to know the maximum voltage allowed by the analog inputs you are using to read the signal. On an Arduino this is 5V. So, already we know the maximum value we need for **Vout**. The **Vin** is simply the amount of voltage already present on the circuit before it reaches the first resistor. You should be able to find the maximum voltage your sensor outputs by looking on the datasheet. This is the maximum amount of voltage your sensor will let through given the voltage in of your circuit. Now we have exactly one variable left, the value of the second resistor. Solve for R2 and you will have all the components of your voltage divider figured out! We solve for R1's highest value because a smaller resistor will simply give us a smaller signal which will be readable by our analog inputs.

Powering an **analog Reference** is exactly the same as reading a sensor except you have to calculate for the Voltage Out value you want to use as the analog Reference.

Given three of these values you can always solve for the missing value using a little algebra, making it pretty easy to put together your own voltage divider. The S.I.K. has many voltage dividers in the example circuits. These include: Circuits # 2, 5, 6, 9 and 10.

3

Programming

Name:
Date:

// Basic Operators

Often when you are programming you will need to do simple (and sometimes not so simple) mathematical operations. The signs used to do this vary from very simple to confusing if you've never seen them before. Below is a table of definitions as well as some examples:

Arithmetic Operators

Arithmetic operators are your standard mathematical signs

+ (addition)
- (subtraction)
* (multiplication)
/ (division)
% (modulus)
= (assignment)

Relational Operators

Relational operators are used to compare values and variables

== (equality)
!= (inequality)
> (greater-than)
< (less-than)
>= (greater than or equal to)
<= (less than or equal to)

Pay attention to = and ==.
= is used to assign variable values,
== to compare values.

Logical Operators

Logical operators are used to join two or more conditional statements together

! (NOT)
&& (AND)
|| (OR)

Relational Operator Example

```
if (x!=7){
//loop body code here
}
```

Compares x to the number 7,
executes code inside body loop
if the value of x **does not equal 7**

Logical Operator Example:

```
if ((x==7)||(x==9)){
//loop body code here
}
```

Compares x to the number 7 and 9,
executes code inside body loop
if the value of x equals 7 **or** 9

Name:
Date:

// Comments

As you use code other people have written you will notice //, /* and */ symbols. These are used to “comment” lines out so they do not affect the code. This way people who write code can add comments to help you understand what the code does. Good code has comments that explain what each block of code (functions, classes, etc.) does but does not explain simpler portions of the code as this would be a waste of time. Commenting lines out is also a very useful tool when you are writing code yourself. If you have a section of code you are working on, but isn’t quite finished or doesn’t work, you can comment it out so it does not effect the rest of your code when you compile or upload it.

//

This is used to comment out a single line

```
//commented out line
```

/*

This is used to start a section of commented lines

```
/*comments start here
```

*/

This is used to end or close a section of commented lines

```
comments end here*/
```

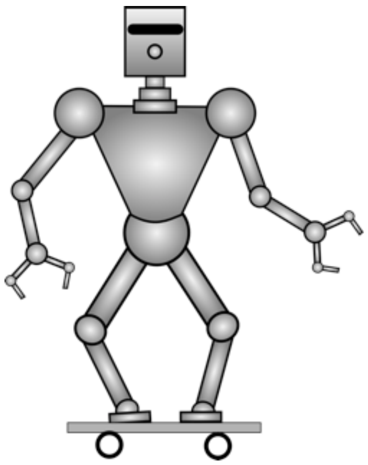
Name:
Date:

// Vocabulary: Variable, Boolean, Integer, Character, Value

Variables are one of the most important concepts in computer programming. But what exactly are **variables**? **Variables** are like baskets that hold pieces of information. There are a couple different kinds of **variables** depending on what kind of information you need to keep track of. You have probably already heard of most of the different kinds of **variables**. Here are the definitions of three different kinds of **variables**. There are more types of variables, but, let's start with these.

- **Boolean variable:** A boolean variable can be true or false (one or zero).
- **Integer variable:** An integer variable can be any whole number between –32768 and 32767.
- **Character variable:** A character variable can be any one letter (or punctuation or symbol).

Below is a robot, answer the questions to the right of the robot and be as silly as you want. Then write the type of **variable** you would use to store this information. For a **boolean** write "boolean", for an **integer** write "int" and for a **character** write "char".



Is this robot good at skateboarding? ________________	Variable type:____________
How old is this robot?___________________________	Variable type:____________
What is the first letter of this robot's name?___________	Variable type:____________
How many years has it been skateboarding?__________	Variable type:____________
Is it wearing pants? ____________________________	Variable type:____________
What is the first letter of the robot's dog's name? _______	Variable type:____________
Is the robot going to crash? _____________________	Variable type:____________
How many feet of air has this robot gotten? __________	Variable type:____________

The number, or character, you put into a **variable** is called its **value**. Once you have created a **variable** you can change the **value** whenever you need to. For example, if we decided the robot is 1000 years old, in a year we need to be able to change its age to 1001. First we need to create a **variable** to keep track of its age. We can name the **variable** whatever we want, but "age" makes sense so we'll go with that. Then we need to put a **value** into the **variable**. The first **value** was 1000, but a year later we delete that **value** and replace it with the new **value**, 1001. Pretty easy, huh? If we wanted to keep track of how old the robot used to be when we met it we could create a new **variable** called "ageWeMet". That way when we have to change the "age" **variable** we can keep track of how old the robot was when we met it in the other **variable** "ageWeMet". You may have noticed that there are no spaces in the name of this second **variable**. That is because **variable** names can't have any spaces.

Circle the **variable** in the sentences below and put a box around the **value**.

The robot's favorite letter is Q. The robot's height is 100 ft.

The robot's power is on.

Name:
Date:

// Vocabulary: Boolean, Declare, Assign

OK! You're ready to start programming your first **boolean** variable. Anytime you see *italics like this* it is an example of how you would write something in the Arduino language.

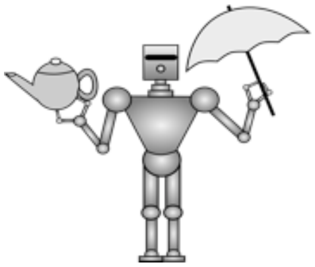
- A **Boolean** variable is the simplest kind of variable, it is either true or false.
- True is represented by a one or HIGH and false is represented by a zero or LOW.
- HIGH can be used as true, but it means there is electricity flowing through a circuit.
- LOW can be used as false, but it means there is no electricity flowing through a circuit.
- To create a **Boolean** variable you type the following: *boolean variableName;*
- Creating a variable is called "**declaring**" a variable.
- The variableName can be anything you like, but it should make sense to you.

For example you could **declare** a **Boolean** variable named *dayLight(boolean dayLight;)* that represents whether it is daytime or not. Once you have **declared** your variable it is not equal to anything, it is empty and waiting for you to set it equal to true or false. To do this you type the following: *dayLight = true;* or *dayLight = 1;*. (Don't forget the ; at the end, it's very important! It is called a semicolon and it tells the computer that you are finished doing something.)

This means that dayLight is true, and you can see the sun. Setting a variable equal to a value is called "**assigning**". **Declare** three **Boolean** variables about the robot on this page in the spaces below and then **assign** them values of true or false (or one or zero). Remember, you can name the variables whatever you want! They're your variables, it's up to you. Look at the example above if you are unsure of how to **declare** and **assign**. (Don't forget the semicolons at the end of each line, they're important!)

Declare:

Assign:



List three of the silliest things you can think of that you might keep track of with a **boolean** variable. Examples: Do I have peanut butter in my ear? Are penguins good to use as dodgeballs?

____

____

____

____

Now pick one of the silly ideas above. In the space below **declare** your silly variable and then **assign** it a value. For example: *boolean peanutButter; peanutButter = true;* This means that I do have peanut butter in my ear... maybe I am saving it for lunch.

____

____

____

____

Name:
Date:

// Vocabulary: Integer, Declare, Assign

Wow! You're ready to start programming your first **integer** variable. Anytime you see *italics* it is an example of how you would write something in Arduino language.

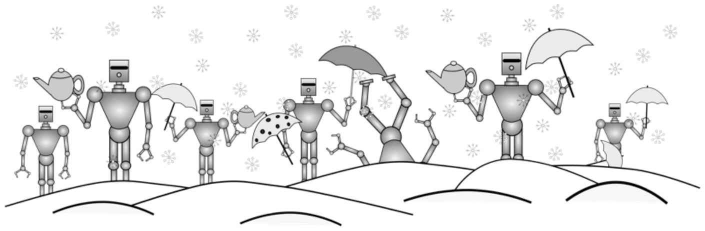
- An **Integer** variable is a number (no fractions or decimals) between -32768 and 32767.
- To create an **Integer** variable you type the following: *int variableName;*
- This is called "**declaring**" a variable.
- The variableName can be anything you like, but it should make sense to you.
- To **assign** an **Integer** variable the value 120 type the following: *variableName = 120;*

For example you could **declare** an **Integer** variable named clouds (*int clouds;*) that represents the number of clouds in the sky. Once you have **declared** your variable it is not equal to anything, it is empty and waiting for you to set it equal to a number between –32768 and 32767. To do this you type the following: *clouds = 8;*. (Don't forget the ; at the end. This is called a semicolon and it's how the computer knows you are finished doing something.)

This means that you can see eight clouds in the sky. Setting a variable equal to a value is called "**assigning**". **Declare** three **Integer** variables about the picture on this page in the spaces below and then **assign** them values between –32768 and 32767. Include at least one variable with a negative value and one variable with a value greater than ten. Feel free to make up variables and values that you can't actually see in the picture. Try to keep it making sense. Look at the example above if you are unsure of how to **declare** and **assign**. (Don't forget the semicolons at the end of each line!)

Declare:

Assign:



List three of the silliest things you can think of that you might keep track of with an **integer** variable. Example: How many pieces of ham do I have in my pocket? How many bugs could you fit in a rocket?

Now pick one of the ideas above. In the space below declare your variable and **assign** it a value. For example: *int ham; ham = 1073;* I either have big pockets or small pieces of ham.

Name:
Date:

// Vocabulary: Character, Declare, Assign

OK! You're ready to start programming your first **character** variable. Anytime you see *italics* it is an example of how you would write something in the Arduino language.

- A **Character** variable is a single letter, symbol or number.
- To create a **Character** variable you type the following: *char variableName;*
- This is called "**declaring**" a variable.
- The variableName can be anything you like, but it should make sense to you.
- To **assign** a **Character** variable the value "Q" you type the following: *variableName = 'Q';*

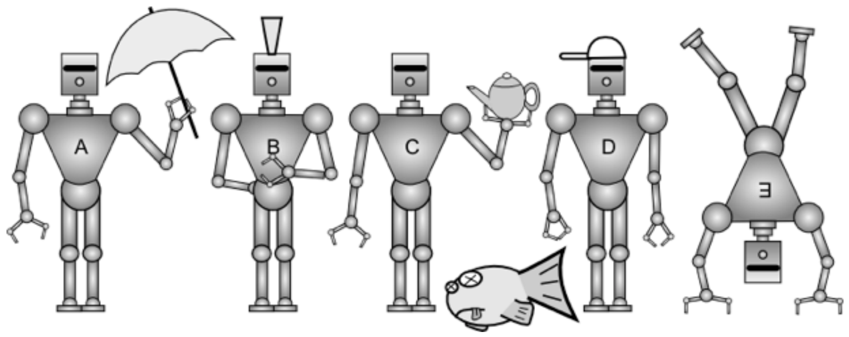
For example you can **declare** a **character** variable named weather (*char weather;*) that uses a letter to represents the weather. You can use the letter R to mean it is raining, S for snow, and C for clear. Once you have **declared** your variable it is not equal to anything, it is empty and waiting for you to set it equal to a **character**.

To do this you type the following: *weather = 'C';*. (Don't forget the ; at the end. This is called a semicolon and it's how the computer knows you are finished doing something.) Also, there are many different **character** types other than a letter: !?*%$&@ are all valid **character**s.

For example, *weather = 'C';* means that the sky is clear, but that's just because you decided it means that. C could mean whatever you need to keep track of. For example C could mean that it is cold out, if that's what you decided. Setting a variable equal to a value is called "**assigning**". **Declare** three **Character** variables about the picture on this page in the spaces below and then **assign** them **character** values that make sense. Check the example when you are **assigning** a value, this can get tricky. Make sure the variable names describe the object you want to keep track of. Look at the example above if you are unsure of how to **declare** and **assign**. (Don't forget the quotation marks and semicolons at the end of each line!)

Declare:

Assign:



List three of the silliest things you can think of that you might keep track of with a **Character** variable. Example: What color lollipops do robots eat? What's a pirate's favorite letter?

Activity

CHAPTER 3
Programming Concepts, Variables

Name:
Date:

Purpose: Group activity teaching how to declare and assign the variable types Boolean, Integer and Character. Text in italics denotes actual Arduino code.

Materials: None

Vocabulary to be explained prior to activity:
Variable: A way to store a piece of information that may change.
Value: Piece of information assigned to a variable.
Declaration: Creating a variable, when you declare a variable it has no value.
Assignment: Sets or resets the value of a variable.

Types of variables:
Boolean: This variable type has only two values. True or false, which can also be represented as one and zero or HIGH and LOW. Arduino syntax: *boolean*
Integer: This variable type is used to store whole numbers. Because RedBoard uses two bytes to store integers it can only store numbers from –32768 to 32767. Arduino syntax: *int*

Character: This variable is used to store any character you can type on a keyboard (and some you can't). It is basically an integer, but it is used for letters and characters. It is mainly used to print messages or send messages when human interaction is needed. Arduino syntax: *char*

Declaring variables:
Boolean: *boolean variableName;*
variableName can be anything as long as it makes sense and has no spaces in it.
Example: *boolean pamHappy;* This variable could be used to indicate if Pam is happy or not. Remember the semicolon, it's important!

Integer: *int variableName;*
variableName can be anything as long as it makes sense and has no spaces in it.
Example: *int pamAge;* This variable could be used to indicate how old Pam is. Remember the semicolon, it's important!

Character: char *variableName;*
variableName can be anything as long as it makes sense and has no spaces in it.
Example: *char pamShirtColor;* This variable could be used to indicate the color of Pam's shirt. Remember the semicolon, it's important!

Assigning variables:
Assigning variables is really easy! No matter what type of variable you simply type the variable name followed by a single equals sign and then the value you are assigning to your variable followed by a semicolon. Example: *pamShirtColor = 'p';* Values have certain requirements depending on their types. A boolean needs to be true or false (or one or zero), an integer should be a number between -32768 and 32767 and a character should be a single character with single quotation marks around it. **Finally, remember the semicolon, it's important!**

Activity

Activity

CHAPTER 3
Programming Concepts, Variables

Name:
Date:

Activity:
Students should have completed the introduction to variables worksheet that comes with this activity. Examples of variable types, declarations and assignments can be posted somewhere visible in the classroom to help students who are not completely comfortable with the concepts yet.

Students go around in a circle declaring variables that apply to themselves and other students. For example, if they wish to declare a variable about their age they would need to declare an integer variable with a name that makes sense. It is up to the students how specific they want to get, they can declare an integer variable named age, or they could go so far as to declare a variable named pamAge. The difference is that the variable age can apply to anyone, the variable pamAge is specific to a person named Pam. A boolean variable can be used for any quality that is either yes or no. For example, a student might declare pamHappy as a boolean variable to indicate whether Pam is happy or not. Character variables can be used to keep track of anything that does not fit nicely into either integer or boolean. For example, a student may create a variable called pamShirtColor. Declaration of variables should be in the syntax used in Arduino, for examples see previous page.

Once each student has declared a variable go around the circle and have each student assign a value to their variable. Assignment of variables should be in the syntax used in Arduino, for examples see previous page.

Additional activities:
Students can declare their variables on pieces of construction paper. Each variable type should have a distinct color or shape (or some other way to identify the variable type other than the declaration). Students can write their variable declaration and assignment for display and personalize the construction paper so it makes sense with their variable name. Throughout the unit students should be encouraged to reassign the value assigned to their variable if it changes. Obviously you will probably want to have a designated time for variable reassignment to avoid classroom disruption. For example, Pam may declare char pamShirtColor; on a shirt shaped piece of yellow construction paper (yellow to designate it a character variable). Pam can then tape a piece of paper with the letter 'B' (don't forget the single quotation marks) to indicate she is wearing a blue shirt. The next day Pam may then replace the letter 'B' with a 'P' to indicate that today she is wearing a purple shirt. You may want to limit reassignment to once a week if your class has a tendency to be overzealous about activities like this.

If your students are having difficulty with the concept of variable types try this activity: Create three different shaped holes in a board, designate one hole for each of the three variable types. Label each hole with the corresponding variable type and definition. Create or buy a bunch of objects that can only fit through one of the holes and label the objects with values that correspond to the variable type. Give the objects out to students and explain that each object can only be one of the three different type of variables and the students need to match up the objects with the variable types by putting them in the corresponding holes.

Activity

Name:
Date:

// Vocabulary: If, Parenthesis, Curly Brackets

The *If statement* is one of the most basic building blocks in computer programming. The easiest way to understand a computer language If statement is to look at real life *If statements* first. *If statements* have two different parts, the question and what happens if the answer to the question is yes. Below are a bunch of real life if statements. On the left are the questions or "if" portions of the *If statements*. On the right are the actions that happen when the answer to the questions are true. Unfortunately only the first *If statement* is connected to the correct action, the rest are up to you.

Draw a line between the two that make the most sense together.

The first one is done for you:

If you play around with electronics	→	**Then you can build some cool stuff.**
If you run over a porcupine with your bike		**Then your feet will smell funny.**
If you are an alien		**Then you pollute less.**
If you do push ups and pull ups		**Then you say Arrrrr a lot.**
If you put peanut butter in your sock		**Then you have feathers and don't like cats.**
If you eat too much candy		**Then you might catch a fish or fall in.**
If you bike everywhere you go		**Then you might have six arms and one eye.**
If you go fishing in a canoe		**Then someone might sing Happy Birthday.**
If you are a pirate		**Then you get stronger.**
If you today is your Birthday		**Then you get a flat tire.**
If you are a parakeet		**Then you get sick.**

In computer programming the *If statement* works the same way as real life. There is a question and something that happens if the answer to the question is "yes". The question is written inside of the parenthesis ( ) and whatever happens if the question is true is written inside of the curly brackets { }.

Here are a couple examples of pseudo-code versions of *If statements:*

If (you play around with electronics){then you can build some cool stuff}

If (you remember parenthesis and curly brackets){then *If statements* are easy}

If (you understand *If statements*){then you are on your way to learning programming}

Just remember: If (the answer to this question is yes) {then do this}

Example of an *If statement*:

```
if ( val == HIGH ) {
digitalWrite ( ledPin, LOW );
}
```

All *If statements* start with "if" followed by the question in parenthesis. In this example the question is; does the variable "val" equal HIGH? (HIGH is a boolean value that is the same as true. HIGH means there is electricity present and LOW means there is not.) If "val" does equal HIGH then Arduino does whatever is inside of the two curly brackets { }. In this case it tells ledPin it should not conduct electricity. Here is a pseudo-code of the same If statement:

If (the variable "val" has electricity running through it) {then tell (the pin ledPin, to turn off) }

If parts of this last example don't make sense, don't worry, the important thing is to understand what an If statement is. So... If (the last example didn't make sense) {don't worry}.

Name:
Date:

Write three of the funniest, or most interesting, *If statements* you can think of in the space below. Don't worry about putting them inside of parenthesis and curly brackets, we'll get to that later.

Example 1: If dinosaurs were still alive then we would have to run a lot more.

____

____

____

Now write your *If statements* the way they would look with the parenthesis. Don't forget the difference between the two different kinds of parenthesis!

Example 1: If (dinosaurs were still alive) {then we would have to run a lot more.}

____

____

____

But what if there are two or more things that could happen if the question is true?

Example 2: If dinosaurs were still alive then we would have to run when we were outside, but if they were our pets we could walk and we would need really big litter boxes.

Is this really just one *If statement?* No, it's actually two, and one of the *If statements* is inside the other. Don't worry! This is ok, in fact it happens all the time. Here is how it looks in pseudo-code:

Example 2:

```
If (dinosaurs were still alive){
then we would have to run a lot more, but
If (they were our pets) {
we could walk and we would need really big litter boxes} }
```

It may look complicated but it's just one *If statement* inside of another. There is no limit to how many *Ifs* you can put inside of another *If statement*. Go ahead and write one *If statement* with another *If statement* inside of it in plain English below. Make sure you use the word "if" twice.

____

____

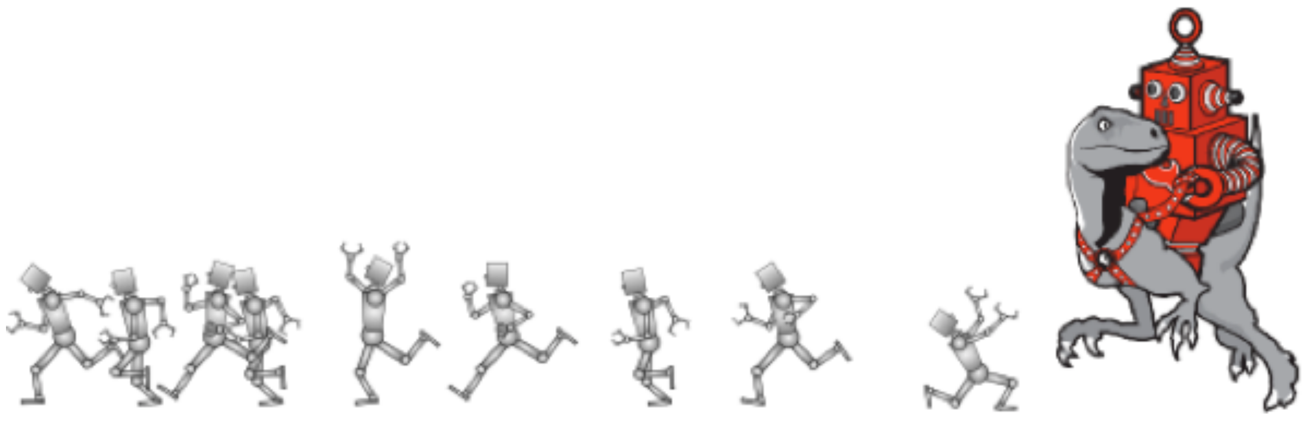
____

Now you're going to take that sentence and turn it into pseudo-code. Pay attention to where the parentheses and curly brackets are and how many there are. Start with writing the first question, put a curly bracket just after the question like this { and then put a curly bracket at the very end of the lines like this }. Now put what happens when the question is true and the second *If statement* inside of your first two curly brackets. If (you're confused) {look at example number two.}

____

____

____



Name:
Date:

Now that you understand the basics of *If statements* you're going to practice filling in various parts of some *If statements*. These *If statements* are not written in code, but you should be getting comfortable with what goes where as well as the parenthesis and curly brackets. Remember, you will only do what is in the curly brackets if the question is true. Fill in the blanks and if you feel like it make them funny.

If (____)
{ then you can fly. }

(your dog runs away)
{ then you need to go looking for your dog. }

If (you are hungry)
{ ___________________________ }

If (____)
{ then you burp. }

If (you want to become an astronaut)
{ _______________________________ }

If (_____________________________)
{ ____________________________ }

(you build a robot)
{ ____ }

If (you build an electronic drum set)
then you can practice quietly.

If (you are an elephant)
{ _______________________________ }

(you make pancakes)
{__________________________}

If (_____________________________________)
{ you should hit the pinata. }

(you want pizza)
{______________________________}

Activity

CHAPTER 3

Programming Concepts, If Statements

Name:

Date:

Purpose: Group activity teaching the concept of *If statements* and their syntax.

Materials: Cut up sheet of silly conditionals and actions.

Vocabulary to be explained prior to activity:

If statement:
These simple statements exist in real life as well as in computer programming. They are simple statements that indicate if something is true or has occurred, then a resulting action takes place.
If statement pseudo-code: If (conditional) { action }

if:
The word that always starts an *If statement*. It's never capitalized.

Parenthesis ( ) :
Indicates and bookends the conditional portion of an *If statement*.

Conditional:
The question or condition that if true initiates the action of the If statement.

Curly brackets { } :
Indicates and bookends the action portion of an *If statement*.

Action:
Portion of code that occurs when the conditional is true. This can be anything including another *If statement*.

Activity:

Preparation:
Cut up the conditional and action portions of the silly *If statements* included with this activity, or you can write your own and cut those up.

Activity:
First mix and then distribute the slips of paper among your students. Explain the concept of an *If statement* to your students and then have them try to match up all the conditionals with the resulting actions. It is possible to mismatch the conditionals and actions, but this portion of the activity is mainly to have fun and establish the idea of a conditional and a resulting action, so don't worry if the kids mismatch some, just make sure you get some laughter out of this portion of the activity.

Second have seven students stand up to model portions of the *If statement*. The first student is the "If", the second student is the first parenthesis, the third student is the conditional, the fourth student is the closing parenthesis, the fifth student is the first curly bracket, the sixth student is the resulting action and the final student is the closing curly bracket. Students then model one of the silly *If statements* they have matched up. Each student reads or says aloud the portion of the *If statement* they represent. Once the seven students have gone through the *If statement*, the last student sits down, all the standing students move one space over to the right and a new student stands up to join the group as the "If" portion. Students should cycle through this way until either everyone has had a turn to be each part of the *If statement*, or all the silly *If statements* have been used up. Encourage students who are representing the parenthesis and curly brackets to make parenthesis and curly brackets with their arms to demonstrate which are opening parenthesis and curly brackets and which are closing parenthesis and curly brackets.

Once the *If statements* and position of the parenthesis and brackets have been established in your classroom you can use the semantics where ever you see fit. For example, If (we line up quickly and quietly) { then we will have more recess time. }

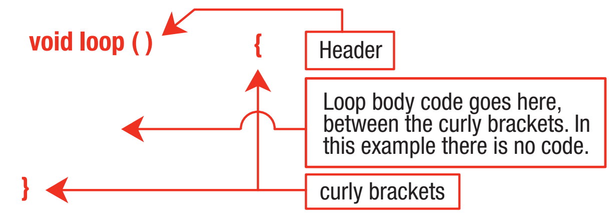
Name:
Date:

// Vocabulary: repetition, header, loop body, curly brackets

In computer programming **repetition** means repeating a portion of code. This can happen in a bunch of different ways, but the most important thing is to first understand how it happens, not all the different ways it can happen. There are really only two portions to any **repetition**, the **header** and the **loop body**. The **header** usually looks about the same, but the **loop body** can contain any kind of code depending on what you are programming. The **loop body** can even contain another **repetition**!

Repetition with the header, loop body, semicolons and curly brackets labeled:

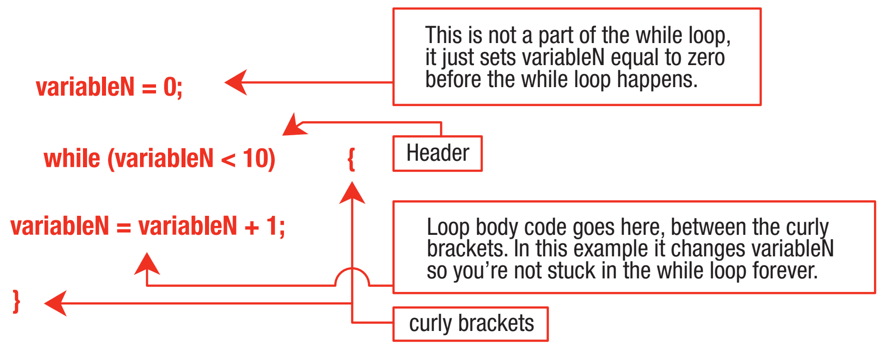
loop ( ):



```
void loop ( )          {



}
```

Header
Loop body code goes here, between the curly brackets. In this example there is no code.
curly brackets

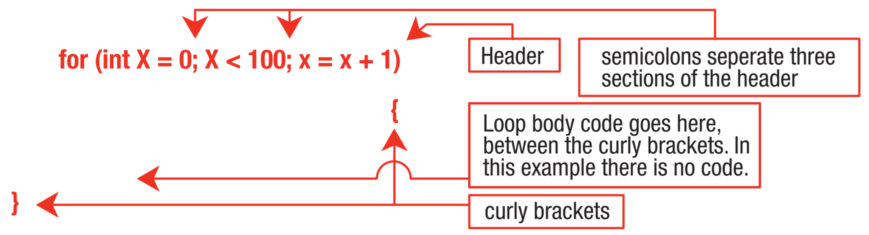
while ( ):



```
variableN = 0;

while (variableN < 10)        {

variableN = variableN + 1;

}
```

This is not a part of the while loop, it just sets variableN equal to zero before the while loop happens.
Header
Loop body code goes here, between the curly brackets. In this example it changes variableN so you're not stuck in the while loop forever.
curly brackets

for ( ):



```
for (int X = 0; X < 100; x = x + 1)
                                    {


}
```

Header
semicolons seperate three sections of the header
Loop body code goes here, between the curly brackets. In this example there is no code.
curly brackets

Just so we're clear on the important concepts that we will use when we talk about each different kind of repetition, please fill in definitions or explanations of the terms below.

Repetition: ____

Header: ____

Loop body: ____

Curly brackets: ____

Name:
Date:

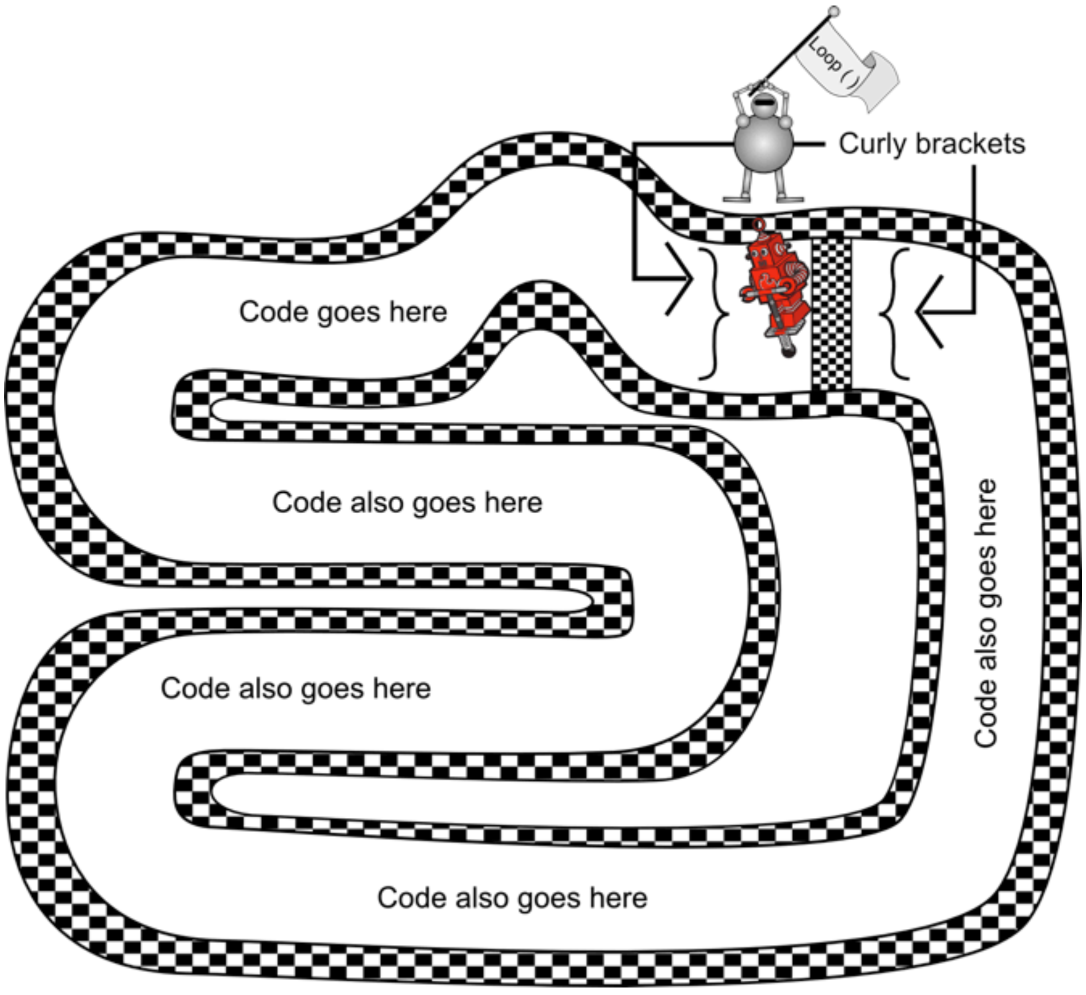
// Vocabulary: loop ( )

The most common form of iteration in Arduino is called the loop( ) function. It exists in all Arduino sketches and its whole purpose is to do all the code written inside of it once, then start over back at the beginning of the loop( ) function and do it all again. Pretty simple, right? The most important things to remember about the loop( ) function are that it is present in every single Arduino sketch, can only be used once per sketch, and it never ends. You will not find a single Arduino sketch that does not have a loop( )function in it and whenever anything happens in your sketch it is because of code inside the loop( ) function.

The loop( ) function looks like this:

```
void loop( ){
 // Lots (or just a little) of loop body code here between curly
brackets.
}
```

Pay attention to the header and the curly brackets which are at the beginning and end of the loop body code. The header is just void loop( ). Think of the loop( ) function as a racetrack. The loop( ) header portion is the flag that starts the computer going around the racetrack and the curly brackets are the beginning and end of the racetrack. Now imagine your computer, Arduino, or robot running around and around the racetrack. It's up to you, the programmer, to put If statements, variables and other code along the way around the racetrack.



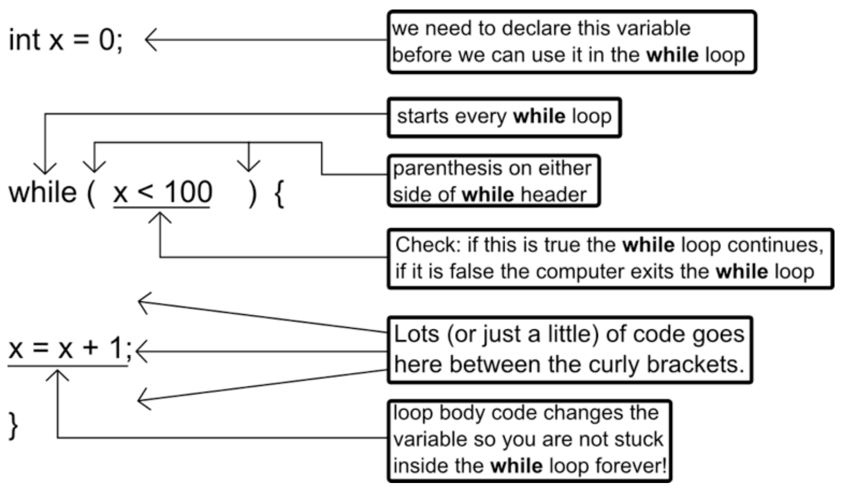
Name:
Date:

// Vocabulary: while, loop ( )

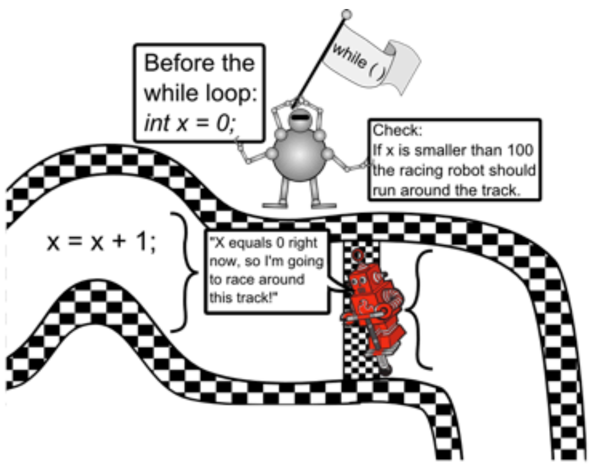
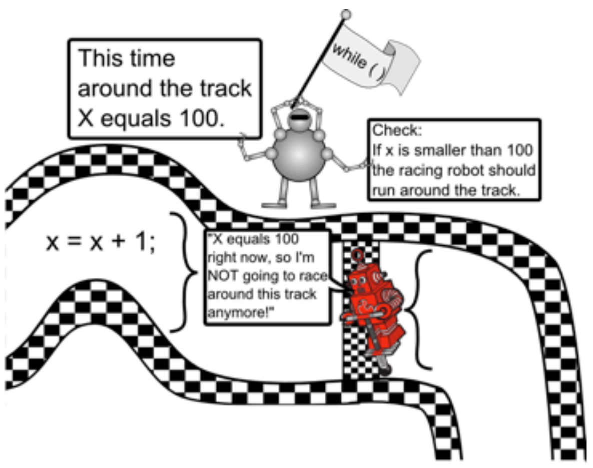
So, you just learned about loop( ), which is the simplest form of repetition, but there are many other forms of repetition in Arduino. Another very common form of repetition is the while loop. A while loop is used when you want the computer or Arduino to do some code while a statement is true. The while loop is usually found inside of the loop( ) function. The code of a while loop has two parts, the header and the loop body code. The header is the most important part to learn and always has the same structure. The code in the curly brackets below the header can be anything, it just depends on what you want to happen each time the computer goes around your while loop.

The header of a while loop has the word while and a statement inside of parenthesis. The while loop checks to see if the statement inside of the parenthesis is true and will repeat as long as that statement remains true. Pretty simple, right?

While loop example with variable declaration. Explanation of the while loop example.


int x = 0;
while (x < 100) {
x = x + 1;
}
we need to declare this variable before we can use it in the while loop
starts every while loop
parenthesis on either side of while header
Check: if this is true the while loop continues, if it is false the computer exits the while loop
Lots (or just a little) of code goes here between the curly brackets.
loop body code changes the variable so you are not stuck inside the while loop forever!

What happens during the while loop above using our robot racetrack as an example:

At the beginning:	Later on, after 100 laps:
	

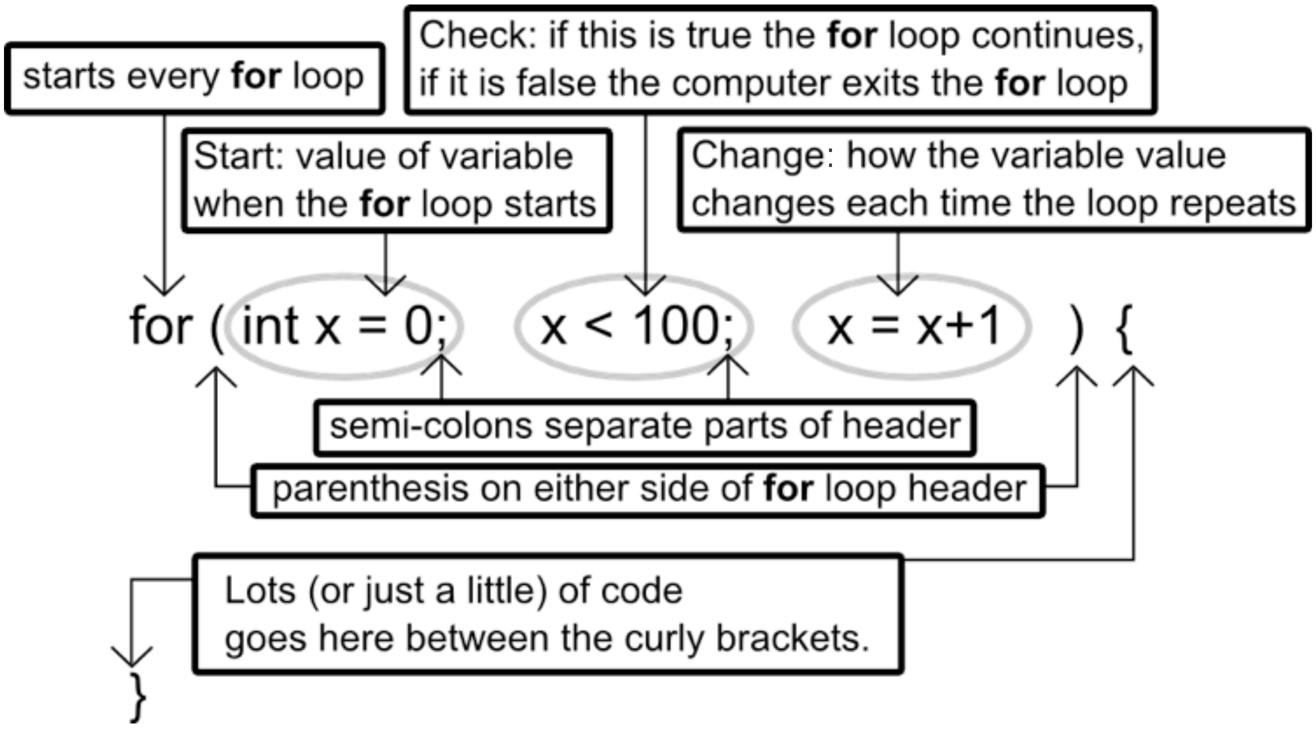
Before the while loop: int x = 0;
while ( )
Check: If x is smaller than 100 the racing robot should run around the track.
x = x + 1;
"X equals 0 right now, so I'm going to race around this track!"
This time around the track X equals 100.
while ( )
Check: If x is smaller than 100 the racing robot should run around the track.
x = x + 1;
"X equals 100 right now, so I'm NOT going to race around this track anymore!"

Name:
Date:

// Vocabulary: for, loop ( )

So, you just learned about while, which is a simple form of repetition, but there are many other loop functions. Another very common form of repetition is the for loop. A for loop is used when you want the computer or Arduino to change a variable each time through the loop and do code which often uses that variable. For loops are usually found inside of the loop( ) function. The code of a for loop has two parts, the header and the code inside the loop. The header is the most important part to learn and always looks about the same. The loop body code in the curly brackets below the header can be anything, it just depends on what you want to happen each time the computer goes around your for loop.

The header of a for loop has the word for and in parenthesis three parts called start, check and change. Each of these parts have semicolons between them so you can tell them apart. These three parts (circled in gray below) are the most important parts to understand, they are the three simple parts you need to make a for loop work.


starts every **for** loop
Check: if this is true the **for** loop continues, if it is false the computer exits the **for** loop
Start: value of variable when the **for** loop starts
Change: how the variable value changes each time the loop repeats
for (int x = 0; x < 100; x = x+1) {
semi-colons separate parts of header
parenthesis on either side of **for** loop header
Lots (or just a little) of code goes here between the curly brackets.
}

Start:
The first circled part is start, this happens before anything else, it's sort of like putting on running shoes before starting to run around the track. It is a simple declaration and assignment of a variable, in this case the variable is an integer named x.

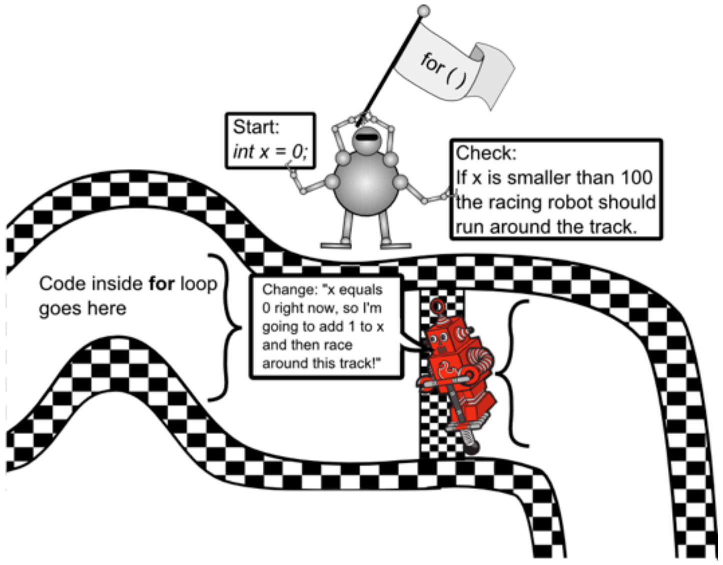
Check:
The second circled part is check. Every time the computer gets to the end of the for loop the computer will check to see if this part is true. The first time the for loop above checks, x is equal to zero, so the for loop continues, does change and then the code inside the curly brackets. It's kind of like checking how many laps a racer has completed to see if the racer has finished the race.

Change:
The third circled part is change, after the variable is checked it changes so that it is closer to making the check statement false so the for loop stops. For a racer this part of the for loop is like adding to (or updating) the number of laps or miles the racer has completed so far in the race.

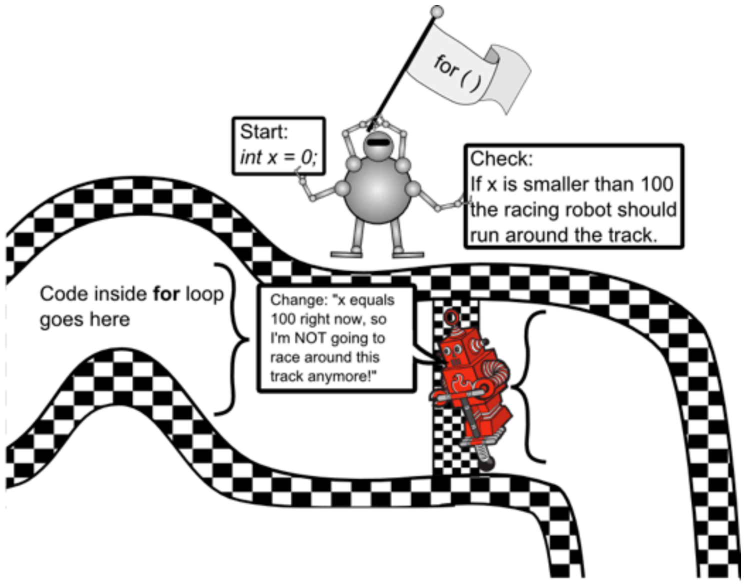
Name:
Date:

// Vocabulary: for, loop ( )

Here is an example of what happens when the for loop on the previous page begins using our robot racetrack as an example:


for ( )
Start:
int x = 0;
Check:
If x is smaller than 100 the racing robot should run around the track.
Code inside **for** loop goes here
Change: "x equals 0 right now, so I'm going to add 1 to x and then race around this track!"

The next time the racing robot makes its way around the track to the starting line it has to check again. It doesn't have to start again, but it does need to check to see if the race is over. The first time around the track, x will equal one and the check that x is smaller than 100 is still true. The robot changes the variable by adding one to x again (x now equals two) and then the robot runs around the track executing the loop body code between the curly brackets. The robot will continue to run around the racetrack until x equals 100 at which point the computer exits the for loop.


for ( )
Start:
int x = 0;
Check:
If x is smaller than 100 the racing robot should run around the track.
Code inside **for** loop goes here
Change: "x equals 100 right now, so I'm NOT going to race around this track anymore!"

Name:
Date:

// Vocabulary: Nested, Repetition

Now that you know about repetition we can talk about ways to put code inside of other code, which is called nesting, and in fact most loops are nested loops since they are inside of the original loop( ) function. It's easy, all you do is put your loop inside the curly brackets of another loop. Nested if statements work exactly the same way as nested loops.

Example of nested loop:

```
void loop ( ) {
int x = 0;
while (x < 10) {
x  =  x + 1;
}
}
```

Example of nested if statement:

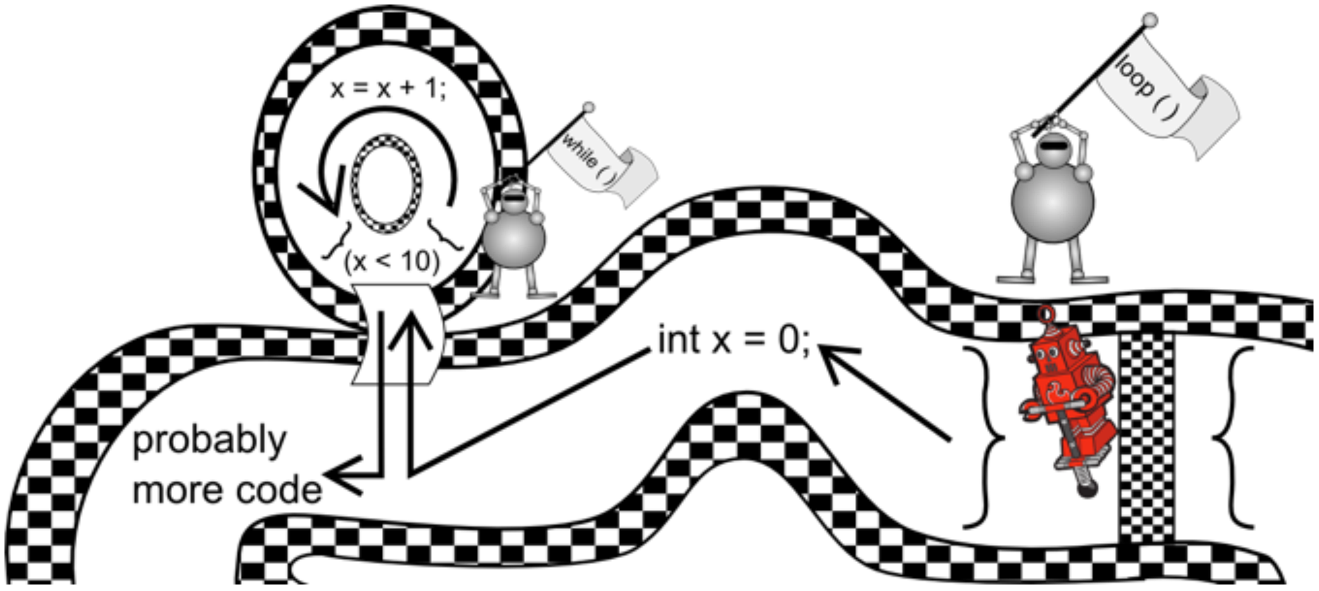
```
if (int x < 10) {
if ( x == 5 ) {
//code here happens if x < 10 & x = 5
}
//code here happens if x < 10
}
```

Imagine your loop( ) racetrack with another for loop racetrack attached to it. This way each time the robot runs (or drives or whatever) around the racetrack it must stop when it reaches a new while loop, run around that race track until that while loop is over and then it can continue running around the larger loop( ) racetrack.

The robot has to run through the whole while loop before it can continue running around the larger loop racetrack. But let's break it down a little more; x starts as zero, if x is less than ten the robot continues running around the while loop until x is not less than ten. If the robot is adding one to x each time it checks the while loop then the robot must run around the while loop a total of ten times. The robot then exits the while loop and continues around the loop racetrack. Next time around the racetrack the variable x will be set to zero again just before the while loop. So, you don't have to worry about the while loop not working due to x being more than or equal to ten.

You can nest as many loops inside of other loops as you like, just make sure you don't get stuck inside of a loop. One way to do this is to misplace curly brackets, so make sure they're in the right spot. If this happens your computer or Arduino will just freeze and you won't really be able to tell why.

Nesting works for code other than just loops! You can nest if statements, loops and many other code structures. All you need to remember is that nesting is a complicated way to say "put code inside of other code" and that the computer eventually needs to get out of the nested statements and back to the loop( ) function so everything can start over again.



Name:
Date:

Purpose: Group activity teaching the concept of repetition as used in Arduino programming. Text like this denotes actual Arduino code.

Materials: Cones, large boards to display loop headers and pseudo-code, equipment for physical activities, and a field or gym.

Vocabulary to be explained prior to activity:

loop or repetition:
A section of code that repeats.

repetition header:
The line at the very beginning of a loop that tells the computer how the code inside the loop will repeat. This section is different for each different type of loop.

Conditional or question:
This is the statement that is checked to see if the loop is completed. Conditionals are present in loop headers and often look like this: $x < 10$. This indicates that the loop will continue until $x < 10$ is false.

Increment:
The section of code (may be in the header or may be in the loop body code) used to change the variable that is checked in the conditional. Using the example above, $x = x + 1$, one is added to x getting it a little closer to being larger than or equal to 10.

Nested repetition:
A loop inside of a loop. This concept is key for any type of even slightly advanced programming.

Types of loops:

- **loop:**

This loop is the most basic of all loops (that's why it's called loop) and is present in all Arduino sketches. loop( ) repeats as long as there is power to the Arduino. Inside this form of repetition is where you will find all other forms of repetition.

Header:
loop( )

Increment:
N/A

Conditional:
Power must be on.

- **while:**

This loop repeats as long as the conditional listed inside the parenthesis is true. This loop's conditional is incremented in the body code or through an Arduino input.

Header:
while( )

Increment:
In body code or Arduino input

Conditional:
Inside header parenthesis.

- **for:**

This loop repeats as long as the conditional listed inside the parenthesis is true. The for loop header declares a variable, checks a conditional and increments the conditional variable all inside the parenthesis...

Header:
for (int x = 0; x < 10; x = x + 1)

Increment:
Inside header parenthesis, in this example $x = x + 1$.

Conditional:
Inside header parenthesis, in this example $x < 10$.

Activity
CHAPTER 3
Programming Concepts, Repetition

Name:
Date:

Preparation:
This activity is a physical activity and you will need to set up an obstacle loop or course that reflects the repetitions you have decided to include in this activity. You may wish to work with a gym teacher in order to set this activity up.

The examples in this activity require three different stations. These include a “loop” station at the beginning of the obstacle course with a teacher or student helper, a “while” station with jump ropes and an area for spinning in circles, and a “for” station with an area for doing jumping jacks and shooting basketballs.

Each station will need a poster displaying the pseudo-code that students need to follow in order to complete the obstacle loop. The poster materials are included with the rest of the activity materials in the folder programming in the file called LoopActivityMaterials.

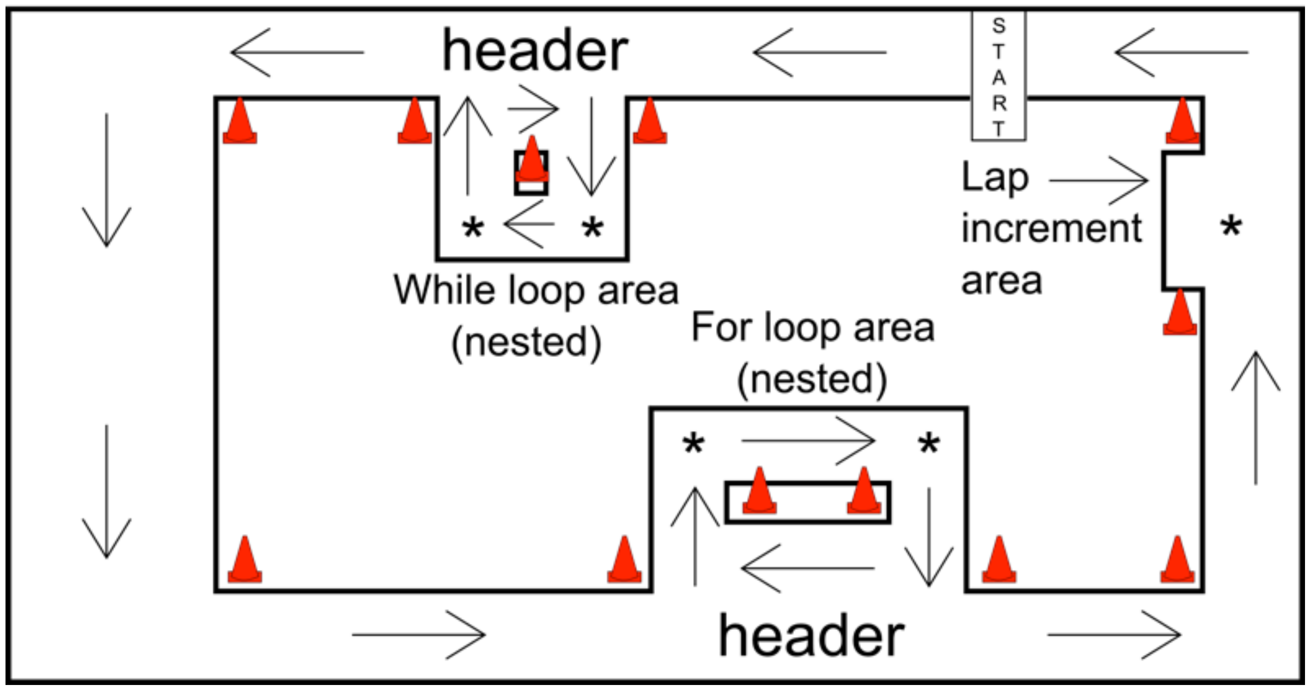
You will also need a field for kids to run around or cones to set up an area for kids to run around inside a gym.

Also- this is a really big activity. It takes a lot of prep and will probably be chaos the first time you try it, but it is easily customizable to age or skill level and should be lots of fun if you stick with it.

Activity:
Students should have completed the introduction to repetition worksheets that come with this activity. Students should also be familiar with variables and if statements.

What your loop activity might look like:

▲	Cones kids have to run around
→	Direction kids should be running
*	Areas with physical activities that kids complete
header	Instructions in pseudo-code form for above activities



Name:
Date:

The idea is that the complete obstacle course from the start position back to the start position represents the loop ( ) function. Inside this loop ( ) function are two nested loops, a while ( ) loop and a for ( ) loop.

At the beginning of the obstacle course each student needs to declare an integer variable called lapNumber or something similar. This variable will be used in each of the loop activity areas and the lap increment area. The lapNumber variable can also be used to end the obstacle course if you do not wish to have students run the obstacle course until the end of the period. When students start the obstacle course lapNumber should equal zero since they have not run any laps yet.

Nested loop activity areas: These areas are nested loops where students will perform a certain number of tasks depending on what your loop headers say. You can have as many or as few activity areas as you like. You may also tailor the number of physical tasks inside these nested loops to make your obstacle course more fun for your students.

These activity areas should look like little loops that the students can run around completing tasks. The headers should follow the format of the loop type it represents. For examples see the end of the activity. Once inside the nested loop activity area students must complete the physical activities according to the pseudo-code posted inside the nested loop activity area. Once students are done with the first repetition of the physical activities inside the nested loop activity areas they should look at the header again and decide if they have completed the nested loop represented by the header. With younger students you may want to have someone helping them with this step. (This can be fun, the observer can yell out error in a friendly voice if students exit the loop too quickly) Once students have completed the nested loop activity area they continue around the obstacle course to the next activity.

Header examples:

If a student's lapNumber is equal to three and the pseudo-code header reads:

```
while (lapNumber > basketsMade) {
do (lapNumber * 2) jump ropes at jump rope station
shoot lapNumber basketball baskets
}
```

This time around the obstacle course, the student would run through the nested loop activity area once, jumping rope six times and shooting three baskets along the way.

If a student's lapNumber is equal to three and the pseudo-code header reads:

```
for (int x = 0; x < lapNumber * 2; x = x + 1) {
do (x * 2) jump ropes at jump rope station
shoot lapNumber basketball baskets
}
```

The student would run through this nested loop activity six times jumping rope a different amount and shooting three baskets each time for a total of thirty six jump ropes and eighteen baskets.

There is a lot of room for personalization in this activity; it's an opportunity to really solidify the loop concept as well as getting your kids some exercise.

Activity

CHAPTER 3

Programming Concepts, Repetition

Name:

Date:

Lap increment area:
The lap increment area is where students will add one to their lapNumber variable to keep track of how many times they have run the obstacle course. You can also set up the headers and nested loop activities to use the lapNumber variable. The lap increment area is where you might insert an if statement to end the obstacle course after students complete a certain number of laps.

Additional thoughts:
Definitely call the obstacle course a loop( ) instead of an obstacle course in order to really get kids comfortable with the concepts. You may also wish to include your students in the planning of the obstacle course. Planning the obstacle course is another opportunity to talk about the loop concept and it gives them a stake in the learning exercise. Lastly, not that this needs pointing out, but this is a great activity just prior to computer lab time. Instead of having kids bouncing off the monitors they will be calmer and ready to sit still applying the concepts they just solidified through physical activity. This is great for kinesthetic learners in particular.

4

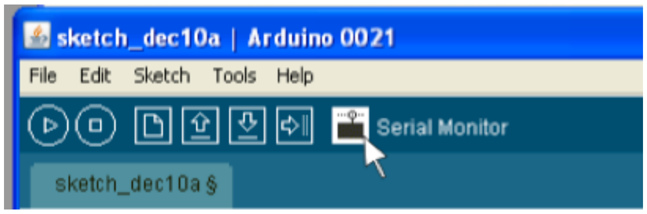
Serial

Name:
Date:

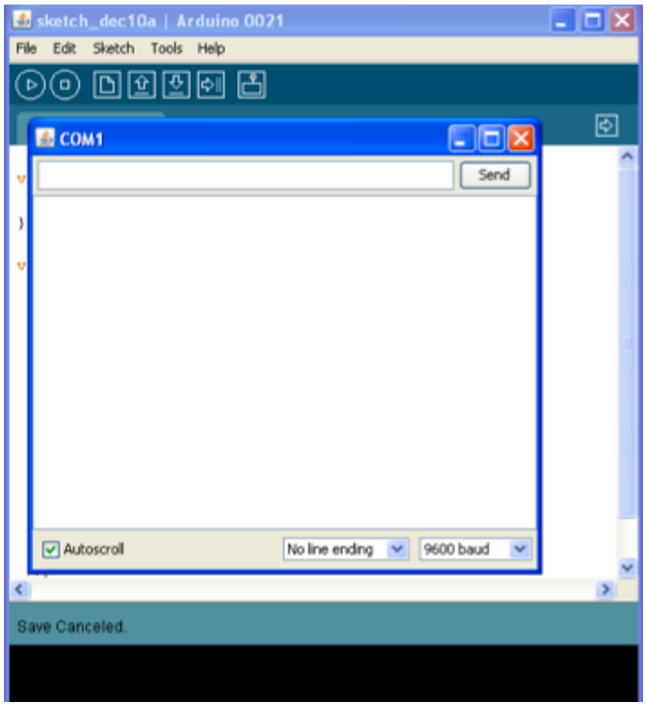
Serial is used to communicate between your computer and the RedBoard as well as between RedBoard boards and other devices. Serial uses a serial port also known as UART, which stands for universal asynchronous receiver/transmitter to transmit and receive information. In this case the computer outputs Serial Communication via USB while the RedBoard receives and transmits Serial using, you guessed it, the RX and TX pins. You use serial communication every time you upload code to your Arduino board. You will also use it to debug code and troubleshoot circuits. Basic serial communication is outlined in the following pages along with a simple activity to help you understand the concepts.

Serial Monitor:
This is where you monitor your serial communication and set baud rate.

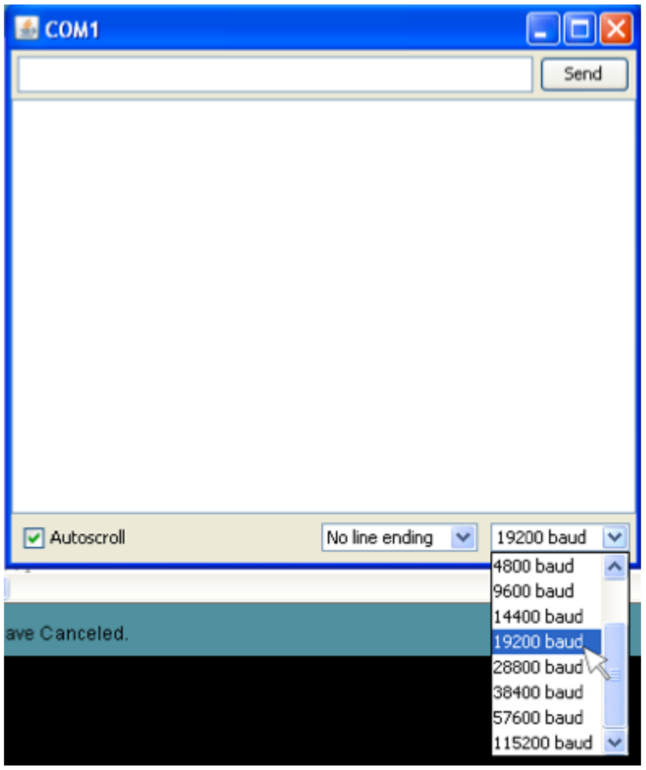
Activating the Serial Monitor:



What the activated Serial Monitor looks



Setting the Serial Monitor baud rate:



There are many different baud rates, (9600 is the standard for Arduino) the higher the baud rate the faster the machines are communicating.

In the examples above there is no Serial communication taking place yet. When you are running code that uses Serial any messages or information you tell Serial to display will show up in the window that opens when you activate the monitor.

Things to remember about Serial from this page:
1. Serial is used to communicate, debug and troubleshoot.
2. Serial baud rate is the rate at which the machines communicate.

Name:
Date:

Serial setup:
The first thing you need to know to use Serial with your Arduino code is Serial setup. To setup Serial you simply type the following line inside your setup( ) function:
Serial.begin (9600);

This line establishes that you are using the Digital Pins # 0 and # 1 for Serial communication. This means that you will not be able to use these pins as Input or Output because you are dedicating them to Serial communication. The number 9600 is the baud rate, this is the rate at which the computer and the Arduino communicate. You can change the baud rate depending on your needs but you need to make sure that the baud rate in your Serial setup and the baud rate on your Serial Monitor are the same. If your baud rates do not match up the Serial Monitor will display what appears to be gibberish, but is actually the correct communication incorrectly translated.

Using Serial for code debugging and circuit troubleshooting:
Once Serial is configured using the basic communication for debugging and troubleshooting is pretty easy. Anywhere in your sketch you wish the Arduino board to send a message type the line Serial.println("communication here");. This command will print whatever you type inside the quotation marks to the Serial Monitor followed by a return so that the next communication will print to the next line. If you wish to print something without the return use Serial.print("communication here");. To display the value of a variable using println simple remove the quotation marks and type the variable name inside the parenthesis. For example, type Serial.println(i); to display the value of the variable named i. This is useful in many different ways, if, for example, you wish to print some text followed by a variable or you want to display multiple variables before starting a new line in the Serial Monitor.

These lines are useful if you are trying to figure out what exactly your Arduino code is doing. Place a println command anywhere in the code, if the text in the println command shows up in your Serial Monitor you will know exactly when the Arduino reached that portion of code, if the text does not show up in the Serial Monitor you know that portion of code never executed and you need to rewrite.

To use Serial to troubleshoot a circuit use the println command just after reading an input or changing an output. This way you can print the value of a pin signal. For example, type Serial.print("Analog pin 0 reads:"); and Serial.println(analogRead(A0)); to display the signal on Analog Input Pin # 0. Replace the second portion with Serial.println(digitalRead(10)); to display the signal on Digital Pin # 10.

Things to remember about Serial from this page:
1. If Serial is displaying gibberish check the baud rates.
2. Use Serial.print("communication here"); to display text.
3. Use Serial.println("communication here"); to display text and start a new line.
4. Use Serial.print(variableName); to display the value stored in variableName.
5. Use Serial.print(digitalRead(10)); to display the state of Digital Pin # 10.

Name:
Date:

Using Serial for communication:
This is definitely beyond the scope of the S.I.K. but here are some basics for using Serial for device to device communication (other than your computer), not just debugging or troubleshooting. (The following paragraphs assume that you have Serial Communication hardware properly connected and powered on two different devices.)

First set up Serial as outlined on the previous page.

Use Serial.println("Outgoing communication here"); to send information out on the transmit line.

When receiving communication the Serial commands get a little more complicated. First you need to tell the RedBoard to listen for incoming communication. To do this you use the command Serial.available();, this command tells the computer how many bytes have been sent to the receive pin and are available for reading. The Serial receive buffer (computer speak for a temporary information storage space) can hold up to 128 bytes of information.

Once the RedBoard knows that there is information available in the Serial receive buffer you can assign that information to a variable and then use the value of that variable to execute code. For example to assign the information in the Serial receive buffer to the variable incomingByte type the line; incomingByte = Serial.read(); Serial.read() will only read the first available byte in the Serial receive buffer, so either use one byte communications or study up on parsing and string variable types. Below is an example of code that might be used to receive Serial communication at a baud rate of 9600.

```
//declare the variable incomingByte and assign it the value 0.
int incomingByte = 0;

void setup ( ) {
//establish serial communication at a baud rate of 9600
        Serial.begin(9600);
}
void loop ( ) {
//if there is information in the Serial receive buffer
        if (Serial.available() > 0){
//assign the first byte in buffer to incomingByte
                incomingByte = Serial.read();
        }
        if (incomingByte == 'A'){     //if incomingByte is A
                //execute code inside these brackets if
incomingByte is A
        }
        if (incomingByte == 'B'){     //if incomingByte is B
                //execute code inside these brackets if
incomingByte is B
        }
}
```

Additional things to note about Serial:
You cannot transmit and receive at the same time using Serial, you must do one or the other. You cannot hook more than two devices up to the same Serial line. In order to communicate between more than two devices you will need to use an Arduino library such as NewSoftSerial.

Things to remember about Serial from this page:
1. Serial communication requires knowing some code, but you can just look it up!
2. You cannot transmit and receive at the same time or hook up more than two devices.

Name:
Date:

// Activity

Practicing simple Serial for debugging code:
If you would like to practice using Serial for code debugging open the code file Serial01, copy the text and paste it into an Arduino sketch. This sketch is mainly empty and waiting for you to add the Serial commands.

1. First type in the command line that begins Serial at a baud rate of your choosing below where the comment reads “place serial setup here”.

2. Next open the Serial Monitor and make sure it matches the baud rate you chose. This is an example of setting up Serial communication.

3. Next type a single Serial command that will display the text “Loop starts here” below the comment “place serial statement 1 here”. Make sure the command you use starts a new line after this text is displayed.

4. Then type a single Serial command that will display the text “Variable i is equal to “ below the comment “place serial statement 2 here”. Make sure you use the command that does not start a new line after this text is displayed.

5. Now add a command below this that will display the variable i. If you are having trouble with this portion don’t forget that only text needs quotation marks around it for display in the Serial Monitor. This is an example of how you can use Serial communication to label your communication when you are trying to debug a troublesome variable.

6. Below the comment “place serial statement 3 here” add a command that will display the text “this text displays when i is equal to 8” and then start a new line. This is an example of how to display a variable value for debugging.

7. Below the comment “place serial statement 4 here” add a command that will display the text “this text displays when i is equal to 9” and then start a new line. This is an example of using Serial communication see if a portion of code ever actually executes.

8. Below the comment “place serial statement 5 here” add a command that will display the text “Loop ends here” and then start a new line.

Name:
Date:

// Activity

Using Serial for communication:
This is definitely beyond the scope of the S.I.K. but here are some basics for using Serial for device to device communication (other than your computer), not just debugging or troubleshooting. (The following paragraphs assume that you have Serial Communication hardware properly connected and powered on two different devices.)

First set up Serial as outlined on the previous page.

Use Serial.println("Outgoing communication here"); to send information out on the transmit line.

When receiving communication the Serial commands get a little more complicated. First you need to tell the Arduino to listen for incoming communication. To do this you use the command Serial.available();, this command tells the computer how many bytes have been sent to the receive pin and are available for reading. The Serial receive buffer (computer speak for a temporary information storage space) can hold up to 128 bytes of information.

Once the RedBoard knows that there is information available in the Serial receive buffer you can assign that information to a variable and then use the value of that variable to execute code. For example to assign the information in the Serial receive buffer to the variable incomingByte type the line; incomingByte = Serial.read(); Serial.read() will only read the first available byte in the Serial receive buffer, so either use one byte communications or study up on parsing and string variable types. Below is an example of code that might be used to receive Serial communication at a baud rate of 9600.

5

Logic Flow/Schematics

//Logic Flow Charts (part 1)

Logic Flow Charts are a great way to sketch out how you want a circuit or chunk of code to act once it is completed. This way you can figure out how the whole project will act without getting distracted or confused by little details like electricity or programming. It's kind of like a game plan that a coach will put together before a game.

There are four major pieces that you will use over and over again when creating Logic Flow Charts. The four Logic Flow pieces are represented by a circle, a square, a diamond and lines connecting all the circles, squares and diamonds.

The **circle** is used to represent either a starting point, or a stopping point. This is easy to remember since you start every single Logic Flow Chart with a circle containing the word Start or Begin. Often you will end a Logic Flow Chart with an End or Finish circle, but sometimes there is no end to the chart and it simply begins again. This is the case with any circuits that never turn off, but are always on and collecting data.

The **square** is used to represent any action which has only one outcome. For example, when a video game console is turned on it always checks to see what video game is in it. It does this every time after it starts up and it never checks in a different way. This kind of action is represented by the square, it never changes and there is always only one outcome.

The **diamond** is used to represent a question or actions with more than one possible outcome. For example, once your video game has loaded there is often a menu with a bunch of options. This would be written in a Logic Flow Chart as a diamond with something like the words "Start Up Menu" written inside of it. Each action the user can take from this menu would be represented by lines coming off the diamond leading to another square, diamond, or circle. Maybe our example Logic Flow Chart would have three options leading away from the "Start Up Menu" diamond, one line to start a new game, one to continue a saved game and another for game settings. In the Logic Flow Chart each option is written beside the line leading away from the diamond. It is possible to have as many options as you like leading away from a diamond in a Logic Flow Chart.

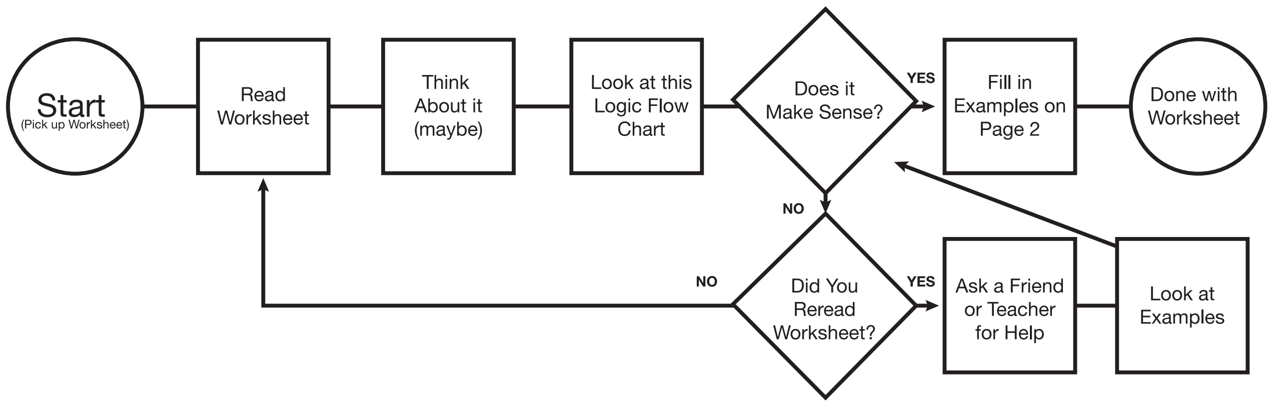
The **lines** in a Logic Flow Chart connect all the different pieces. These are there so the reader knows how to follow the Logic Flow Chart. The lines often have arrows on them and lead to whichever piece (circle, square, diamond) makes the most sense next. The lines usually have explanation of what has happened when they lead away from diamonds, so the reader knows which one to follow. Often some of these lines will run to a point closer to the beginning of the Logic Flow Chart. For example, the "Save Game" option might lead back to the "Start Up Menu" diamond, or it might lead straight to "Save and Quit." It's up to you, you're the one making the Logic Flow Chart! All it has to do is make sense to you. Use the first Logic Flow Chart on the next page to help figure out how to use a Logic Flow Chart. Look at the second example, then complete the remaining Logic Flow Chart examples.

Name:
Date:

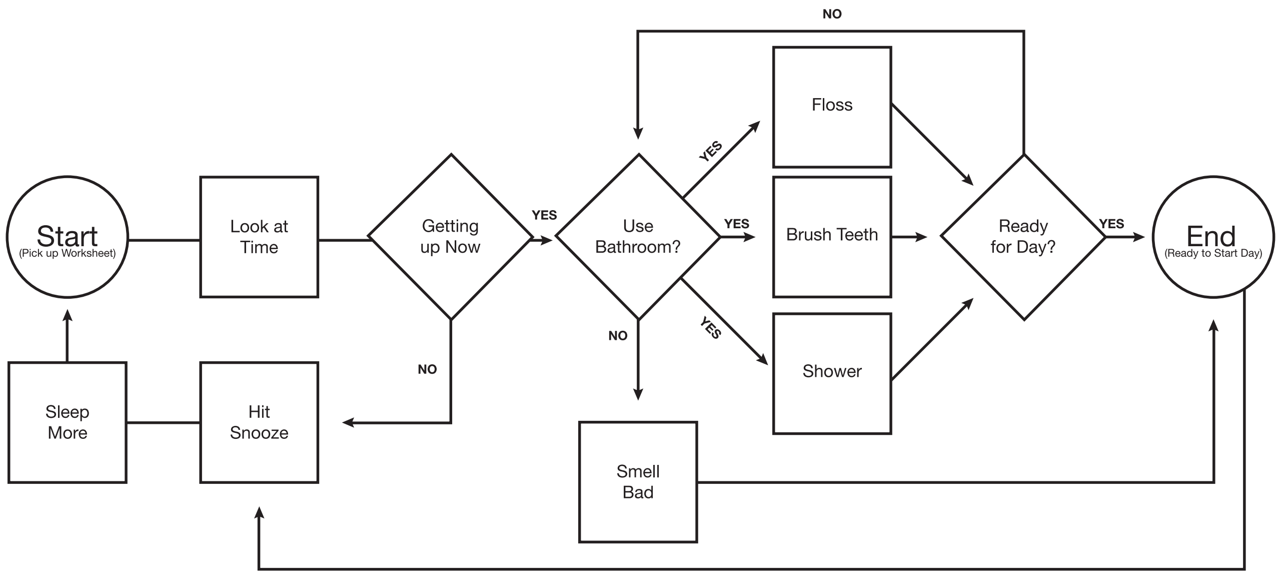
// Logic Flow Charts (part 2)

**Circles represent either start or end. Squares represent actions with one outcome. Diamonds represent a question or action with multiple possible outcomes.
Lines and arrows represent logical paths between the circles, squares and diamonds.**

Example 1:



Example 2:

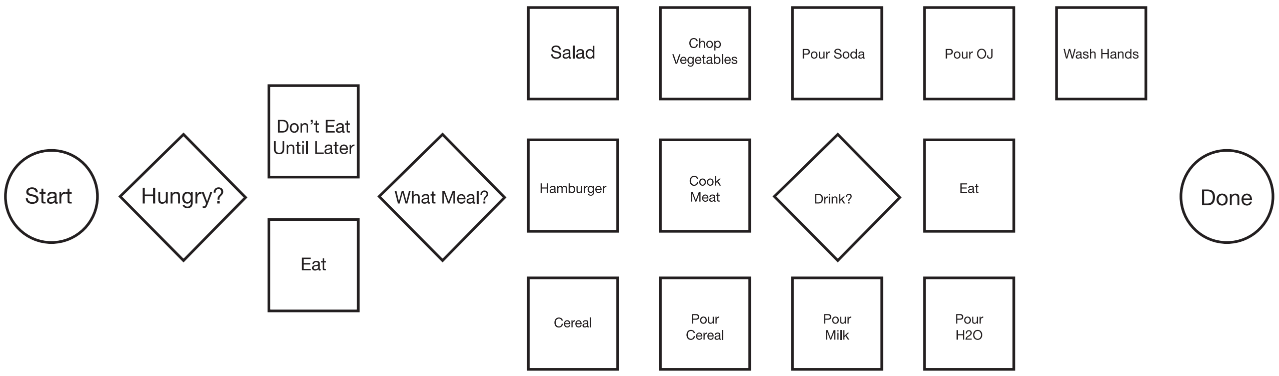


Name:
Date:

// Logic Flow Charts (part 3)

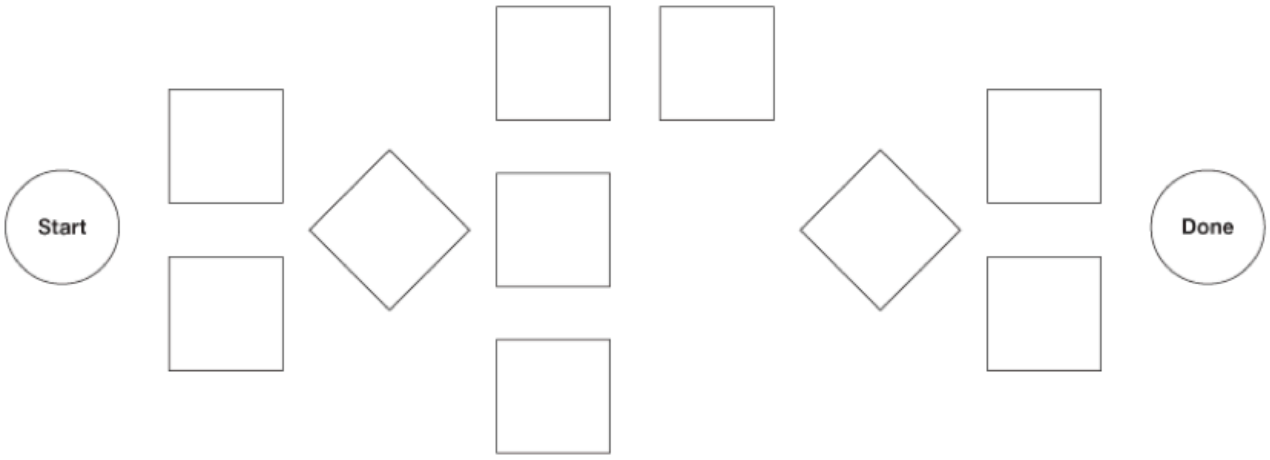
Example 3:

Fill in the lines and arrows. There is no right answer, but it must make sense.



Example 4:

Fill in the lines and text. Write outside of the boxes as necessary or use the back of the worksheet.

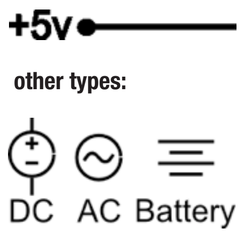


Name:
Date:

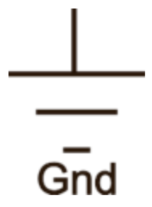
// Schematics

An electrical schematic is a bunch of symbols representing one or more electrical circuits. Electrical schematics are a great way to sketch out the physical layout of a circuit. With a schematic it is possible to share circuit and prototypes ideas without giving away your electronics or creating an overlay. Being able to read schematics is definitely a useful skill anytime electronics are involved. Once you are able to read schematics, creating schematics just takes practice and a sharp eye when looking at wires and components.

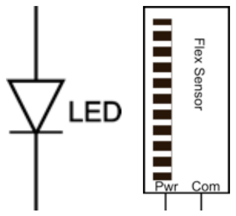
There are four major pieces that you will use over and over again when creating schematics. The four schematic pieces are power source, ground, components and the wire (or whatever your conductive connecting material is) connecting all the different parts. While schematics can be created by hand, these days electrical schematics are usually created using Electrical CAD software. Electrical CAD software is used to make sure that all schematics follow the same guidelines, making them easier to understand. While electrical schematics show how circuits are connected, they do not show what the completed circuit will look like. Schematics are guidelines, not physical representations of the circuits.



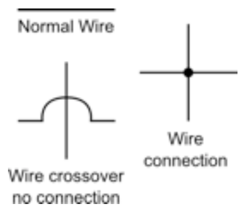
The power source symbol is a small circle with the voltage written beside it. Power sources come in all sizes, but the S.I.K. mainly uses a 5V power source. Arduino output pins can also be used as power sources, so even though they are categorized as components (or at least a part of a component, the Arduino) they are often treated as power sources. Power sources are where the electricity necessary to make circuits work comes from.



The ground symbol used in the S.I.K. is three horizontal lines, which decrease in width as they get closer to the bottom of the symbol with the letters Gnd beside or below them. In an electrical schematic ground can have a couple different types, this worksheet explains the most common, earth ground. An earth ground is a return path for electrical current as well as a reference point for measuring voltage. The term "earth ground" implies that the ground is a physical connection to the earth. This is sometimes true, but often ground is simply a connection to the lowest voltage value in a circuit or piece of equipment. The voltage value of ground will never change no matter how much electrical current it is absorbing.



Components are represented by a bunch of different symbols. There are tons of different components with new types being invented every day! These are the parts of the circuit that use the electrical current to make stuff happen. This can be input or output, digital or analog, complicated (Arduino board) or very simple (resistor). These components can do many different things and it is important to understand the particular component in the schematic if you really want to know what the circuit is doing and how it uses electrical current.

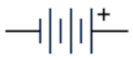
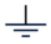
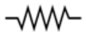
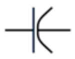
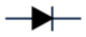
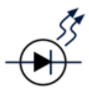
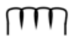
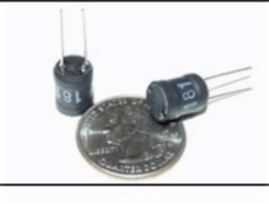
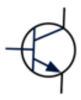
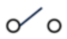


Wires are represented by simple lines. The wires in a schematic connect all the different pieces. Pay attention to what wires look like in a schematic when they cross. If the wires are connected the lines will be straight with a circle where they cross, if the wires are not connected one of the wires will form a semi-circle where the lines cross.

Name:
Date:

// Common Schematic Symbols

Common Schematic Symbols

Power / Battery / Cell			
Ground			
Resistor (Colored bands indicate resistive strength - see cheatsheet for details)			
Capacitor (Electrolytic on left; Ceramic on right)			
Diode			
Light Emitting Diode (LED)			
Inductor			
Transistor (arrow pointing out is NPN, pointing in is PNP)			
Switch			
Speaker