sparkfun™ ELECTRONICS

WEBSITE: sparkfun.com
6175 LONGBOW DRIVE, SUITE 200
BOULDER. COLORADO USA
ZIP CODE: 80301
[303] 284.0979 [GENERAL] P
443.0048 F

Summer Semester Overview
SparkFun Electronics Summer Semester

Overview of Summer Semester Class

Day One:

Napkin Schematics
Datasheets and Catalogs
Intro to Arduino using SIK

A brief explanation of the prototype planning process, discussion of student's ideas for projects and an Introduction to Arduino

Day Two:

Soldering Simon (PTH or SMD)
Stenciling
Lilypad Discussion with take home kit
Etcha-A-Simon Processing

Learn to solder by putting together a Simon and Arduino Pro, then learn how to interface with Processing

Day Three:

SIK extended with Programming Discussion
Eagle

More in depth use of the SIK and PCB layout using Eagle

Day Four:

Wireless Overview
XBee Activities
GPS Overview
GPS Activity

Learn how to use XBee series 2 and the EM406 GPS unit, hope for good weather cause we'll be going outside to test

Day Five:

Independent Pursuit until 5 pm

With Engineer Support

?

Table of Contents

Prototyping Tricks and Tips

Here are some tips and tricks that will help anyone, no matter what their level of expertise is, when prototyping, debugging or just plain old fixing a Physical Computing project. Some of them are from Nate's experience working at a Kid's Museum called World Of Wonder, other tips and tricks where gathered from other people at SparkFun. Got a tip or trick that you think would be useful to others? Let us know and we'll put it in this document.

1) Document your work. Realize you won't be around next month so please leave behind everything you can to help the next person support (repair) your work. Things I now leave inside the actual exhibit: clear schematics, a wiring diagram, an overall layout, all the firmware on a jump drive, operation instructions and maintenance information for the support staff, and contact details (my name, my email, cell # if needed, etc).

2) Use polarized connectors. This may seem silly but I've seen a lot of projects where the creator soldered wires point to point. This means that the wires were soldered directly onto the connections on the board. Not only is this slow, it is unmaintainable. You should *always* be able to remove a board or sensor by unplugging the various wires. This makes it possible to swap out parts to determine what might be broken, increasing the speed of debugging. Furthermore, you should always use polarized connectors. If the staff needs to relocate a piece to a different part of the museum, asking them to decipher and follow a thin sharpie mark (or even worse labeling) on a perf board is dangerous. Polarized connectors will help prevent your piece from sitting unused in the back room.

3) Buy pre-made cables. This goes hand in hand with the polarized connector lesson. You should not be running down gremlins (aka problems) in your wiring harness. Use premade cables that you can trust. I personally love to use CAT5 cable: it's cheap, readily available in many different lengths, and you can even get different colors! If multiple of the same type of cable run into a board then throw the cable color ("Red") or cable name ("IR2") on the PCB for easy identification. I also use the SparkFun JST assemblies by the handful. Buying these pre-made cables and connectors allowed me to concentrate on bigger problems than getting my crimps correct.

4) Label stuff. On a complex piece there will be large amounts of wires running everywhere. A $30 label printer will save your sanity and the person who looks at the piece after you. Label *everything*. This is obviously a USB cable, but what does it go to? **Main controller debug 38400bps** would be a great label. **"IR LED 4"**, **"Red to Detectors 1-4"**, **"To Light Controller"** are all great cable labels.

5) Leave the debug cables in the unit. Illumitune (piece at World Of Wonder) had a total of 4 microcontrollers. At any given time I needed to reconfigure two of them regularly. I had the idea of running USB cables from the boards down to the side access panel. I left these USB cables permanently installed in the exhibit. This was awesome! It saved me from having to climb a ladder and plug in a USB cable every time I needed to debug the system. By leaving the 'debug cable' in your exhibit you also increase the odds the next person to maintain your exhibit will be able to connect to it. Please don't assume that next user will be able to connect to the three odd spots on your perf board to get to TX/RX/GND. By leaving a standard USB A-to-B cable in the exhibit I assume that USB ports will be supported for the next 3-5 years.

6) Give me some debug information. Make the debug port obvious. I should be able to walk up to a piece and within a few minutes be able to plug in a netbook and see something coming out. Ideally the output would tell me what I'm looking at and looking for. For example, when you power up Illumitune, it states 'IR Controller Online' then 'Light Controller Online' then it displays '$3#' when beam 3 is broken.

7) Own a logic analyzer. I wasted about 8 hours of painful troubleshooting because my IR timing was off. 15 minutes after hooking up my logic analyzer and it was obvious what was wrong and I had a solution in place. I usually do all of my debugging with print statements. This works for much of the time, but for the situations where one is needed (troubleshooting I2C, SPI, or carefully timed procedures like IR transmission) a logic analyzer is worth its weight in platinum. No really, get one. Your sanity will thank me later.

8. Double check your Serial lines. This is the number one error every engineer at SparkFun occasionally makes. It's silly and it's easy to do even after years of experience. Luckily it's easy to check as well.

9. Work with a breadboard first. There's a reason why engineers use breadboards before embedding. With breadboards you can see if your circuit works before moving on to the step of embedding or hooking up an interface. Breadboards are your friends.

10. Don't be afraid to just plug it in. Stop worrying and see if it works. That said, we highly recommend using PTC's for basic breadboard prototyping. If it worked on the breadboard it should work when you plug it in.

11. Order twice what you think you need of the cheap stuff. You're going to want to make a second prototype, you're going to burn some materials, and you're going to measure incorrectly. Why not save yourself another trip to the hardware store by buying a little extra on the first trip?

12. Don't be afraid of the basics. Often times the most useful and innovative designs and devices are built off of the solid foundations of previous technology. By all means don't hesitate to make those creative leaps and operate without a rope so to speak, if you feel comfortable doing so. But be aware that if you're trying something completely new and going for pure innovation, there's nothing to double-check, use as a marker for truth, or sanity check when something goes wrong. The problem with exploring completely new territory is that if you get lost, there's no easy way to find your way back home. This is why it's always good practice to utilize those basic designs that have been rehashed a thousand times. Building a good percentage of your project off of tried and true technology gives you a compass for troubleshooting your design should something malfunction, eliminating the guesswork, and isolating the problem to a much smaller set of probable issues. The best engineers will tell you that drilling the basics is the most important and useful way to ensure that your next crazy, new idea gets off the ground.

13. Don't use ferric chloride for etching your own circuit boards. Use hydrogen peroxide mixed with muriatic acid (both available from home depot). If you have access to a laser engraver, you can make decent double-sided prototype boards by coating the copper clad board with flat black spray paint and then etching away the paint around your circuit traces. Then just etch the board using muriatic acid and H2O2.

14. Double check your PCB files. When troubleshooting your PCB try to catch as many errors as you can instead of just fixing one error and waiting for the PCB to come back so you can test it and find another error. If you try this approach you will save yourself a lot of time as well as PCB cost. In order to catch all the errors first troubleshoot do whatever it takes to get the first run PCB working with your hardware. Green wire fix everything that doesn’t work and take note so you can adjust in your Gerber files. Also assume the manufacturer is going to misprint your board a couple of times you order PCBs.

15. Less moving parts is better. Try to find simpler, more elegant solutions for things like physical interfaces and mechanical aspects of your prototypes. Less moving parts means less stuff that could conceivably break, makes sense right?

16. And finally, even though you don’t want to… take a break. No really, take a break, eat some food, take a walk, take a nap, or if you refuse to walk away at least work on a different portion of the project. Save yourself some frustration, because things will seem so much clearer at 3 pm than at 3 am.

Napkin Schematics

SparkFun Electronics Summer Semester

1. Short Description

Write a brief description of your project here. List inputs and outputs, existing systems it will integrate with and any other notes that occur to you. Don't spend too long on this section.

2. Sketch

Sketch an image of what you imagine your project or system to look like here.

3. Block Diagrams

Draw a diagram where each of the components in your project is represented by a simple square with lines connecting the components that will be connected. Don't worry about getting all the connections perfect; what's important is that you're thinking about all the different components and connections. Be sure to include things like power sources, antennas, buttons or other interface components and always include at least one LED to indicate the system is on. Although you'll probably want more LEDs than just the one, they make troubleshooting and debugging easier.

4. Logic Flow

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 by details like electricity or programming.

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

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 that 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. Lines coming off the diamond leading to another square, diamond, or circle would represent each action the user can take from this menu. 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; all it has to do is make sense to you.

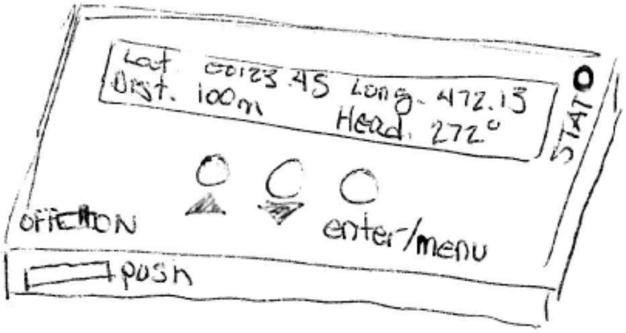
Napkin Schematics

SparkFun Electronics Summer Semester

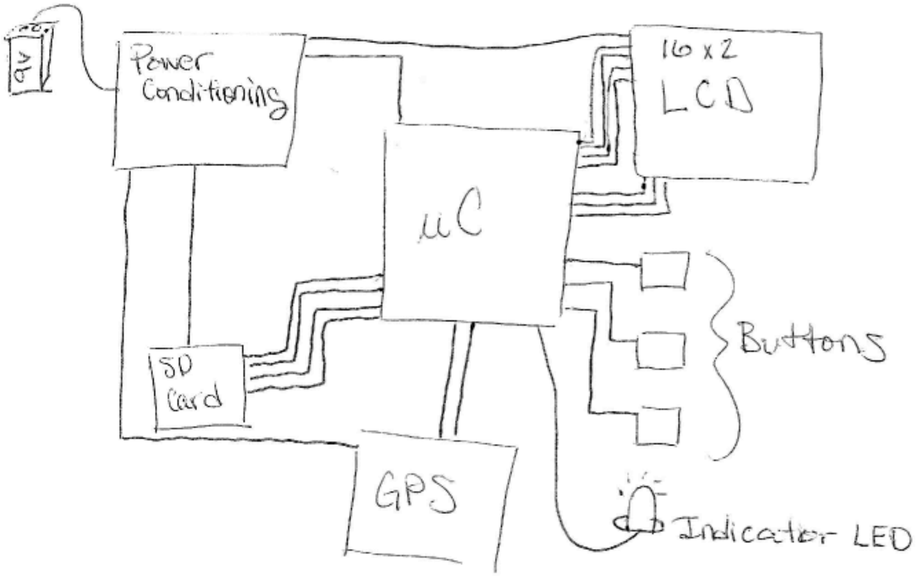
1. Short Description (Example)

A GPS enable device that provides heading and distance information to a preconfigured latitude and longitude via a character LED display. User would be able to select from a GPS screen; current coordinates, time and altitude. “Destination” coordinates would be stored on an SD card.

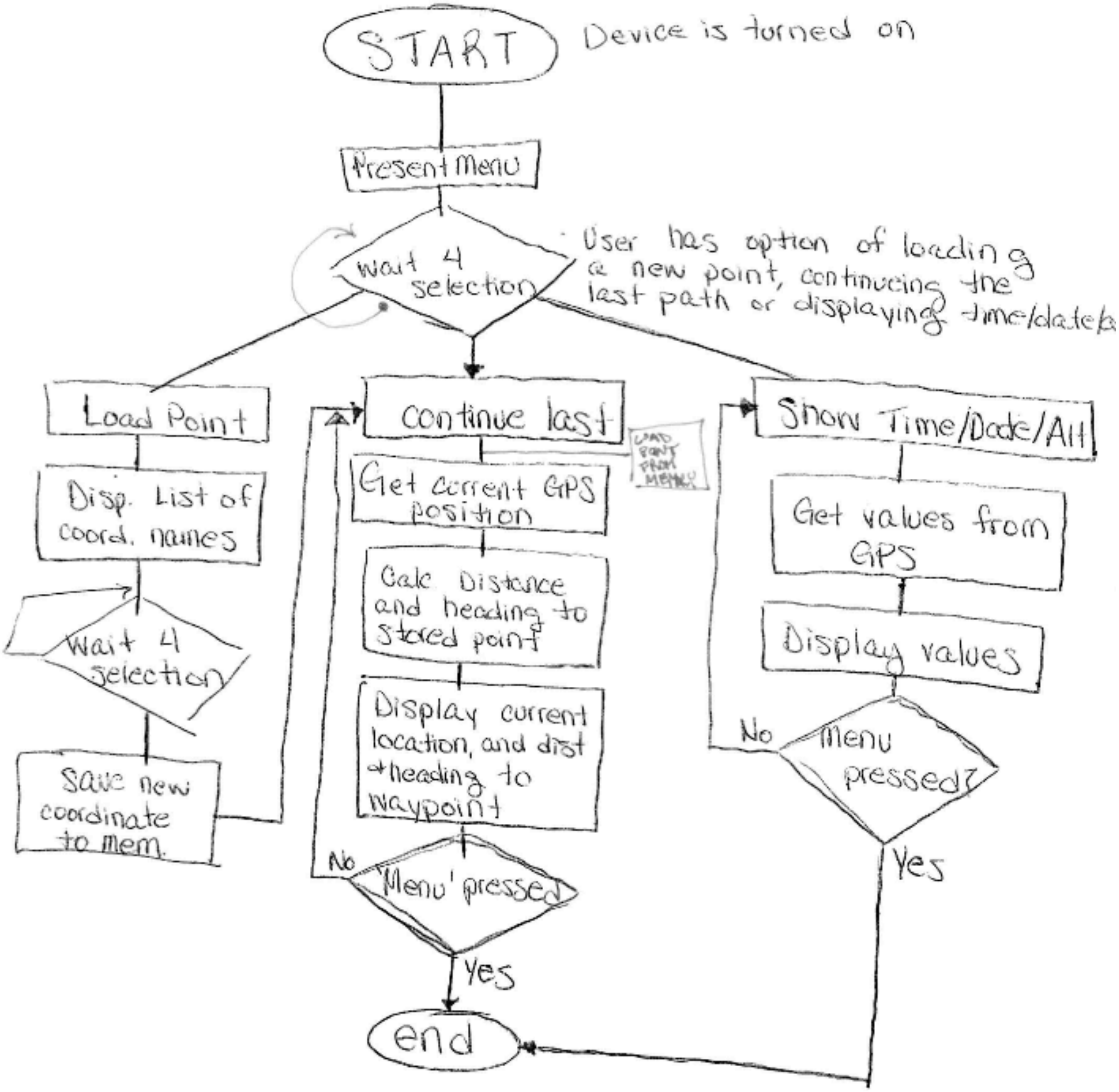
2. Sketch (Example)



3. Block Diagrams (Example)



4. Logic Flow (Example)


START
Device is turned on
Present Menu
wait 4 selection
User has option of loading a new point, continucing the last path or displaying time/date/a
Load Point
Disp. List of Coord. names
Wait 4 selection
Save new coordinate to mem.
continue last
LOAD POINT FROM MEMORY
Get current GPS position
Calc Distance and heading to stored point
Display current location, and dist +heading to waypoint
No
'Menu' pressed
Yes
Show Time/Dode/Alt
Get values from GPS
Display values
No
Menu pressed?
Yes
end

Napkin Schematics

SparkFun Electronics Summer Semester

1. Short Description

2. Sketch

3. Block Diagrams

4. Logic Flow

Napkin Schematics

SparkFun Electronics Summer Semester

1. Short Description

2. Sketch

3. Block Diagrams

4. Logic Flow

Napkin Schematics

SparkFun Electronics Summer Semester

1. Short Description

2. Sketch

3. Block Diagrams

4. Logic Flow

Part Procurement Sources

SparkFun Electronics Summer Semester

SparkFun Electronics
www.sparkfun.com

Mouser Electronics
www.mouser.com

Digi-Key Corporation
www.digikey.com

Jameco Electronics
www.jameco.com

Solarbotics
www.solarbotics.com

Pololu Robotics and Electronics
www.pololu.com

Futurlec
http://www.futurlec.com

Future Electronics
http://www.futureelectronics.com

Ladyada
http://www.ladyada.net

Maker Shed
http://www.makershed.com

Alibaba
http://www.alibaba.com

Find Chips
http://www.findchips.com

sparkfun ELECTRONICS

Intro to Arduino

Zero to Virtual Prototyping in Seven Hours

Linz Craig, Erik Hallqvist, Jordan McConnell,
Dave Stillman and Aaron Weiss

Overview of Class

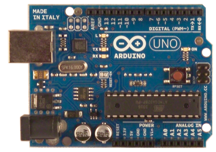
Getting Started:
Installation, Applications and Materials
Electrical:
Components, Ohm's Law, Input and Output, Analog and Digital

One Hour Break For Lunch

Programming:
Split into groups depending on experience
Serial Communication Basics:
Troubleshooting and Debugging
Virtual Prototyping:
Schematics and PCB Layout in Fritzing

Arduino Board

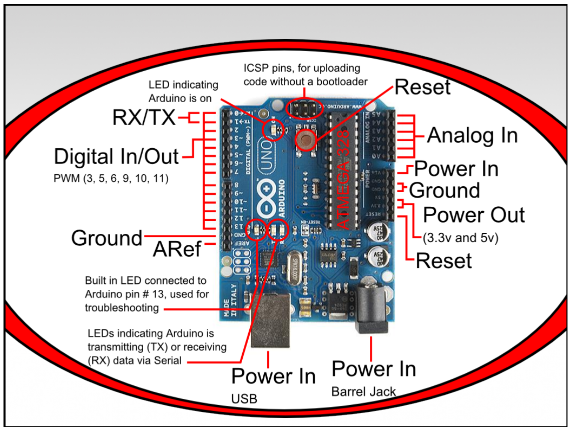
"Strong Friend" Created in Ivrea, Italy
in 2005 by Massimo Banzi & David Cuartielles
Open Source Hardware
Atmel Processor
Coding is accessible (C++, Processing, ModKit and MiniBloq)



Team Arduino: (Left to right)
David Cuartielles, Massimo Banzi & Gianluca Martino

Getting Started

- **Installation:** Arduino (v.22)
  Java and Drivers
- **Materials:** SIK Guide
  Analog I/O, Digital I/O, Serial,
- **Applications:** Arduino IDE for programming board
  Processing



ICSP pins, for uploading code without a bootloader

LED indicating Arduino is on

RX/TX

Reset

Analog In

Digital In/Out
PWM (3, 5, 6, 9, 10, 11)

Power In

Ground

Power Out
(3.3v and 5v)

Ground

ARef

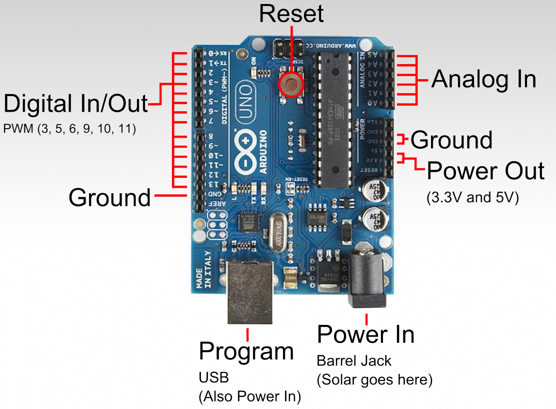
Reset

Built in LED connected to Arduino pin # 13, used for troubleshooting

LEDs indicating Arduino is transmitting (TX) or receiving (RX) data via Serial

Power In
USB

Power In
Barrel Jack



Reset

Digital In/Out
PWM (3, 5, 6, 9, 10, 11)

Analog In

Ground

Power Out
(3.3V and 5V)

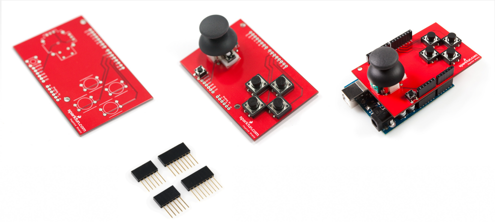
Ground

Program
USB
(Also Power In)

Power In
Barrel Jack
(Solar goes here)

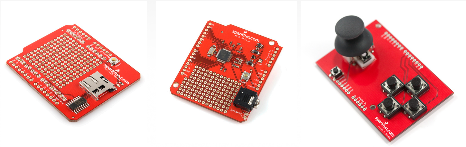
Arduino Shields

PCB — Built Shield — Inserted Shield



Arduino Shields

Micro SD — MP3 Trigger — Joystick



Components

Name	Image	Type	Function	Notes
Button		Digital Input	Closes or opens circuit	Polarized, needs resistor
Trimpot		Analog Input	Variable resistor	
Photoresistor		Analog Input	Variable resistor	
Relay		Digital Output & Input	Switches between circuits	Used to control larger voltages
Temp Sensor		Analog Input	Variable resistor	
Flex Sensor		Analog Input	Variable resistor	Only bends one way
Soft Trimpot		Analog Input	Variable resistor	Careful of shorts
RGB LED		Dig. & Analog Output	16,777,216 different colors	Ooh... So pretty.

Ohm's Law

Ohm's Law describes the direct relationship between the Voltage, Current and Resistance of a circuit.

The three different forms of Ohm's Law are as follows:

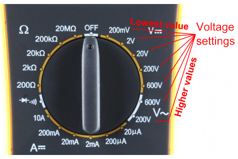
V = I * R I = V / R R = V / I

Where V is Voltage, I is Current and R is Resistance.

Ohm's Law

Why do I care about Voltage?

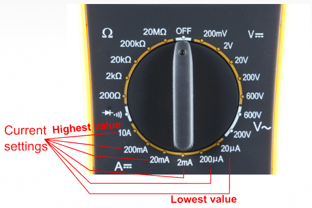
Voltage allows electricity to travel through all the components of the circuit. Voltage of your power source must be more than the total voltage drop of your circuit so the electricity can travel from power to ground.



Ohm's Law

Why do I care about Current?

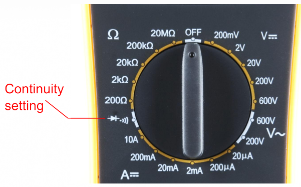
Current is the aspect of electricity that performs physical work or movement.



Not Ohm's Law

Why do I care about Continuity?

Continuity is important to make portions of circuits are connect. Continuity is the simplest and possibly the most important setting on your multimeter.

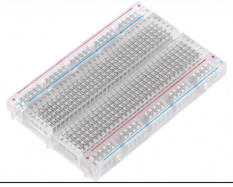


Continuity setting

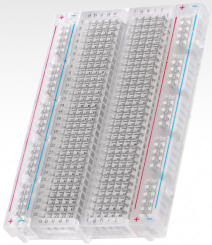
What's a Breadboard?

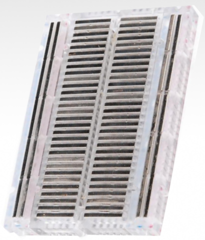
One of the most useful tools in an engineer or Maker's toolkit. The three most important thing to remember:

- A breadboard is easier than soldering
- A lot of those little holes are connected, which ones?
- Sometimes breadboards break



What's a Breadboard?





Analog and Digital

- All Arduino signals are either Analog or Digital
- All computers including Arduino, only understand Digital
- It is important to understand the difference between Analog and Digital signals since Analog signals require an Analog to Digital conversion

Output

Output is any signal exiting an electrical system

- Almost all systems that use physical computing will have some form of output
- The outputs in SIK include LEDs, a motor, a servo, a piezo element, a relay and an RGB LED

Output

Output is <u>always</u> Digital

To Output a Digital signal (On or Off) use this code:

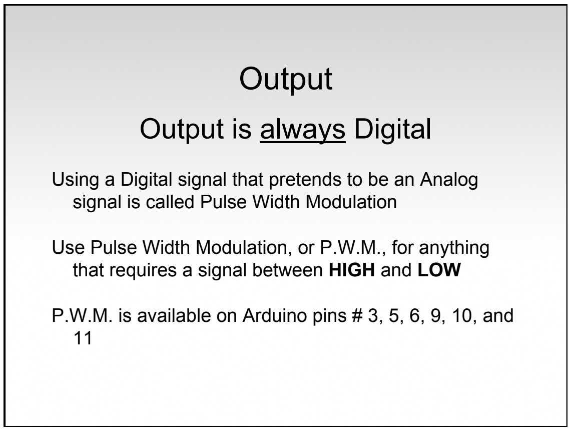
digitalWrite (pinNumber , value);

Where value is **HIGH** or **LOW**

To output a signal that pretends to be Analog use this code:

analogWrite (pinNumber, value);

Where value is a number 0 - 255



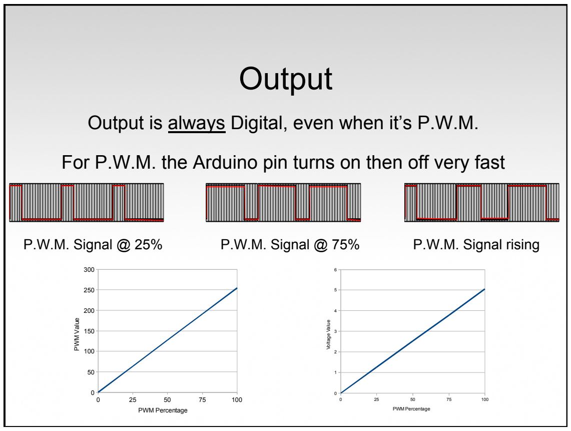
Output

Output is always Digital

Using a Digital signal that pretends to be an Analog signal is called Pulse Width Modulation

Use Pulse Width Modulation, or P.W.M., for anything that requires a signal between **HIGH** and **LOW**

P.W.M. is available on Arduino pins # 3, 5, 6, 9, 10, and 11



Output

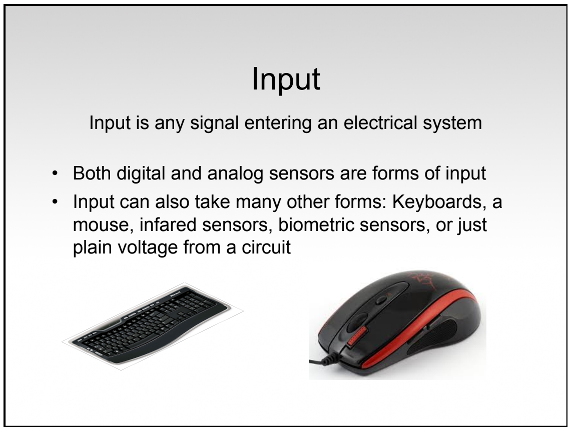
Output is always Digital, even when it's P.W.M.

For P.W.M. the Arduino pin turns on then off very fast

P.W.M. Signal @ 25%

P.W.M. Signal @ 75%

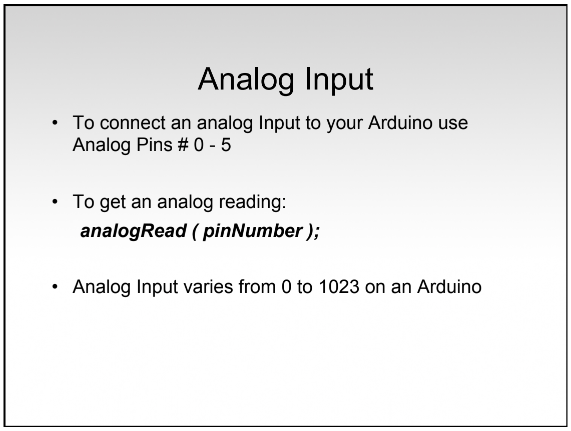
P.W.M. Signal rising



Input

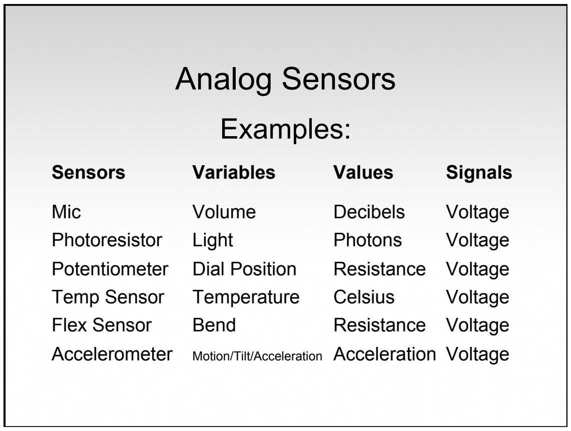
Input is any signal entering an electrical system

- Both digital and analog sensors are forms of input
- Input can also take many other forms: Keyboards, a mouse, infared sensors, biometric sensors, or just plain voltage from a circuit



Analog Input

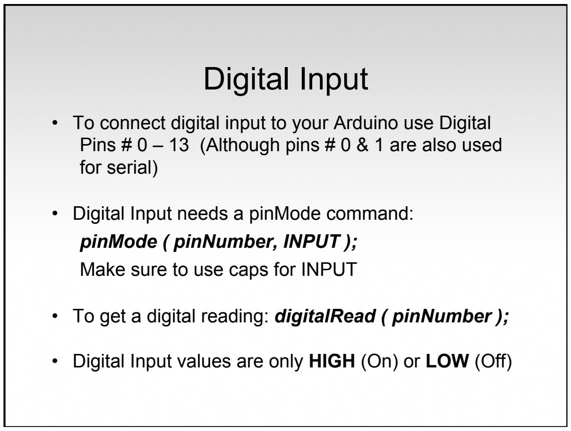
- To connect an analog Input to your Arduino use Analog Pins # 0 - 5
- To get an analog reading:
  analogRead (pinNumber);
- Analog Input varies from 0 to 1023 on an Arduino



Analog Sensors

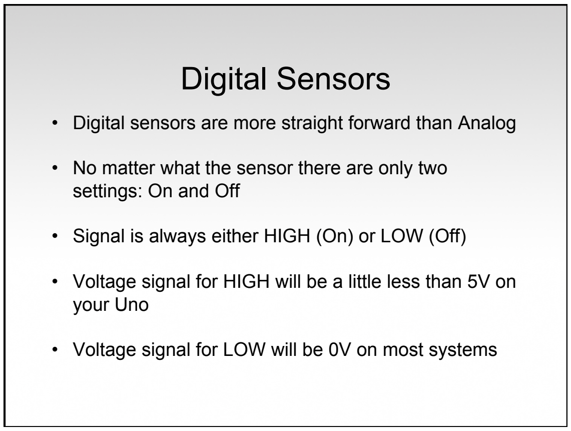
Examples:

Sensors	**Variables**	**Values**	**Signals**
Mic	Volume	Decibels	Voltage
Photoresistor	Light	Photons	Voltage
Potentiometer	Dial Position	Resistance	Voltage
Temp Sensor	Temperature	Celsius	Voltage
Flex Sensor	Bend	Resistance	Voltage
Accelerometer	Motion/Tilt/Acceleration	Acceleration	Voltage



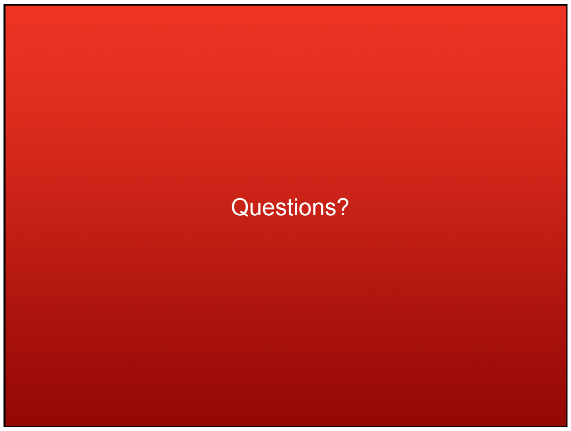
Digital Input

- To connect digital input to your Arduino use Digital Pins # 0 – 13 (Although pins # 0 & 1 are also used for serial)
- Digital Input needs a pinMode command:
  pinMode (pinNumber, INPUT);
  Make sure to use caps for INPUT
- To get a digital reading: ***digitalRead (pinNumber);***
- Digital Input values are only **HIGH** (On) or **LOW** (Off)



Digital Sensors

- Digital sensors are more straight forward than Analog
- No matter what the sensor there are only two settings: On and Off
- Signal is always either HIGH (On) or LOW (Off)
- Voltage signal for HIGH will be a little less than 5V on your Uno
- Voltage signal for LOW will be 0V on most systems



Questions?

sparkfun™ ELECTRONICS

www.sparkfun.com
6175 Longbow Drive, Suite 200
Boulder, Colorado 80301

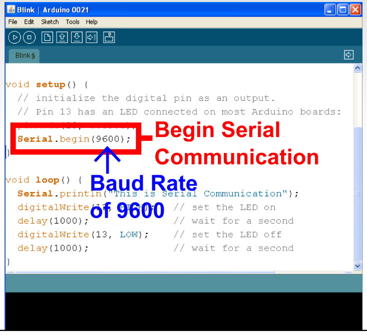
sparkfun ELECTRONICS

Programming, Serial and Virtual Prototyping

Serial Communication

Serial Communication is the transferring and receiving of information between two machines, the Arduino dedicates pin # 0 to receiving information and pin 1 to transferring information

Serial in Setup

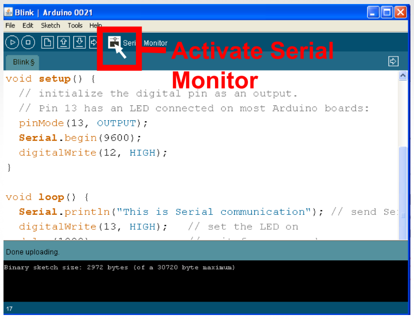


```
void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  Serial.begin(9600);
}

void loop() {
  Serial.println("This is Serial Communication");
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

Begin Serial Communication

Baud Rate of 9600

Serial Monitor

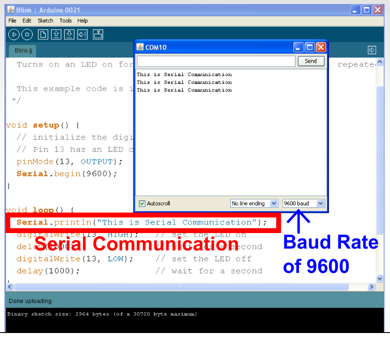


```
void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
  Serial.begin(9600);
  digitalWrite(12, HIGH);
}

void loop() {
  Serial.println("This is Serial communication"); // send Se
  digitalWrite(13, HIGH);   // set the LED on
```

```
Done uploading.
Binary sketch size: 2972 bytes (of a 30720 byte maximum)
```

Activate Serial Monitor

Serial Communication



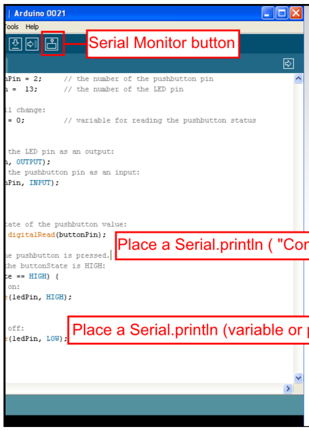
```
  Turns on an LED on for                repeate

  This example code is i
 */

void setup() {
  // initialize the digi
  // Pin 13 has an LED c
  pinMode(13, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  Serial.println("This is Serial Communication");
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
```

```
This is Serial Communication
This is Serial Communication
This is Serial Communication
```

```
Done uploading.
Binary sketch size: 2964 bytes (of a 30720 byte maximum)
```

Serial Communication

Baud Rate of 9600



```
nPin = 2;     // the number of the pushbutton pin
n =  13;      // the number of the LED pin

ll change:
 = 0;         // variable for reading the pushbutton status


 the LED pin as an output:
n, OUTPUT);
 the pushbutton pin as an input:
nPin, INPUT);



tate of the pushbutton value:
 digitalRead(buttonPin);

he pushbutton is pressed.
the buttonState is HIGH:
te == HIGH) {
 on:
e(ledPin, HIGH);


 off:
e(ledPin, LOW);
```

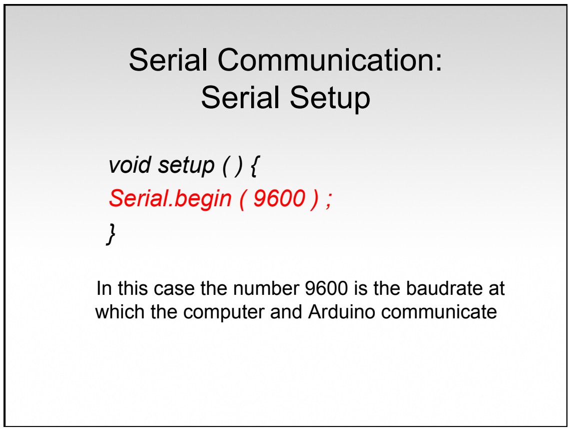
Serial Monitor button

Place a Serial.println ("Comment") here.

Place a Serial.println (variable or pinState) here.

Serial Activity with Circuit 7

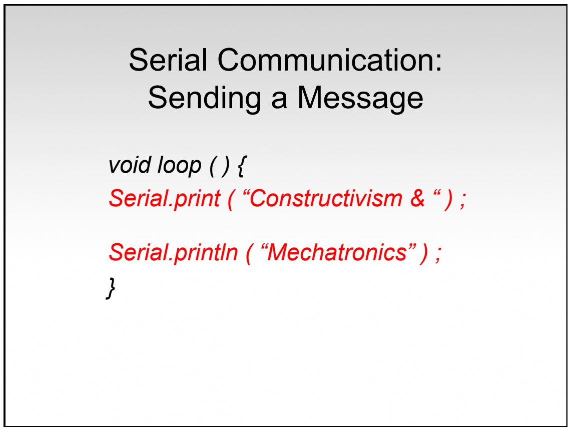
- Communication
- Troubleshooting circuits
- Debugging Code



Serial Communication: Serial Setup

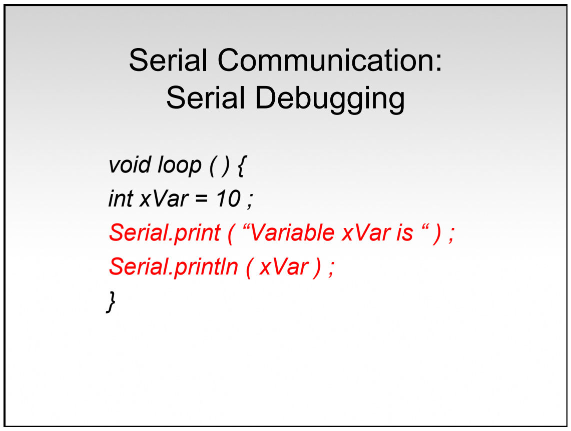
```
void setup ( ) {
Serial.begin ( 9600 ) ;
}
```

In this case the number 9600 is the baudrate at which the computer and Arduino communicate



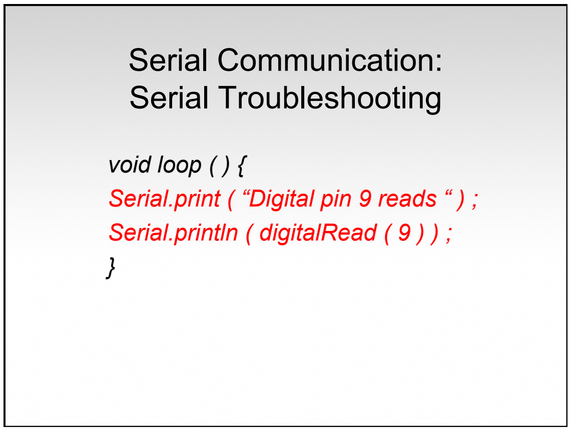
Serial Communication: Sending a Message

```
void loop ( ) {
Serial.print ( “Constructivism & “ ) ;

Serial.println ( “Mechatronics” ) ;
}
```



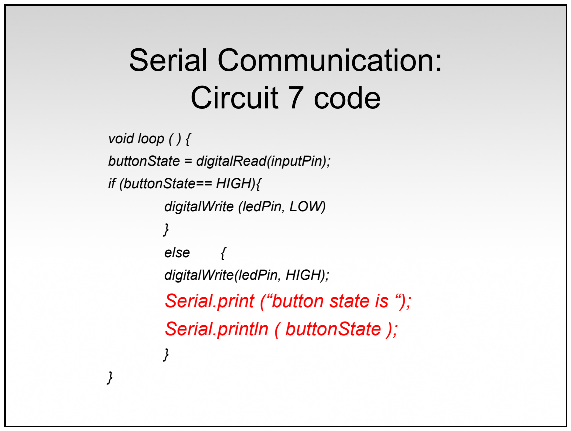
Serial Communication: Serial Debugging

```
void loop ( ) {
int xVar = 10 ;
Serial.print ( “Variable xVar is “ ) ;
Serial.println ( xVar ) ;
}
```



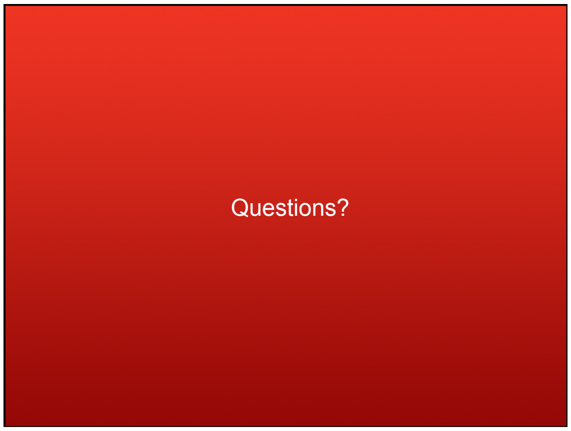
Serial Communication: Serial Troubleshooting

```
void loop ( ) {
Serial.print ( “Digital pin 9 reads “ ) ;
Serial.println ( digitalRead ( 9 ) ) ;
}
```



Serial Communication: Circuit 7 code

```
void loop ( ) {
buttonState = digitalRead(inputPin);
if (buttonState== HIGH){
        digitalWrite (ledPin, LOW)
        }
        else     {
        digitalWrite(ledPin, HIGH);
        Serial.print (“button state is “);
        Serial.println ( buttonState );
        }
}
```



Questions?

sparkfun™
ELECTRONICS

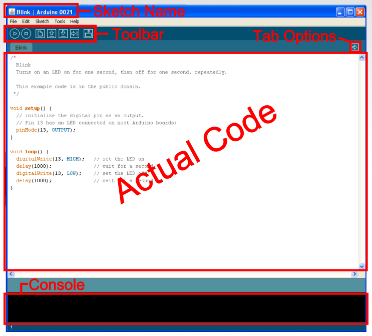
www.sparkfun.com
6175 Longbow Drive, Suite 200
Boulder, Colorado 80301

sparkfun ELECTRONICS

Code

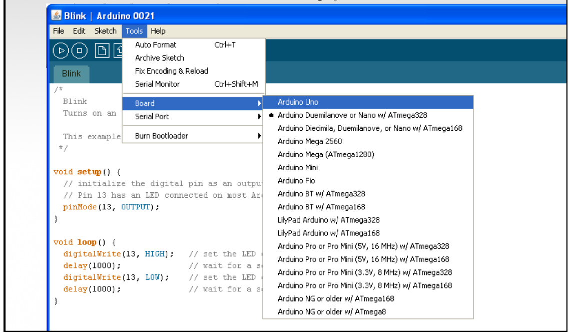
http://arduino.cc/en/Reference/HomePage

The Arduino Environment



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

  This example code is in the public domain.
 */

void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

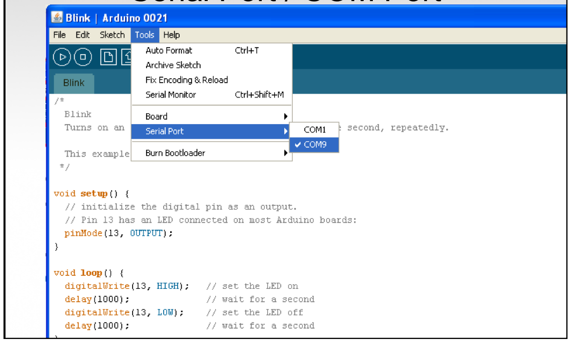
Board Type



Blink | Arduino 0021
File Edit Sketch Tools Help
Blink
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Board
Serial Port
Burn Bootloader
Arduino Uno
● Arduino Duemilanove or Nano w/ ATmega328
Arduino Diecimila, Duemilanove, or Nano w/ ATmega168
Arduino Mega 2560
Arduino Mega (ATmega1280)
Arduino Mini
Arduino Fio
Arduino BT w/ ATmega328
Arduino BT w/ ATmega168
LilyPad Arduino w/ ATmega328
LilyPad Arduino w/ ATmega168
Arduino Pro or Pro Mini (5V, 16 MHz) w/ ATmega328
Arduino Pro or Pro Mini (5V, 16 MHz) w/ ATmega168
Arduino Pro or Pro Mini (3.3V, 8 MHz) w/ ATmega328
Arduino Pro or Pro Mini (3.3V, 8 MHz) w/ ATmega168
Arduino NG or older w/ ATmega168
Arduino NG or older w/ ATmega8

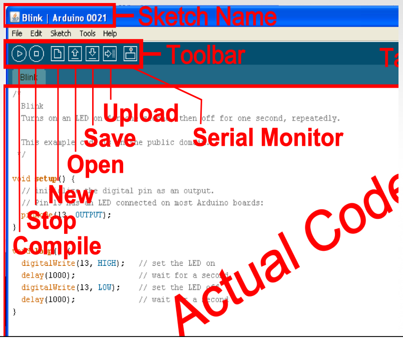
```
/*
  Blink
  Turns on an

  This example
 */

void setup() {
  // initialize the digital pin as an outpu
  // Pin 13 has an LED connected on most Ar
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED
  delay(1000);              // wait for a s
  digitalWrite(13, LOW);    // set the LED
  delay(1000);              // wait for a s
}
```

Serial Port / COM Port



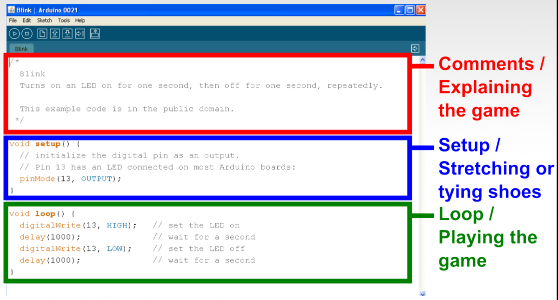
```
/*
  Blink
  Turns on an                    second, repeatedly.

  This example
 */

void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
```

The Environment



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

  This example c           he public dom
 */

void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a seco
  digitalWrite(13, LOW);    // set the LED o
  delay(1000);              // wait    a   cond
}
```

Parts of the Sketch



Blink | Arduino 0021
File Edit Sketch Tools Help
Blink
Comments / Explaining the game
Setup / Stretching or tying shoes
Loop / Playing the game

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

  This example code is in the public domain.
 */

void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

Comments

- Comments can be anywhere

Comments

- Comments can be anywhere
- Comments created with // or /* and */

Comments

- Comments can be anywhere
- Comments created with // or /* and */
- Comments do not affect code

Comments

- Comments can be anywhere
- Comments created with // or /* and */
- Comments do not affect code
- You may not need comments, but think about the community!

Operators

The equals sign

= is used to assign a value

== is used to compare values

Operators

And & Or

&& is “and”

|| is “or”

Variables

Basic variable types:

Boolean
Integer
Character

Declaring Variables

Boolean: ***boolean variableName;***

Declaring Variables

Boolean: ***boolean variableName;***

Integer: ***int variableName;***

Declaring Variables

Boolean: ***boolean variableName;***

Integer: ***int variableName;***

Character: ***char variableName;***

Declaring Variables

Boolean: ***boolean variableName;***

Integer: ***int variableName;***

Character: ***char variableName;***
String: ***stringName [ ];***

Assigning Variables

Boolean: ***variableName = true;***
or ***variableName = false;***

Assigning Variables

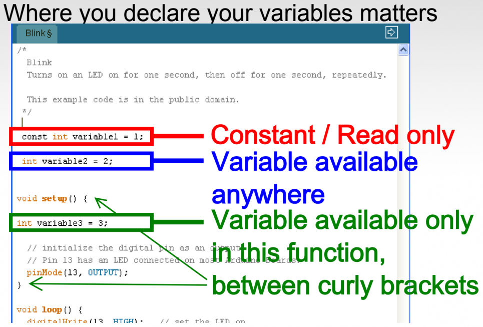
Boolean: ***variableName = true;***
or ***variableName = false;***
Integer: ***variableName = 32767;***
or ***variableName = -32768;***
Integer data size comes from
-2^15 to (2^15)-1

Assigning Variables

Boolean: ***variableName = true;***
or ***variableName = false;***
Integer: ***variableName = 32767;***
or ***variableName = -32768;***
Character: ***variableName = 'A';***
or ***stringName = "SparkFun";***

Variable Scope

Where you declare your variables matters



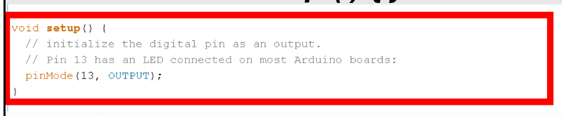
```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
 */

 const int variable1 = 1;

 int variable2 = 2;


void setup() {

int variable3 = 3;

  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED on
```

Constant / Read only

Variable available anywhere

Variable available only in this function, between curly brackets

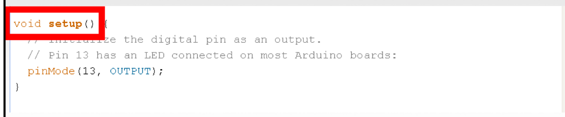
Setup
void setup ( ) { }



```
void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}
```

The setup function comes before the loop function and is necessary for all Arduino sketches

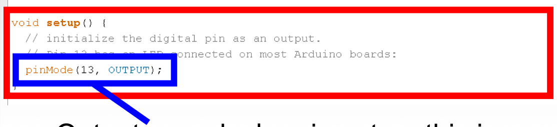
Setup
void setup ( ) { }



```
void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}
```

The setup header will never change, everything else that occurs in setup happens inside the curly brackets

Setup
void setup ( ) {
pinMode (13, OUTPUT); }



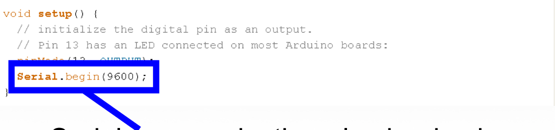
```
void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}
```

Outputs are declare in setup, this is done by using the pinMode function

This particular example declares digital pin # 13 as an output, remember to use CAPS

Setup

void setup ( ) { Serial.begin;}



```
void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
  Serial.begin(9600);
}
```

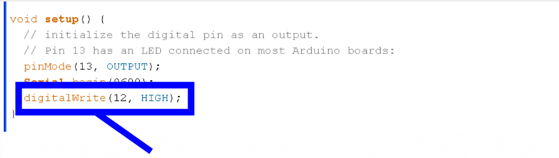
Serial communication also begins in setup

This particular example declares Serial communication at a baud rate of 9600. More on Serial later...

Setup, Internal Pullup Resistors

void setup ( ) {

digitalWrite (12, HIGH); }



```
void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
  Serial.begin(9600);
  digitalWrite(12, HIGH);
}
```

You can also create internal pullup resistors in setup, to do so digitalWrite the pin HIGH

This takes the place of the pullup resistors currently on your circuit 7 buttons

Setup, Interrupts

void setup ( ) {

attachInterrupt (interrupt, function, mode) }

You can designate an interrupt function to Arduino pins # 2 and 3

This is a way around the linear processing of Arduino

Setup, Interrupts

void setup ( ) {

attachInterrupt (interrupt, function, mode) }

Interrupt: the number of the interrupt, 0 or 1, corresponding to Arduino pins # 2 and 3 respectively

Function: the function to call when the interrupt occurs

Mode: defines when the interrupt should be triggered

Setup, Interrupts

void setup ( ) {

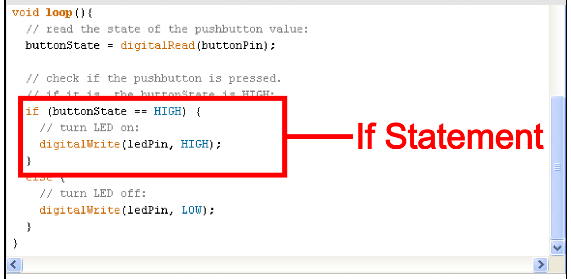
attachInterrupt (interrupt, function, mode) }

- ***LOW*** whenever pin state is low
- ***CHANGE*** whenever pin changes value
- ***RISING*** whenever pin goes from low to high
- ***FALLING*** whenever pin goes from low to high

Don't forget to CAPITALIZE

If Statements

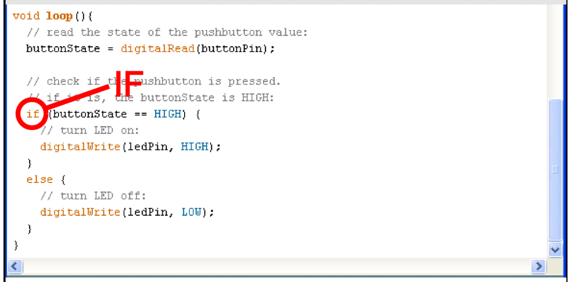
if (this is true) { do this; }



```
void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

If

if (this is true) { do this; }

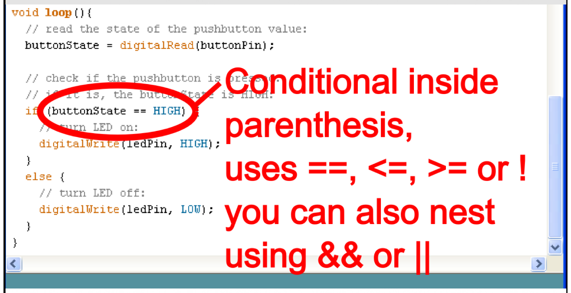


```
void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

IF

Conditional

if (this is true) { do this; }

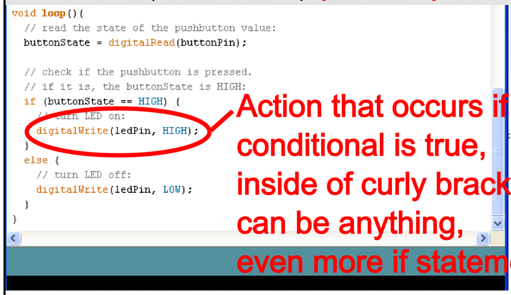


```
void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

Conditional inside parenthesis, uses ==, <=, >= or ! you can also nest using && or ||

Action

if (this is true) { do this; }

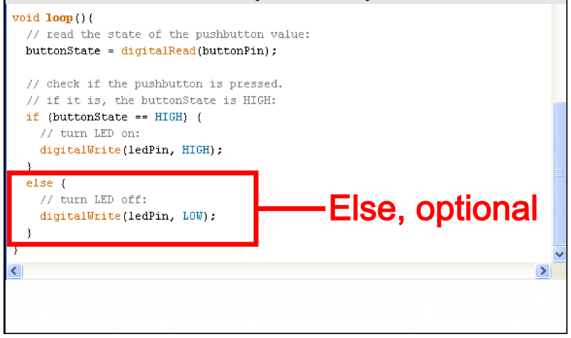


```
void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

Action that occurs if conditional is true, inside of curly brackets, can be anything, even more if statements

Else

else { do this; }



```
void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

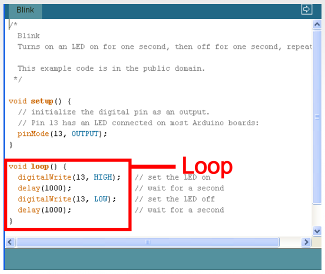
Else, optional

Basic Repetition

- loop
- For
- while

Basic Repetition

void loop ( ) { }

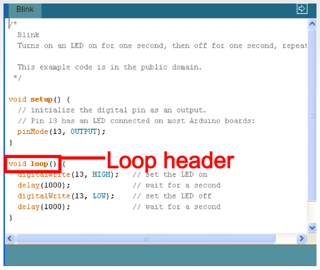


```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeat

  This example code is in the public domain.
 */

void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

Loop

Basic Repetition

void loop ( ) { }



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

  This example code is in the public domain.
 */

void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

Basic Repetition

void loop ( ) { }

The “void” in the header is what the function will return (or spit out) when it happens, in this case it returns nothing so it is void

Basic Repetition

void loop ( ) { }

The “loop” in the header is what the function is called, sometimes you make the name up, sometimes (like loop) the function already has a name

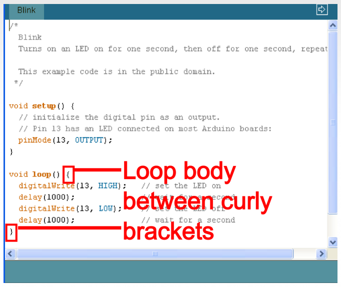
Basic Repetition

void loop ( ) { }

The “( )” in the header is where you declare any variables that you are “passing” (or sending) the function, the loop function is never “passed” any variables

Basic Repetition

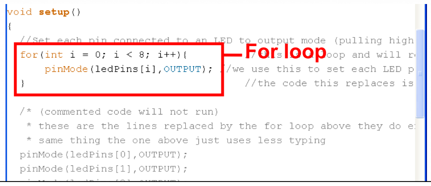
void loop ( ) { }



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

  This example code is in the public domain.
 */

void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

Basic Repetition

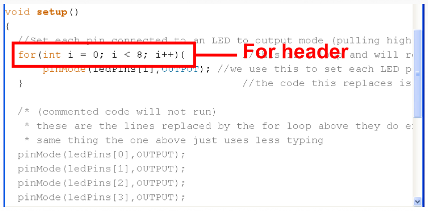
for (int count = 0; count<10; count++)
{
//for action code goes here
//this could be anything
}



```
void setup()
{
  //Set each pin connected to an LED to output mode (pulling high
  for(int i = 0; i < 8; i++){                //this is a loop and will r
      pinMode(ledPins[i],OUTPUT);          //we use this to set each LED p
  }                                          //the code this replaces is

  /* (commented code will not run)
   * these are the lines replaced by the for loop above they do e
   * same thing the one above just uses less typing
  pinMode(ledPins[0],OUTPUT);
  pinMode(ledPins[1],OUTPUT);
```

Basic Repetition

```
for (int count = 0; count<10; count++)
{
//for action code goes here
}
```

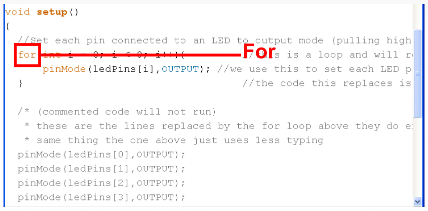


```
void setup()
{
  //Set each pin connected to an LED to output mode (pulling high
  for(int i = 0; i < 8; i++){                   //this is a loop and will r
      pinMode(ledPins[i],OUTPUT); //we use this to set each LED p
  }                                   //the code this replaces is

  /* (commented code will not run)
   * these are the lines replaced by the for loop above they do e
   * same thing the one above just uses less typing
  pinMode(ledPins[0],OUTPUT);
  pinMode(ledPins[1],OUTPUT);
  pinMode(ledPins[2],OUTPUT);
  pinMode(ledPins[3],OUTPUT);
```

For header

Basic Repetition

```
for (int count = 0; count<10; count++)
{
//for action code goes here
}
```

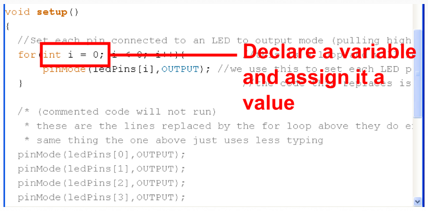


```
void setup()
{
  //Set each pin connected to an LED to output mode (pulling high
  for(int i = 0; i < 8; i++){                   //this is a loop and will r
      pinMode(ledPins[i],OUTPUT); //we use this to set each LED p
  }                                   //the code this replaces is

  /* (commented code will not run)
   * these are the lines replaced by the for loop above they do e
   * same thing the one above just uses less typing
  pinMode(ledPins[0],OUTPUT);
  pinMode(ledPins[1],OUTPUT);
  pinMode(ledPins[2],OUTPUT);
  pinMode(ledPins[3],OUTPUT);
```

For

Basic Repetition

```
for (int count = 0; count<10; count++)
{
//for action code goes here
}
```

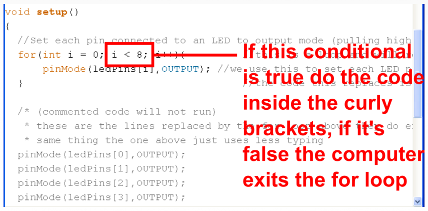


```
void setup()
{
  //Set each pin connected to an LED to output mode (pulling high
  for(int i = 0; i < 8; i++){                   //this is a loop and will r
      pinMode(ledPins[i],OUTPUT); //we use this to set each LED p
  }                                   //the code this replaces is

  /* (commented code will not run)
   * these are the lines replaced by the for loop above they do e
   * same thing the one above just uses less typing
  pinMode(ledPins[0],OUTPUT);
  pinMode(ledPins[1],OUTPUT);
  pinMode(ledPins[2],OUTPUT);
  pinMode(ledPins[3],OUTPUT);
```

Declare a variable and assign it a value

Basic Repetition

```
for (int count = 0; count<10; count++)
{
//for action code goes here
}
```

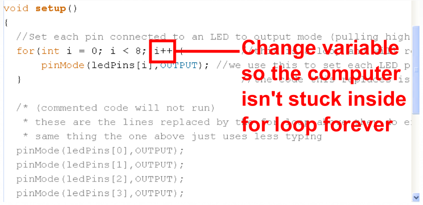


```
void setup()
{
  //Set each pin connected to an LED to output mode (pulling high
  for(int i = 0; i < 8; i++){                   //this is a loop and will r
      pinMode(ledPins[i],OUTPUT); //we use this to set each LED p
  }                                   //the code this replaces is

  /* (commented code will not run)
   * these are the lines replaced by the for loop above they do e
   * same thing the one above just uses less typing
  pinMode(ledPins[0],OUTPUT);
  pinMode(ledPins[1],OUTPUT);
  pinMode(ledPins[2],OUTPUT);
  pinMode(ledPins[3],OUTPUT);
```

If this conditional is true do the code inside the curly brackets, if it's false the computer exits the for loop

Basic Repetition

```
for (int count = 0; count<10; count++)
{
//for action code goes here
}
```

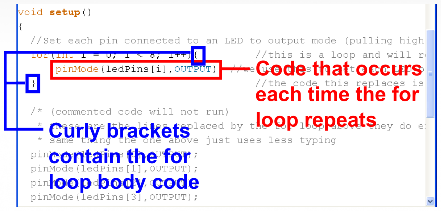


```
void setup()
{
  //Set each pin connected to an LED to output mode (pulling high
  for(int i = 0; i < 8; i++){                   //this is a loop and will r
      pinMode(ledPins[i],OUTPUT); //we use this to set each LED p
  }                                   //the code this replaces is

  /* (commented code will not run)
   * these are the lines replaced by the for loop above they do e
   * same thing the one above just uses less typing
  pinMode(ledPins[0],OUTPUT);
  pinMode(ledPins[1],OUTPUT);
  pinMode(ledPins[2],OUTPUT);
  pinMode(ledPins[3],OUTPUT);
```

Change variable so the computer isn't stuck inside for loop forever

Basic Repetition

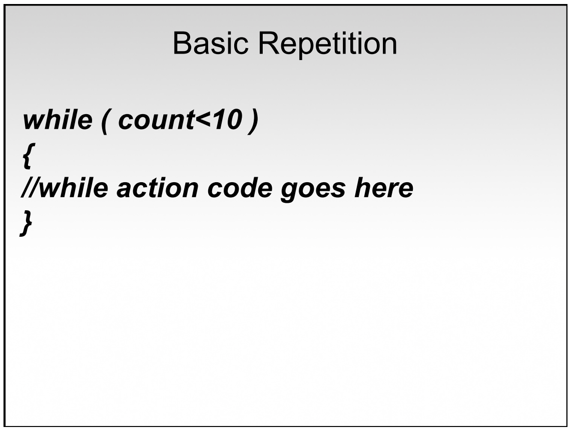
```
for (int count = 0; count<10; count++)
{
//for action code goes here
}
```



```
void setup()
{
  //Set each pin connected to an LED to output mode (pulling high
  for(int i = 0; i < 8; i++){                   //this is a loop and will r
      pinMode(ledPins[i],OUTPUT); //we use this to set each LED p
  }                                   //the code this replaces is

  /* (commented code will not run)
   * these are the lines replaced by the for loop above they do e
   * same thing the one above just uses less typing
  pinMode(ledPins[0],OUTPUT);
  pinMode(ledPins[1],OUTPUT);
  pinMode(ledPins[2],OUTPUT);
  pinMode(ledPins[3],OUTPUT);
```

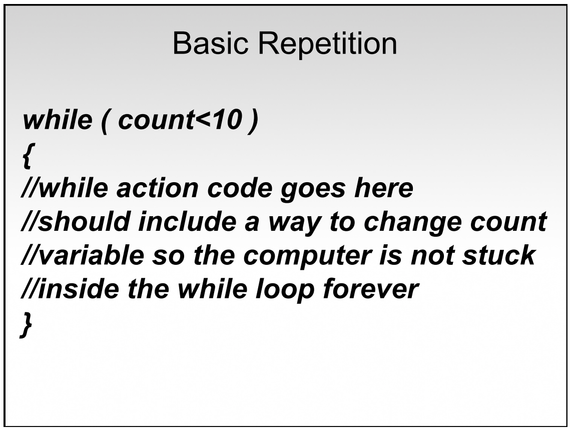
Code that occurs each time the for loop repeats

Curly brackets contain the for loop body code



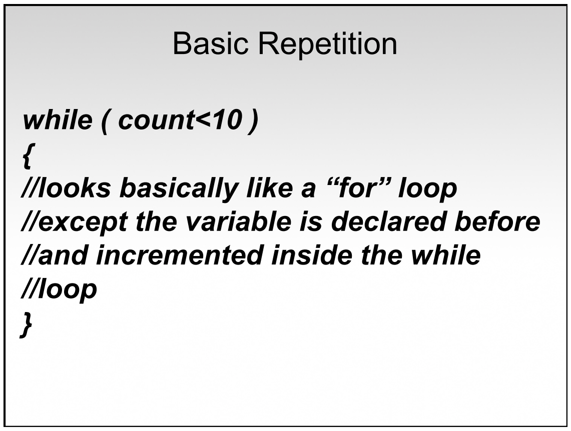
Basic Repetition

```
while ( count<10 )
{
//while action code goes here
}
```



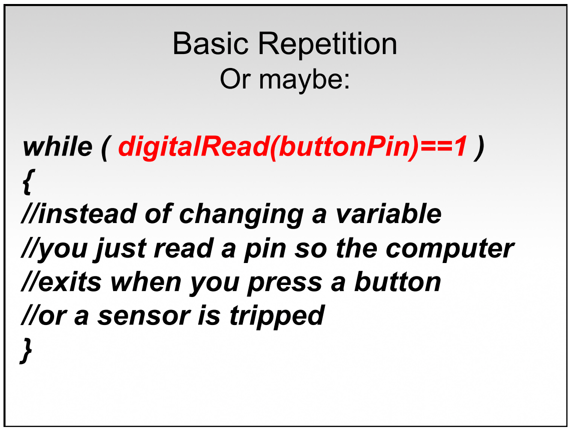
Basic Repetition

```
while ( count<10 )
{
//while action code goes here
//should include a way to change count
//variable so the computer is not stuck
//inside the while loop forever
}
```



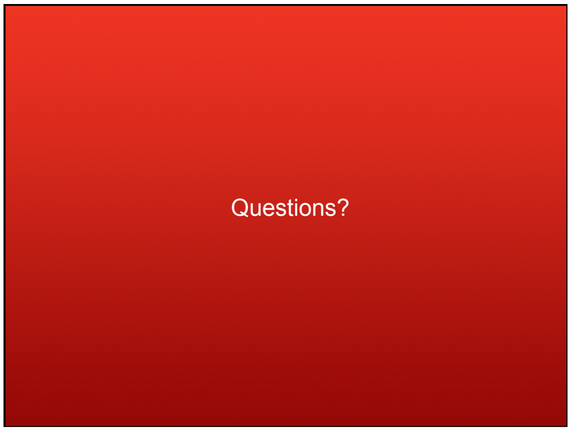
Basic Repetition

```
while ( count<10 )
{
//looks basically like a "for" loop
//except the variable is declared before
//and incremented inside the while
//loop
}
```



Basic Repetition
Or maybe:

```
while ( digitalRead(buttonPin)==1 )
{
//instead of changing a variable
//you just read a pin so the computer
//exits when you press a button
//or a sensor is tripped
}
```



Questions?

sparkfun ELECTRONICS

www.sparkfun.com
6175 Longbow Drive, Suite 200
Boulder, Colorado 80301

Installing Arduino

Mac platform	PC 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.	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)**

Issues with Java? Make sure you have the latest version of Java installed. Otherwise....
Ta da! You're ready to open the Arduino programming environment.

(For Linux info go to http://ardx.org/LINU)

A few notes about Arduino setup

Selecting your board:

You are using the Arduino Uno board with an ATmega 328 micro-controller. 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.

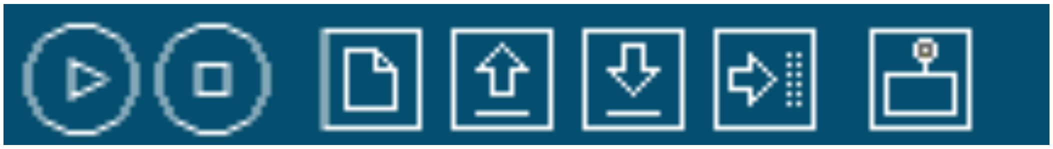
Selecting your com port:

Another option that is necessary to change occasionally 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 ports for your Arduino, here is some information on your com ports depending on which platform you are using:

Mac	PC
The Mac version of Arduino 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.	The PC version of Arduino 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.

A few more tidbits that will help to know....

There are seven buttons that look like this at the top of your Arduino window:



Here is what they do from left to right:

Compile	Stop	New	Open	Save	Upload	Serial Monitor
This checks your code for errors.	This stops the program.	This creates a new sketch.	This opens an existing sketch.	This saves the open sketch.	This uploads the sketch to your board.	Used to display Serial Communication.

For any words or phrases you are unfamiliar with try the Glossary in the back.

Basic Arduino Reference Sheet

Installation: Arduino: http://www.arduino.cc/en/Guide/HomePage
Support: Arduino: http://www.arduino.cc, http://www.freeduino.org, google.com
Forums: Arduino: http://forum.sparkfun.com/viewforum.php?f=32

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 Uno). 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.

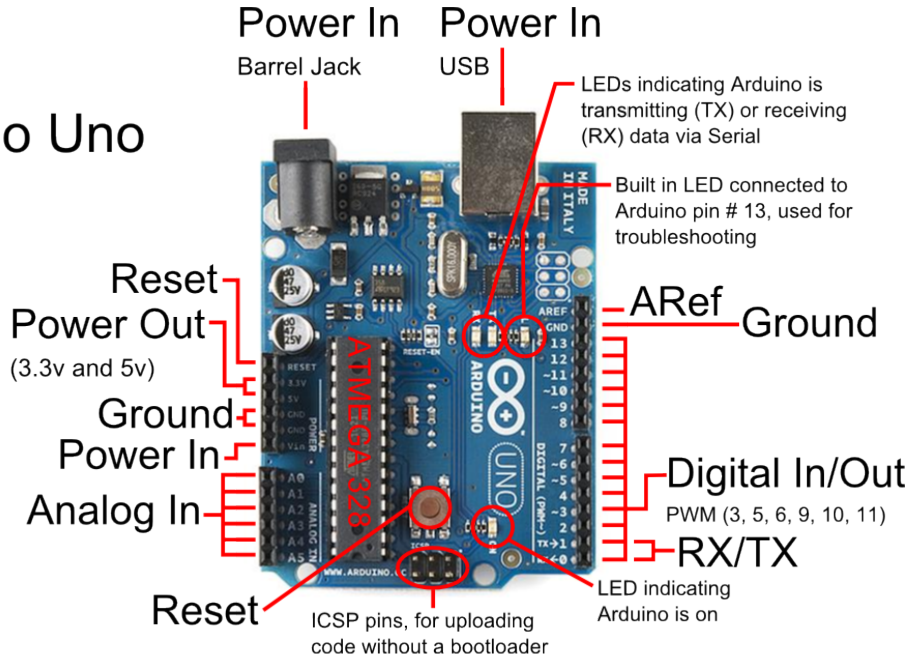
Arduino Uno pin type definitions: (Take a look at your Arduino board)

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

Introduction to Arduino
SparkFun Electronics Summer Semester

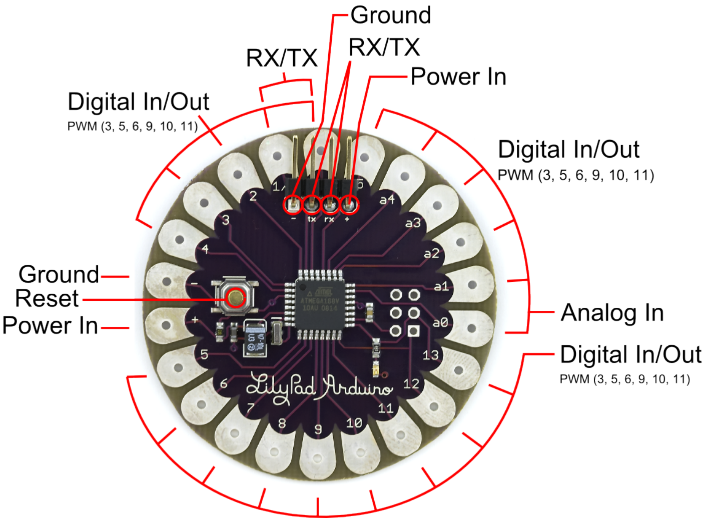
Basic Arduino Pin Reference Sheet

Arduino Uno

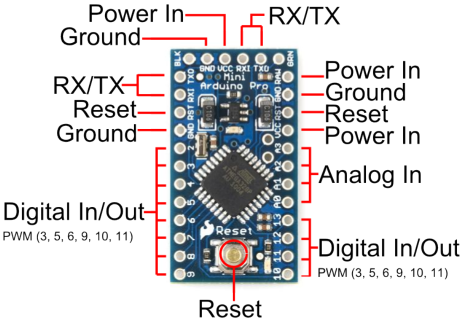

Power In
Barrel Jack
Power In
USB
LEDs indicating Arduino is transmitting (TX) or receiving (RX) data via Serial
Built in LED connected to Arduino pin # 13, used for troubleshooting
ARef
Ground
Reset
Power Out
(3.3v and 5v)
Ground
Power In
Analog In
Digital In/Out
PWM (3, 5, 6, 9, 10, 11)
RX/TX
Reset
ICSP pins, for uploading code without a bootloader
LED indicating Arduino is on
ATMEGA 328

These boards below use the same micro-controller, just in a different package. The Lilypad is designed for use with conductive thread instead of wire and the Arduino Mini is simply 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 Lilypad



Arduino Mini



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. (And if they are out of our range of acceptable temperatures then what are we doing there?)   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 Arduino 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 Arduino then assigns a numerical value to the signal somewhere between 0 (0%) and 1023 (100%). The digital system of the Arduino 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 Arduino 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, spiciness, 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 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})

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.

In the world of Arduino, 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 Arduino the HIGH signals will be a little under 5V and on a 3.3V Arduino 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 Arduino 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 send a Digital signal use: `digitalWrite(pinNumber, value);`
Digital Input and Output are always either HIGH or LOW

Input Signals

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

Analog Input enters your Arduino 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 Arduino 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 Arduino 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 as inputs use:

```
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

Output Signals

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

Output to the Arduino 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 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:

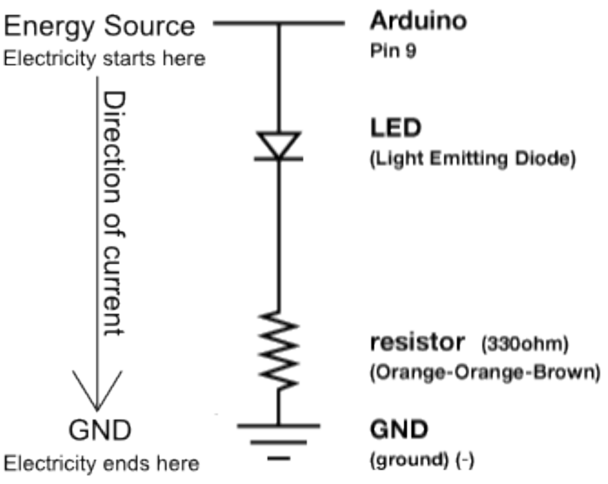
Light Emitted Diodes (LED's), Piezoelectric Speakers, Servo Motors

Circuit 1

Explanation:

This circuit takes electricity from digital Pin # 9 on the Arduino. Pin # 9 on the Arduino 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.

Schematic:



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 = 3;
 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 output loop could read :

```
int ledPin = 3;
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
}
```

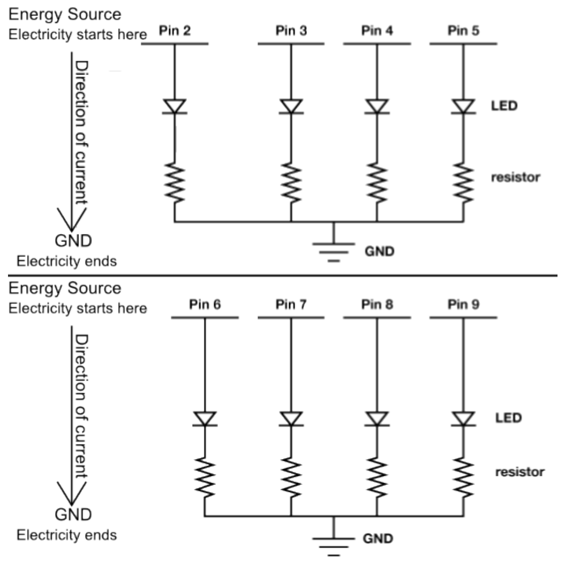
This first circuit is the simplest form of output in the kit. You can use the LED to teach both analog and digital output before moving on to more exciting outputs. There is an LED built into your Arduino board which corresponds to Digital Pin # 13.

Circuit 2

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.

Schematic:



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

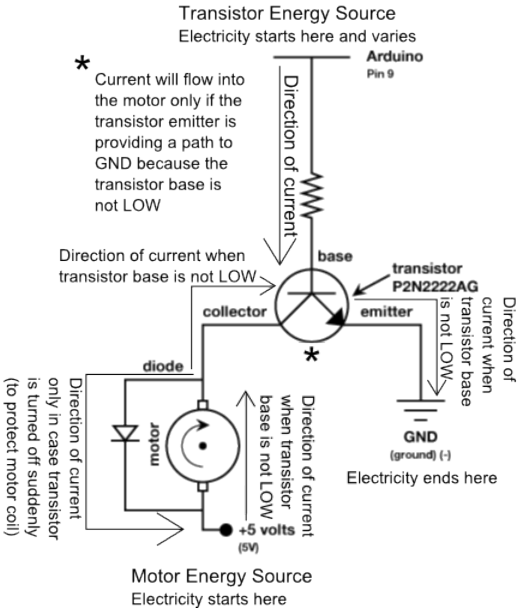
The code examples in the S.I.K get a little complicated for the second circuit, but don't worry, it's just more outputs. Some of the code examples use “for” loops to do something a number of times, if you're not familiar with “for” look it up because it is a key programming concept.

Circuit 3

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.

Schematic:


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
Direction of current only in case transistor is turned off suddenly (to protect motor coil)
motor
Direction of current when transistor base is not LOW
+5 volts
(5V)
GND
(ground) (-)
Electricity ends here
Motor Energy Source
Electricity starts here

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);

   }

}
```

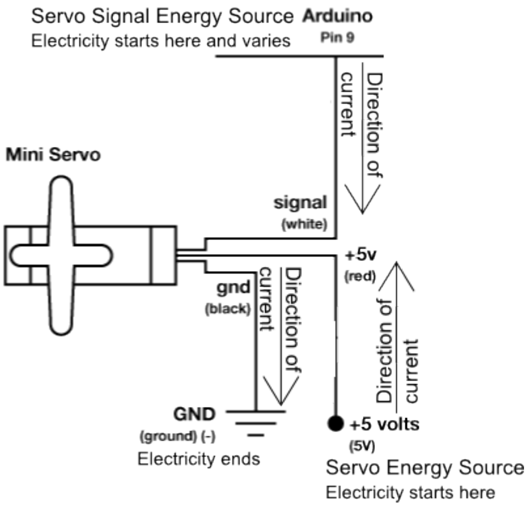
This circuit is great; it teaches about transistors, one of the basic electronic building blocks.

Circuit 4

Explanation:

The servo in this circuit takes electricity from 5V on the Arduino. Pin # 9 on the Arduino supplies a PWM signal that 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.

Schematic:



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);
    }
}
```

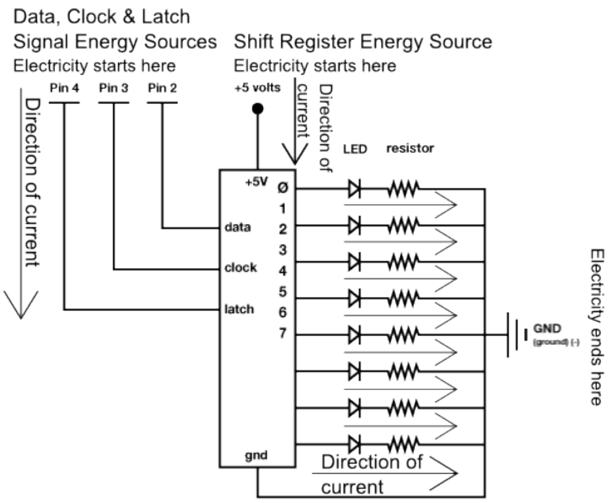
Remember, this is just slightly more complicated output, same as the motor and LED.

Circuit 5

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.

Schematic:



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 to datasheet.

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);
}
```

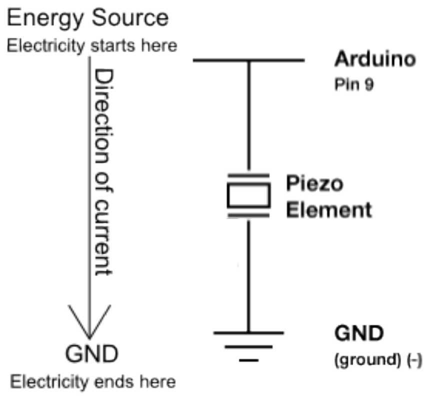
For more advanced components you will need to read Datasheets to figure out how to use them. Any documentation is good as long as you can get the correct information out of it.

Circuit 6

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.

Schematic:



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

Code:

```
/* 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. */

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);
      }
   }
}
```

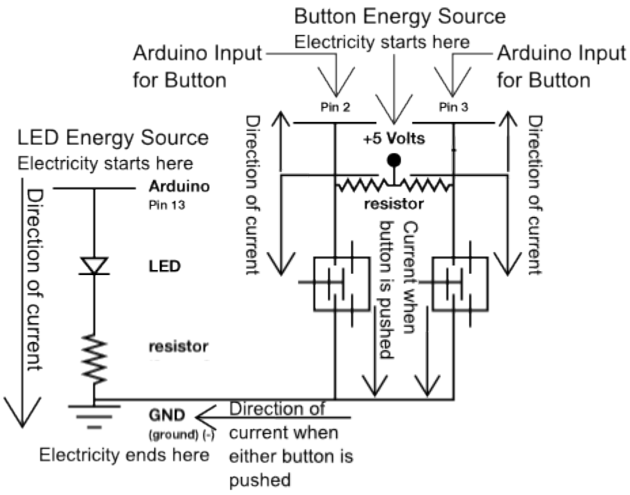
This is another way to use digital pins to create analog output.

Circuit 7

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.

Schematic:



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 10K's 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);
   }
}
```

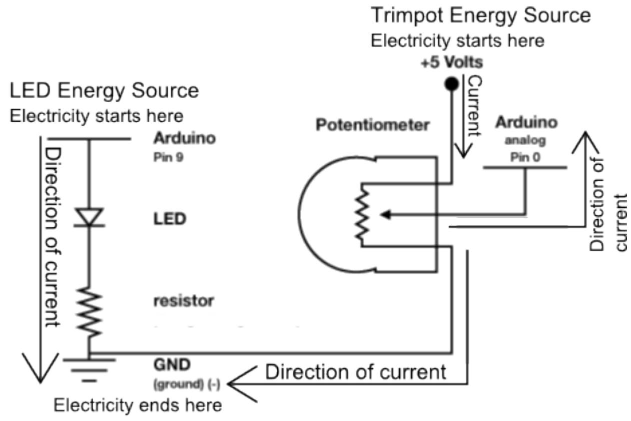
In this circuit you are using a Digital Pin, but you are using it as input rather than output.

Circuit 8

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.

Schematic:



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);
}
```

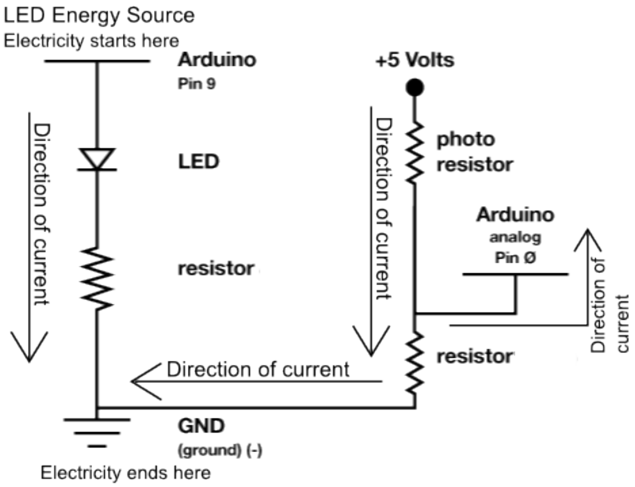
This is another example of input, only this time it is Analog. Circuits 7 and 8 in the S.I.K. introduce you to the two kinds of input your board can receive: Digital and Analog.

Circuit 9

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 affects 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.

Schematic:



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.

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 corresponds to the light hitting the sensor.

+5V: Five Volt power source.

Gnd: Ground

Code:

```
int lightPin = 0;
int ledPin = 9;

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);
}
```

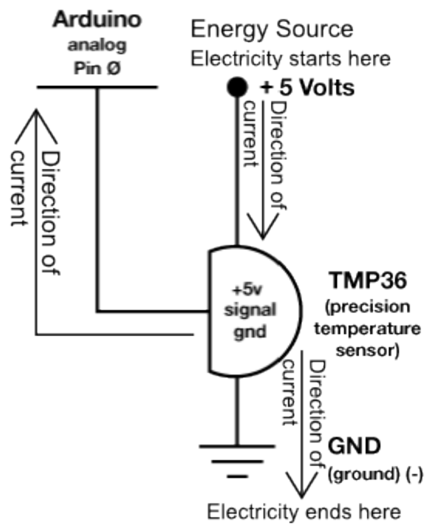
This circuit is another example of Analog input. It is also a perfect example of a voltage divider. Don't worry about the “map” and “constrain” functions, they are explained in the glossary.

Circuit 10

Explanation:

This circuit takes electricity from the 5V on the Arduino. The temp 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 it's Serial monitor to display the temperature reading.

Schematic:



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);
}
```

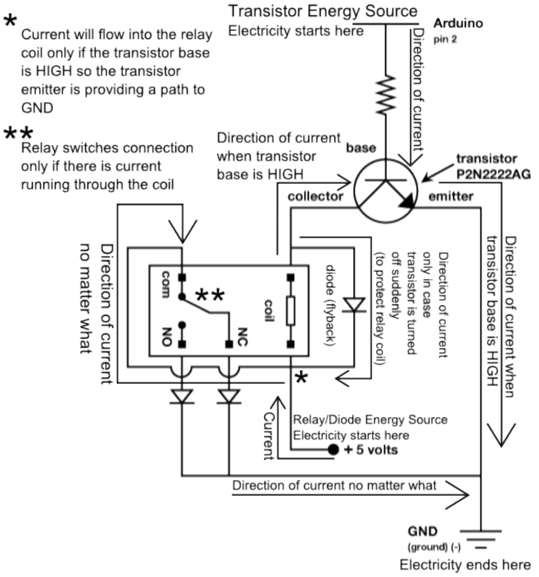
There is a lot of math involved in the code section of this circuit and it all has a reason. But, how would you know you need to offset the temperature reading by .5 unless you had read the Datasheet? Also, pay attention to the code lines that enable Serial communication.

Circuit 11

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.

Schematic:


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

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);
}
```

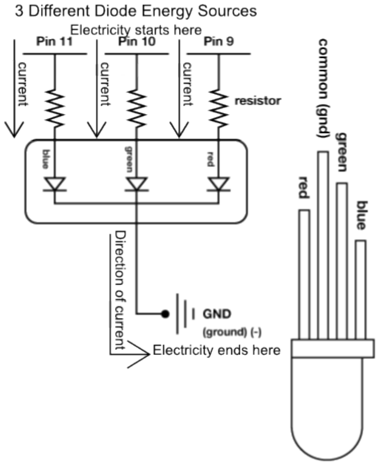
Transistors can be used as switches or amplifiers and they are often called the most important invention of the 20th century. The relay is also a great control component, but it needs something to activate it, hence the transistor.

Circuit 12

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!

Schematic:



Components:

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

RGB LED: Unlike single color LED's, on RGB (Also called 'Tri-Color') LED's, 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:

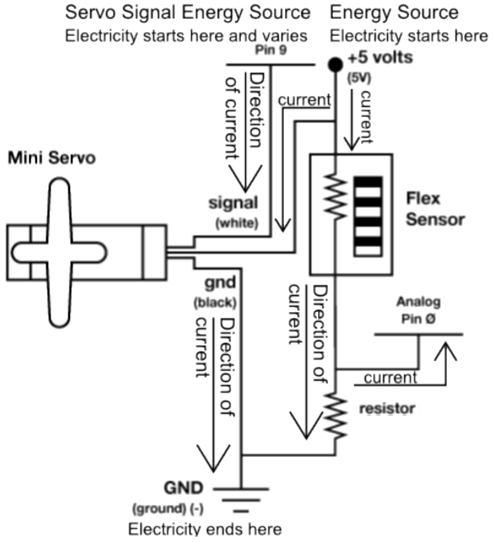
```
const int RED_LED_PIN = 9;
const int GREEN_LED_PIN = 10;
const int BLUE_LED_PIN = 11;
int redIntensity = 0;
int greenIntensity = 0;
int blueIntensity = 0;
const int DISPLAY_TIME = 100;

void setup() {
}

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);
  }
}
```

Circuit 13

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 4 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.

Schematic:



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>
Servo myservo;

int potpin = 0;
int val;

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

void loop() {
  val = analogRead(potpin);
  Serial.println(val);
  val = map(val, 50, 300, 0, 179);
  myservo.write(val);
  delay(15);
}
```

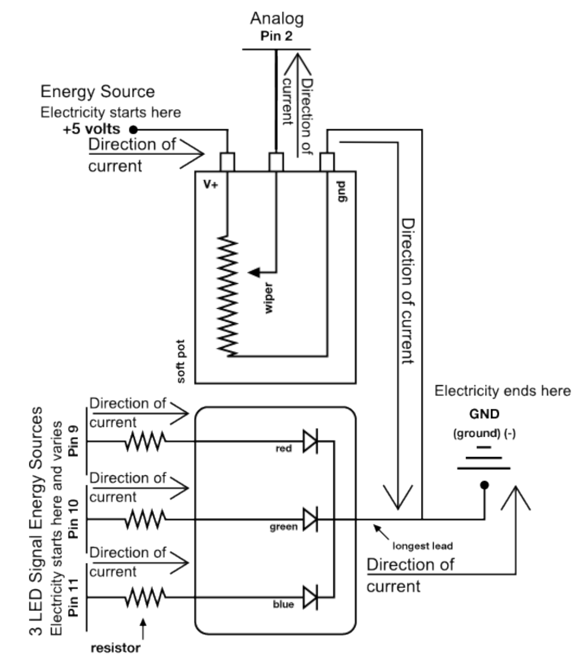
This analog input is definitely very different from any other input we have looked at so far but the concept is the same. We treat the sensor as a resistor in a voltage divider to get a reading and then change our output depending on that reading.

Circuit 14

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 12 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.

Schematic:



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);
}
```

Powering Your Projects

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 Arduino 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.

Powering Your Projects

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.

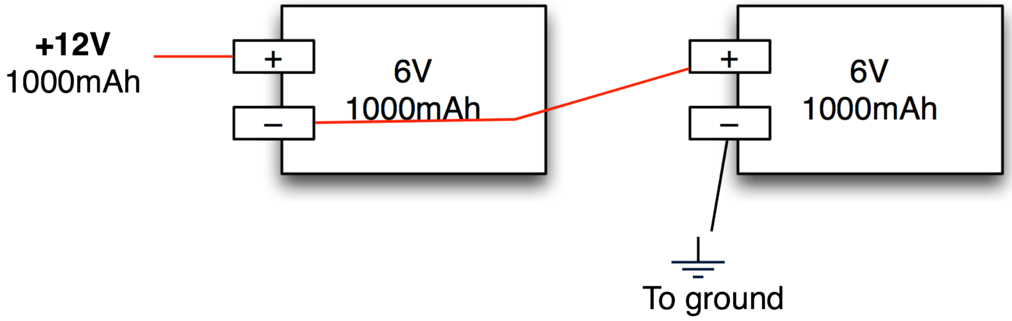
Here's the rule:

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

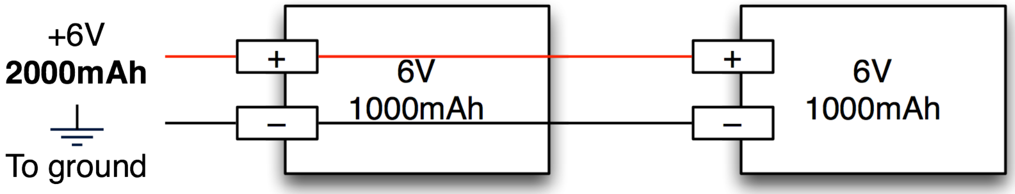
Connecting batteries **in parallel** maintains the voltage but increases the capacity. This is what you want to do if you need your power supply to last longer.

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.

Transistors

What is a transistor?

Transistors are semiconductors used to amplify or switch an electrical signal on and off.

Why is a transistor useful?

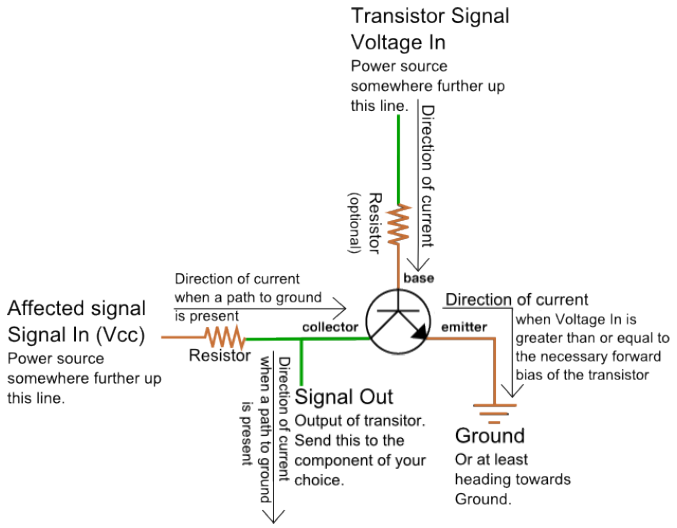
Often you will need more power to run a component than your Arduino can provide. A transistor allows you to control the higher power signal by breaking or closing a circuit to ground. Combining this higher power allows you to amplify the electrical signal in your circuit.

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 circuit?

It's really pretty easy. Here is a schematic and explanation detailing how:



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.

Voltage Dividers

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?

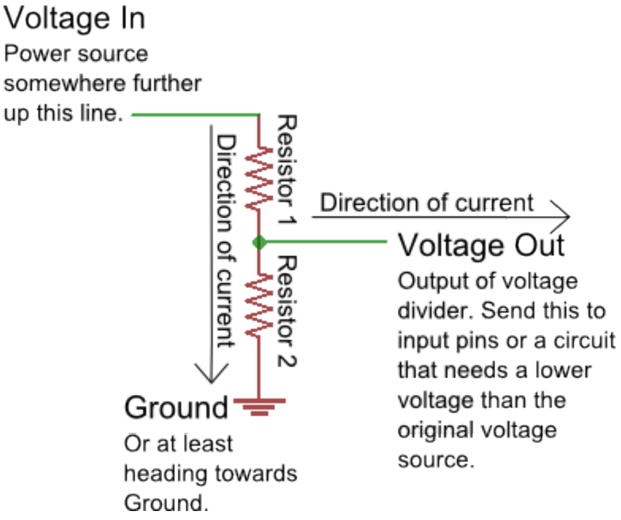
A voltage divider is useful because you can take readings from a circuit that has a voltage beyond the limits of your input pins. By creating a voltage divider you can be sure that you are getting an accurate reading of voltage from a circuit. Voltage dividers are also used to provide an analog Reference signal.

What is in a voltage divider?

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

How do you put together a voltage divider?

It's really pretty easy. Here is a schematic and explanation detailing how:


Voltage In
Power source somewhere further up this line.
Resistor 1
Direction of current
Direction of current
Voltage Out
Output of voltage divider. Send this to input pins or a circuit that needs a lower voltage than the original voltage source.
Resistor 2
Ground
Or at least heading towards Ground.

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 first you 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.

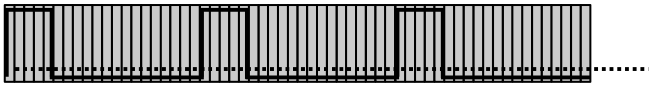
Given three of these values you can always solve for the missing value using a little algebra, making it pretty easy to put together your own voltage divider.

Pulse Width Modulation

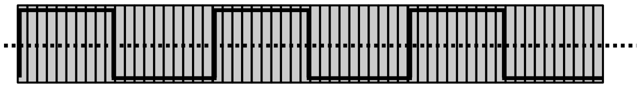
Computers and microprocessors only understand two things, ON and OFF. These are represented in a few different ways. There is ON and OFF, One and Zero, or HIGH and LOW. Ones and Zeros are used in the computer language Binary, HIGH and LOW are used with electricity, ON and OFF are plain old human speak.

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.

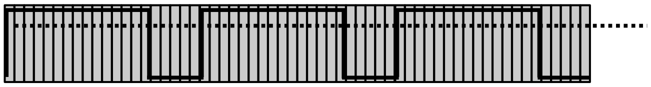
For this reason some of the pins on your Arduino are labeled PWM or Pulse Width Modulation pins. This means you can send a bunch of ones and zeros real quick and the Arduino board will read these ones and zeros as an average somewhere between one and zero. The dotted line in the diagrams represents the average. See tables below.



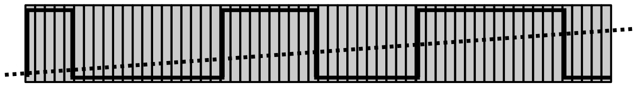
PWM signal at 25%



PWM signal at 50%



PWM signal at 75%



PWM signal rising from 25% to 75 %

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.

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.

To write a PWM value to one of the PWM enabled pins (3, 5, 6, 9, 10, 11) simply use the following code:

```
analogWrite(pin, value);
```

Where pin is one of the PWM enabled pin numbers and value is a value between 0 and 255.

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	Relational operators	Logical operators
+ (addition) **-** (subtraction) ***** (multiplication) **/** (division) **%** (modulus) **=** (assignment)	**==** (equality) **!=** (inequality) **>** (greater-than) **<** (less-than) **>=** (greater-than-or-equal-to) **<=** (less-than-or-equal-to)	**!** (NOT) **&&** (AND) **||** (OR)
Arithmetic operators are your standard mathematical signs (no example provided)	Relational operators are used to compare values and variables	Logical operators are used to join two or more conditional statements together

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

Relational operator example:	Logical 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
	`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

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 programmers 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 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 affect the rest of your code when you compile or upload it.

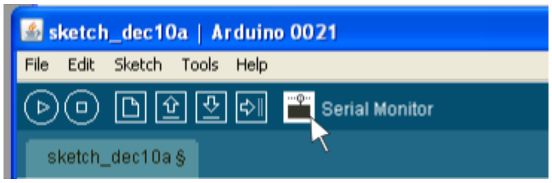
//	/*	*/
This is used comment out a single line	This is used to start a section of commented lines	This is used to end or close a section of commented lines
`//commented out line`	`/*comments start here`	`comments end here*/`

Serial Communication

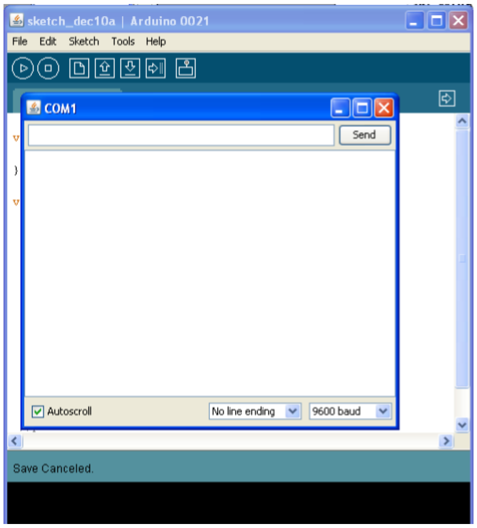
Serial is used to communicate between your computer and the Arduino as well as between Arduino boards and other devices. Serial uses a serial port (makes sense huh?) also known as UART, which stands for universal asynchronous receiver/transmitter to transmit and receive information. In this case the computer outputs Serial Communication via USB while the Arduino 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:

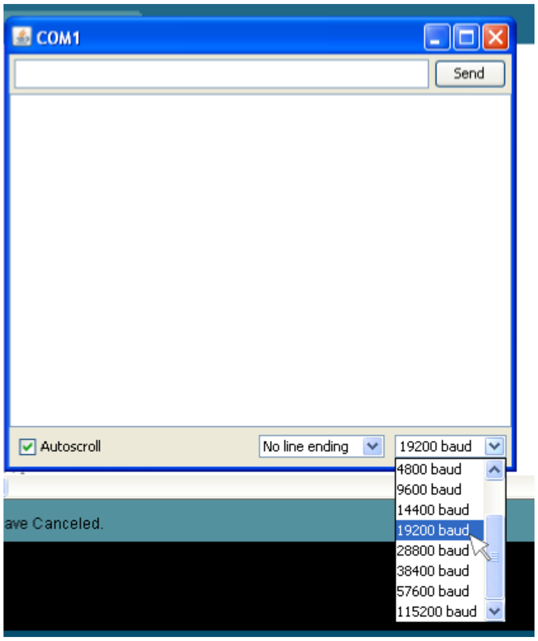


What the activated Serial Monitor looks like:



Save Canceled.

Setting the Serial Monitor baud rate:



ave Canceled.

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.

Serial Communication

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 on the Arduino 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.

Serial Communication

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 Arduino 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.

```
int incomingByte = 0; //declare variable incomingByte, assign it value 0
void setup ( ) {
        Serial.begin(9600); //begin serial communication at baud rate 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
        }
}
```

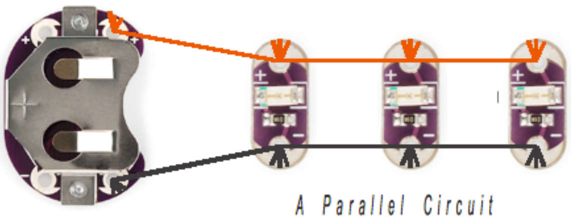
Additional things to note about Serial:

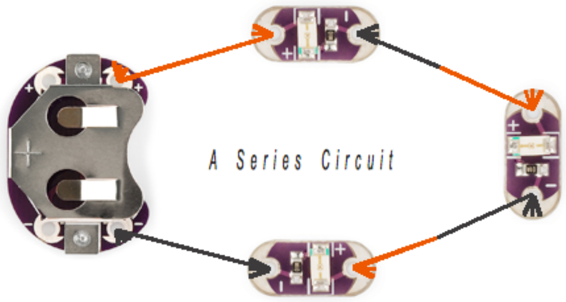
You cannot transmit and receive at the same time using Serial, you must do one or the other. You cannot hook more than two devices up to the same Serial line. In order to communicate between more than two devices you will need to use an Arduino library such as NewSoftSerial.

LilyPad Technology

What are Lilypads? LilyPad is a wearable e-textile technology developed by Leah Buechley and cooperatively designed by Leah and SparkFun. Each LilyPad was creatively designed to have large connecting pads to allow them to be sewn into clothing. Various input, output, power, and sensor boards are available. They're even washable!

Great, how do we use them? Utilizing conductive thread we can connect power to different component boards, but to start we need to cover a few basics of circuit design. In building basic circuits, there are two types of configurations: a parallel and series circuit.





Both have advantages and disadvantages. The largest disadvantage to a series circuit is that you need a very large volt battery to run even a small circuit. Due to this fact, it is recommended that you always use Lilypads in a parallel circuit.

Linking batteries in Series combines the voltages of the batteries to create a power supply with a higher voltage.

Linking batteries in Parallel combines the current of the batteries to create a power supply with a higher Ampage.

LilyPad Battery Life

Before you start it is a good idea to pick all your components and figure out how long a typical battery will run them. For this we will need to do a little, easy math. The first thing to do is to figure out how much current (I) each component will use. In the parallel circuit above, each LED draws about 20 mA (milli-Amps). Since there are three LEDs we need to account for all of them: 20 mA + 20 mA +20 mA = 60 mA. Then we need to find how how many amps the battery holds. For the coin cell battery, this is 250 mAh (milli-Amp hours). This means the battery will run at 250 milli-Amps for one hour. In order to discover how long your battery will run use this equation:

$$\text{Battery run time} = \frac{\text{milli–Amp hour of the battery}}{\text{total milli-Amps of the circuit}}$$

So for the circuit above:

Total milli-Amps of the circuit = 20 mA + 20 mA +20 mA
Total milli-Amps of the circuit = 60 mA
Milli-Amp hour of the battery = 250 mAh
Using these two values in the equation above:
Battery Run time = 250 mAh / 60 mA

Battery Run time = 4.167 hours

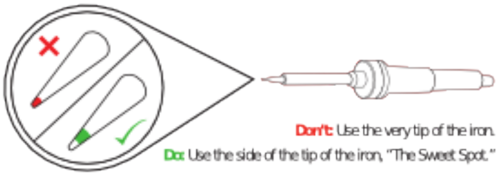
Now we know about how long one coin cell battery can power 3 LEDs. If the battery run time you come up with is not long enough, there are some other options available to you. The Lilypad line has various components designed to allow you to plug in different types of batteries.

Enough science, let's get to the fun part: LilyPad basics
Do not sew any components in with the battery installed. There is no risk of getting hurt, but you might kill the battery.
There are generally two types of thread, thick and thin. The thick type is better if you are going to have LEDs far away from the battery. The thin type will work in a sewing machine but the components cannot be too far away from the battery.
Any time you make a connection between a component and the thread, make a few loops through the connection hole. Also make sure you loop through the portion of the LilyPad components that has metal on it, if the thread is only touching purple PCB it will not conduct.
The thread will melt similar to nylon. It is good to knot the end of your thread at a component and melt the extra, however be careful you do not burn your project.
If you have two threads that need to cross, there must be some fabric between the two, otherwise the circuit will short and not work.
Finally, make sure to clip any long threads at the end of your stitches as these can short out your circuit.

PTH Soldering

SparkFun Electronics Summer Semester

soldering tips



Don't: Use the very tip of the iron.

Do: Use the side of the tip of the iron, "The Sweet Spot."

Do: Touch the iron to the component leg and metal ring at the same time.

Do: While continuing to hold the iron in contact with the leg and metal ring, feed solder into the joint.

Don't: Glob the solder straight onto the iron and try to apply the solder with the iron.

Do: Use a sponge to clean your iron whenever black oxidization builds up on the tip.

soldering tips

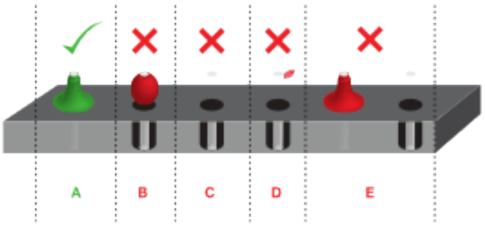
A Solder flows around the leg and fills the hole - forming a volcano-shaped mound of solder.

B Error: Solder balls up on the leg, not connecting the leg to the metal ring.
Solution: Add flux, then touch up with iron.

C Error: Bad Connection (i.e. it doesn't look like a volcano)
Solution: Flux then add solder.

D Error: Bad Connection...and ugly...oh so ugly.
Solution: Flux then add solder.

E Error: Too much solder connecting adjacent legs (aka a solder jumper).
Solution: Wick off excess solder (for tips on this, see pages 18-19).

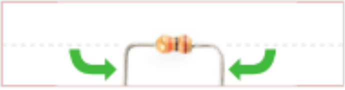


Page 3

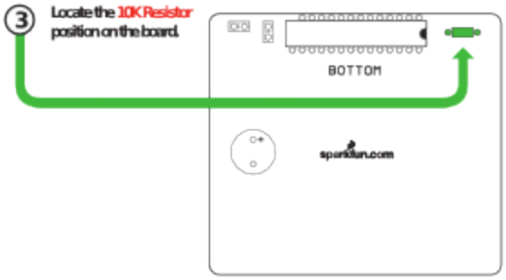
Quickstart - your first component [steps 1 to 11]

① Locate the **10K Resistor.**

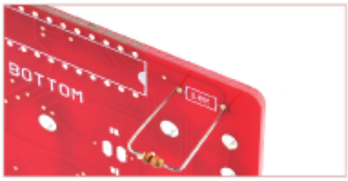
② Bend the legs downward.



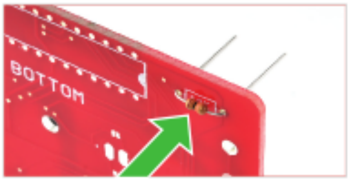
③ Locate the **10K Resistor** position on the board.



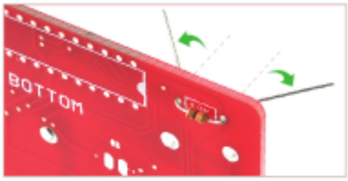
④ Insert the resistor into the PCB.



⑤ Push the resistor in so it is nearly flush with the board.



⑥ Slightly bend the legs outward to hold it in place.



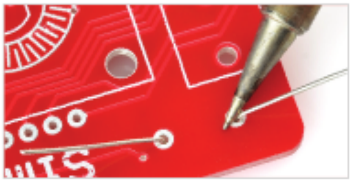
Page 5

PTH Soldering

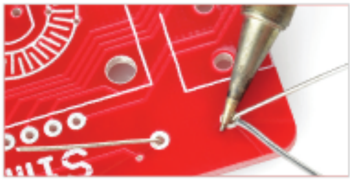
SparkFun Electronics Summer Semester

Quickstart - your first component [steps 1 to 11]

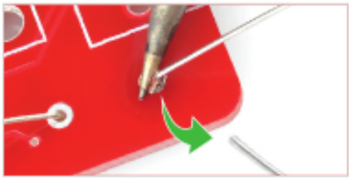
⑦ Flip the board over. Hold the soldering iron's "Sweet Spot" so it touches both the leg and the metal ring. Hold for 2 seconds.



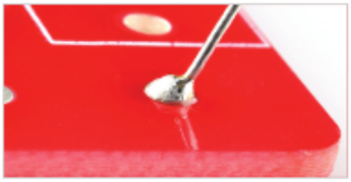
⑧ Feed solder into the joint.



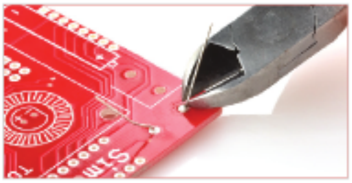
⑨ Pull solder away first.



⑩ Your solder joints should look like this - a tiny volcano.

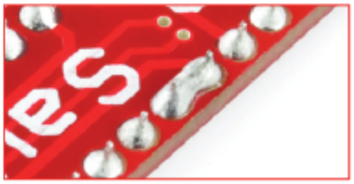


⑪ Clip off any excess leg.



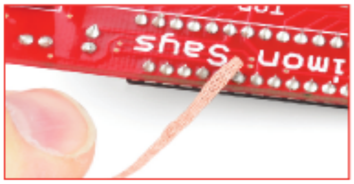
troubleshooting jumpers

Did you accidentally solder a jumper between two legs? Don't fret! Here is a simple process using solder wick to remove the excess solder.

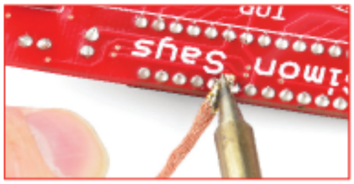


Ⅰ Locate a piece of solder wick.

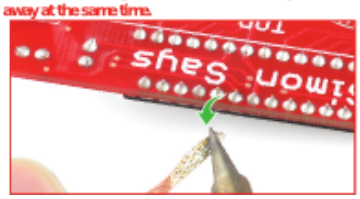
Ⅱ Place solder wick on top of solder.



Ⅲ Place iron on top of solder wick. Hold for 3-4 seconds.



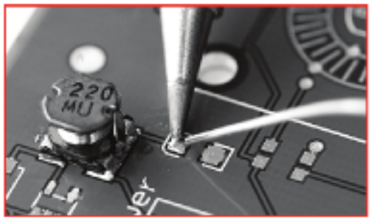
Ⅳ Once the solder begins to flow into the wick, pull the wick and iron away at the same time.



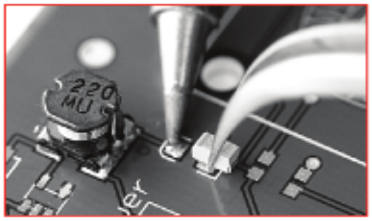
SMD Soldering

SparkFun Electronics Summer Semester

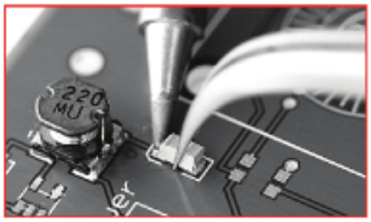
1 Add solder to one pad.



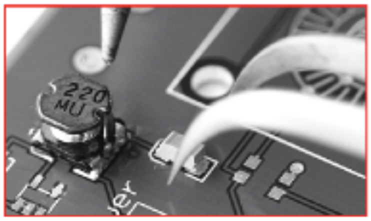
2 While that pad is molten, slide the component into place. Do not push down from the top - slide the component into the blob of solder horizontally.



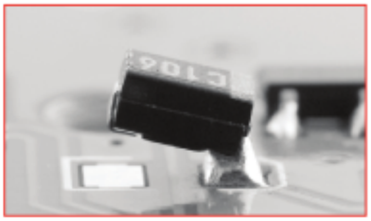
3 Align the component while connection is still molten.



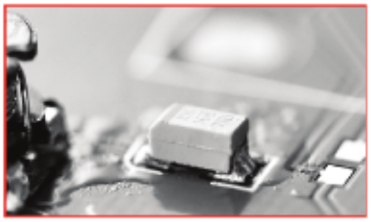
4 Once you have good alignment, continue to hold the component in place, and remove your iron. Continue to hold component for 1-2 seconds while the solder joint solidifies.



5 From above, the alignment looked good. From the side, you can see the rear pad is hovering slightly above the PCB. This can lead to problems on multi-pin components (open connections). Be sure the component is flush up against the PCB before soldering more connections. Re-grip the component, re-heat pad 1 and push the component flush against the PCB.



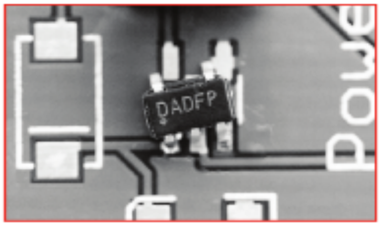
6 This is how a tantalum capacitor should look after making both solder connections.



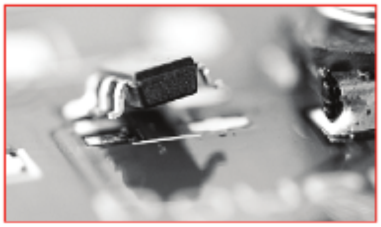
SMD Soldering

SparkFun Electronics Summer Semester

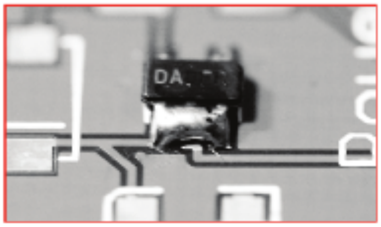
7 If alignment is not good, do not solder more than 1 pad! Re-heat the joint, re-adjust component until aligned correctly, then move on to soldering other connections.



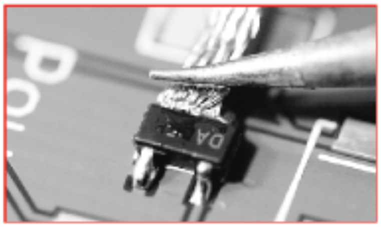
8 This is bad. It would be nearly impossible to finish the connections on this part. Make sure you have the component flush against the PCB.



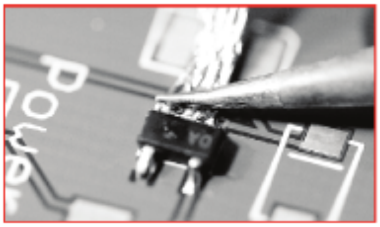
9 If you solder multiple pins together, don't worry about it! It can be easily fixed. Do not worry about jumpers! There are actually three pins under that blob.



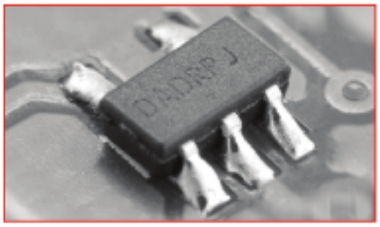
10 Pull out some solder wick. Put a small amount of solder on the end of your iron (this will transfer heat from iron to wick to the jumper). Sandwich the wick in between the iron and the solder jumper.



11 Hold still for 2-3 seconds. You will see solder start to flow up the wick. Once the excess solder has flowed into the wick, carefully lift up the wick and your iron in one fluid motion.



12 Nice and clean!



Intro to Processing / Tweaking Simon

SparkFun Electronics Summer Semester

About Processing

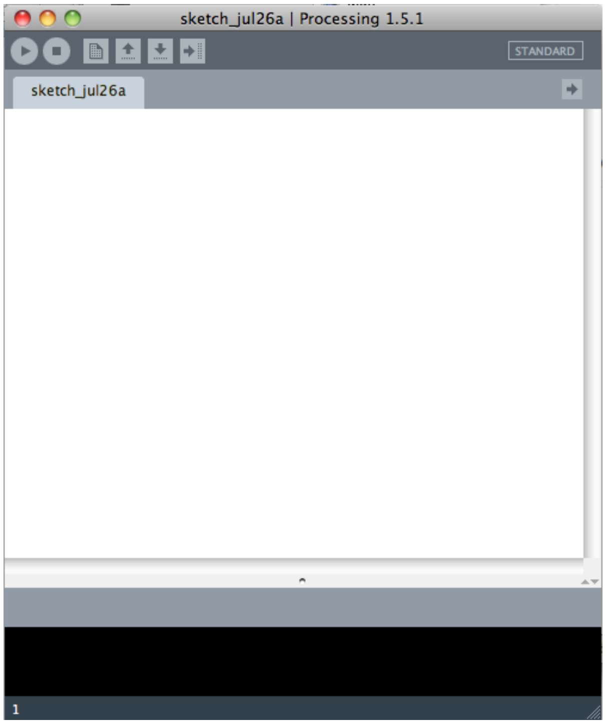
Processing is a free, open source, cross-platform programming language and environment for people who want to create images, animations, and interactions. It's easy to get started, and more importantly, it can serve as an easy way to add a visual component or interaction with your hardware project (visualizing sensor data, playing a game, displaying video, etc.).

Setup

- Download and install the latest version of Processing from http://processing.org (that's it!)

Getting Started

- Open up Processing. You should see something like the window below:



The IDE (Integrated Development Environment, i.e. what all the buttons do)

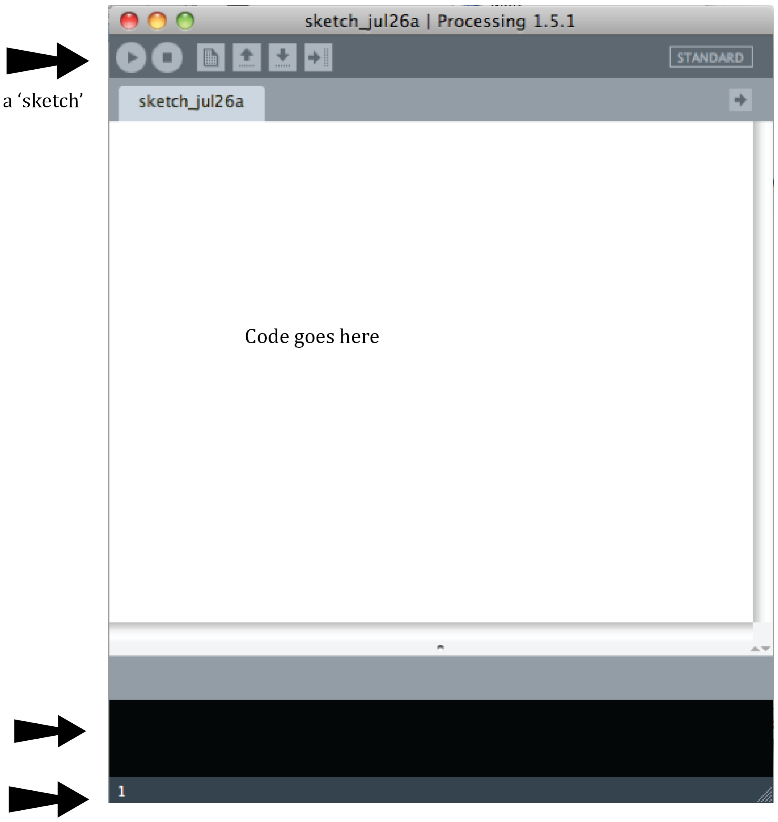
Intro to Processing / Tweaking Simon

SparkFun Electronics Summer Semester

Sketch Name | Processing Version

Controls (See below)

Each program is called a 'sketch'



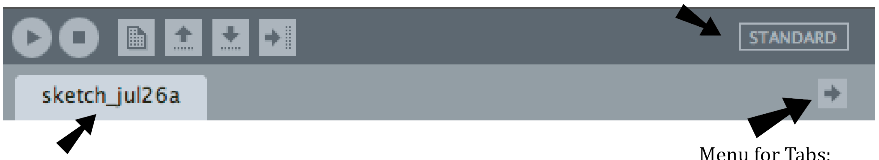
Code goes here

Error messages show up here

Line number

Play (Run) Stop New Open Save Export (As Applet)

Toggle between Standard and Android mode



Sketch Name/ Active Tab

Menu for Tabs:
new tab, delete,
rename, select

Your First Sketch

- In the programming area, type:

```
print("hello world!");
```

- Press the 'run' button. Voila! You should see the message 'hello world!' in the output area of the IDE. You will also notice that a small grey box popped up – that's because Processing is primarily used as a visual tool, and so it comes built in with a canvas ready to draw on.

Drawing

- Drawing simple shapes in Processing is very straightforward. For example, typing:

```
ellipse(40, 50, 100, 110);
```

Gives us an ellipse with an x coordinate of 40, a y coordinate of 50, a width of 100, and a height of 110. Notice how the size goes off the canvas – we'll get to that in a sec.

Processing has a ton of built-in methods and functions - it would take us way too long to cover them all – but the full reference can be found at:
http://processing.org/reference/

Let's get serious

While very simple Processing sketches don't require much in the way of format, for anything dynamic (including motion, repetition, reading data, etc.) you'll want to set up your processing sketch with a certain structure.

To help with this, Processing has a few reserved functions (much like Arduino): setup() and draw()

As you might imagine, we do our setup within the setup() function and our drawing within the draw() function. For practical purposes this means that setup() only gets called once at the beginning of the sketch, and that draw() is called repeatedly, much like the loop() function in Arduino.

In addition, things like import statements for libraries and global variables typically go before the setup function. Custom functions, class declarations, and other reserved methods can go after the draw function or in a separate tab, depending on your preference. (We'll talk more about this later).

Bouncy Ball

In this example, we're going to see the basic structure of a Processing program set up to display a bouncing ball.

• In Processing, navigate to the top menu under File -> Examples. A pop-up window (or drop-down list) should appear. Go to Topics -> Motion -> Bounce and open the sketch.

• Go ahead and run it. You should see a white ball 'bouncing' off the edges of the screen. Here's the code if you can't find it:

```
int size = 60;      // Width of the shape
float xpos, ypos;    // Starting position of shape

float xspeed = 2.8;  // Speed of the shape
float yspeed = 2.2;  // Speed of the shape

int xdirection = 1;  // Left or Right
int ydirection = 1;  // Top to Bottom

void setup()
{
  size(640, 200);
  noStroke();
  frameRate(30);
  smooth();
  // Set the starting position of the shape
  xpos = width/2;
  ypos = height/2;
}

void draw()
{
  background(102);
  // Update the position of the shape
  xpos = xpos + ( xspeed * xdirection );
  ypos = ypos + ( yspeed * ydirection );

  // Test to see if the shape exceeds the boundaries of the screen
  // If it does, reverse its direction by multiplying by -1
  if (xpos > width-size || xpos < 0) {
    xdirection *= -1;
  }
  if (ypos > height-size || ypos < 0) {
    ydirection *= -1;
  }

  // Draw the shape
  ellipse(xpos+size/2, ypos+size/2, size, size);
}
```

A few things worth noticing / doing:

• Notice how setup() and draw() are used and where they're placed, look at how global variables are used

• Look at some of the common built-in functions: size(), background(), width, height, frameRate(), smooth()

• Play with the size, speed, and direction variables and see what happens

• Move the background() command to the setup() function. What happens? Why?

• Change colors using stroke(), fill(), and background()

• Remove smooth() – what happens?

Hacking Simon!

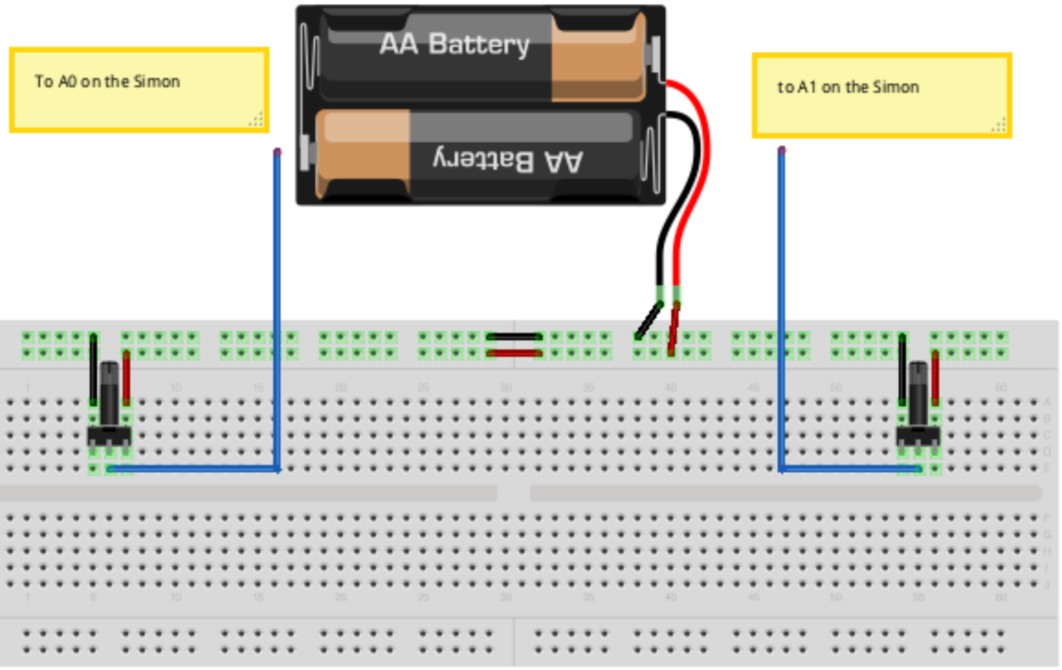
Part 1: Hardware Setup

- If you haven't already, solder in some female headers to the breakout pins on your Simon.

- Next, get your breadboard and run power and ground from the VCC and GND pins on the Simon to the breadboard.

- On your breadboard, set up 2 trimpots, somewhat far apart, and run them to pins A0 and A1 on the Simon. Make sure you run power and ground to your trimpots as well.

A little diagram: Just think of the batteries as your power and ground from the Simon board. You'll want to space out the two potentiometers as much as possible, so that it's easier to turn both at once.



Part 2: Arduino

Now that you're hardware is setup, open a new sketch in Arduino, and paste in the SimonSketch code:

```
/* Simon Sketch - A Simon Tweak from SparkFun */

//define pins for led's and buttons
#define blueLed 13
#define yellowLed 3
#define redLed 5
#define greenLed 10

#define blueButton 12
#define yellowButton 2
#define redButton 6
#define greenButton 9

int leftPot = A0;
int rightPot = A1;


int buttonState;  //variable to detect button press
int numButtons = 4;  //number of buttons
int buttons[] = {  //put our buttons in an array
  blueButton, yellowButton, redButton, greenButton};
int leds[] = { //put our led's in an array
  blueLed, yellowLed, redLed, greenLed};

void setup() {

  //init our pins - input for buttons, output for led's
  for(int i = 0; i < numButtons; i++) {
    pinMode(buttons[i], INPUT);
    pinMode(leds[i], OUTPUT);
    digitalWrite(buttons[i], HIGH);  //init internal pull-up on button pins
  }

  Serial.begin(9600);  //begin serial communication

  //UnComment this line after you configure your button pins
  //establishContact();
}
```

```
void loop() {

//  UnComment these lines after you configure your button pins
//  if (Serial.available() > 0) {
//
//    int inByte = Serial.read();

  //send trimpot values
  int leftPotVal = analogRead(leftPot);
  Serial.print(leftPotVal, DEC);
  Serial.print(";");
  int rightPotVal = analogRead(rightPot);
  Serial.print(rightPotVal, DEC);

  //read buttons
  for(int i = 0; i < numButtons; i ++) {

   //is there a press?
   buttonState = digitalRead(buttons[i]);

   //if so, light up the corresponding led, and send the value to
   // processing via serial
   if (buttonState == 0) {
    digitalWrite(leds[i], HIGH);
    Serial.print(";");
    Serial.print(buttons[i]);
    delay(100);
    digitalWrite(leds[i], LOW);
   }
  }
  Serial.print('\n');

 }
//UnComment this line after you configure your button pins
//}

void establishContact() {
 while (Serial.available() <= 0) {
  Serial.println("hello");   // send a starting message
  delay(300);
 }
}
```

- Next, get your FTDI breakout connected to your computer and to the Simon Board. You may need some male or female headers to do this.

- When plugging in the FTDI breakout to your Simon board, make sure the BLK and GRN markings on the FTDI breakout match up with those on the Simon.

- In Arduino, select 'LilyPad Arduino w/ATmega 328' as your board type.

- Upload the code!

- **IMPORTANT**: You will have to configure your button and LED pins at the top of the sketch to match the colors they are assigned to. (e.g., blueLed should be with blue button and actually make the color blue when pressed). You can do this by opening the Serial Monitor from Arduino and looking at the number that shows up when you press each button. This is the button pin number that should be associated with the color of that LED.

- Note the formatting we do around printing out the sensor values – this is going to help us separate them back out in Processing.

- After you've configured your button pins, Un-comment the lines of code that say 'uncomment this line after you've configured your buttons'. There are 3 places where you have to do this (around line 38, lines 44-46, and line 74).

- Re-upload your code to the Simon.

- If you re-uploaded the Arduino code successfully, open up the Serial Monitor – what do you see? Why?

- If you see 'hello' over and over again, you're done! Congrats!

Part 3: Interlude – Getting serial data into Processing

We're going to walk through the basics of setting up Serial communication in Processing. You're going to want to do most of these steps any time you want to communicate with Processing using the serial port, especially for things like reading in sensor values from an Arduino.

• First, open Processing. (Duh.) Start a new sketch. Call it 'SerialBasic', or whatever you prefer.

• Under the 'Sketch' menu in Processing, go to 'Import Library' and select 'Serial I/O'. You should see something like: import processing.serial.*; in your sketch now. Good job! You just imported your first library. This allows us to make use of some Serial communication commands that will make our job much easier.

• Continue by copying the rest of the program below – make sure to read the comments so you understand what each line is for.

```
/* SerialBasic w/ handshake -
 * Prints out a set of sensor readings from Arduino using the Serial Library
 */

//import the Serial library – should be there already
import processing.serial.*;

Serial myPort;  //the Serial port object

// since we're doing serial handshaking,
// we need to check if we've heard from the microcontroller
boolean firstContact = false;


void setup() {

  //  initialize your serial port:
  //  this code picks the first port in the array of available ports,
  //   and sets the baud rate to 9600
  myPort = new Serial(this, Serial.list()[0], 9600);
  myPort.bufferUntil('\n'); //buffer until we get a carriage return
}
```

```
void draw() {
  //we can leave the draw method empty,
  //because all our programming happens in the SerialEvent (see below)
}

//the serialEvent method is called every time we get new stuff in on the
// serial port
//in this case we're constantly getting info, so it acts as our draw loop
void serialEvent( Serial myPort) {

  //put the incoming data into a String -
  //the '\n' is our end delimiter indicating the end of a complete packet
  String myString = myPort.readStringUntil('\n');

  //make sure our data isn't empty before continuing
  if (myString != null) {

    //trim whitespace and formatting characters (like carriage return)
    myString = trim(myString);
    //println(myString);

    //look for our 'hello' string to start the handshake
    //if it's there, clear the buffer, and send a request for data
    if (firstContact == false) {
      if (myString.equals("hello")) {
        myPort.clear();
        firstContact = true;
        myPort.write('A');
        println("contact");
      }
    }
    else { //if we've already got contact, keep getting and parsing data

      //split the string of data back into separate values, using the
      //semicolon we printed in Arduino
      int sensors[] = int(split(myString, ';'));
      //run through the sensor values and print them out
      for (int sensorNum = 0; sensorNum < sensors.length; sensorNum++) {
        println("Sensor " + sensorNum + ": " + sensors[sensorNum]);
      }
      // when you've parsed the data you have, ask for more:
      myPort.write("A");
    }
  }
}
```

- Plug in your Simon and run your Processing sketch, if there are no errors, you should see the sensor values from your Simon being printed out in the Processing terminal. Sweet!

- If not, check for syntax errors, make sure you imported the serial library, hooked up your Simon properly, and don't have any other serial monitors open.

- Setting up Serial communication in this way in called a 'handshake' – Arduino waits for an incoming byte from Processing before it starts sending data, and waits to send more data until another byte from Processing arrives (signaling that Processing is done with the earlier data). This helps control the flow of data between the two programs.

Part 4: SimonSketch!

At this point, we've got our Simon hooked up to some sensors, and we can read the values of those sensors in Processing. Now it's time to use these values in Processing to create a little thing we call the SimonSketch (think Simon meets Etch-A-Sketch).

1. Copy and Paste your SerialBasic code into a new sketch, called 'SimonSketch'.

2. What does an etch-a-sketch do? It draws, and it moves using two dials, right? So we need to keep track of our position, using info from our two dials (our trim pots). Let's make global variables for our pots and the x and y coordinate of where we're currently drawing.

Add these lines below your 'Serial myPort;' declaration:

```
int leftPot;
int rightPot;

float x = width/2;
float y = height/2;
```

This gives us a place to store the values from the pots, as well as an x and y for the position of our cursor.

3. At the beginning of your Setup() function, we're going to put a few methods to 'setup' our drawing environment.

```
size(1000, 750); //defines the size of our canvas (width, height)
background(255); //defines the starting background color (255 is white)
stroke(255); //defines the starting stroke color
smooth(); //the smooth() function 'smoothes out' motion and curves
```

4. Although we will be drawing on the canvas, we can still leave the draw() method empty and put all the actual drawing in the serialEvent method.

5. Skip down to the section in the serialEvent method after we print out the sensor values (around line 70, shown below). We're going to assign the sensor values from Arduino to our local leftPot and rightPot variables:

```
for (int sensorNum = 0; sensorNum < sensors.length; sensorNum++) {
        println("Sensor " + sensorNum + ": " + sensors[sensorNum]);
        println(sensors.length);
      }

      leftPot = sensors[0];  // <- new stuff
      rightPot = sensors[1];
```

6. Now that we've got the sensor values in their own variables, we've got to turn them into numbers we want to use. In this case it means casting the integers as floats so that we can 'map' them to the width and height of the canvas – this is just an easy way of taking the normal range of sensor values (0 to 1023) and mapping them in a way that makes sure they cover the whole canvas evenly (since our canvas might be bigger or smaller than 1023 pixels squared).

Add the following directly below the lines you just wrote:

```
y = (float)leftPot; //casting values from int to float
x = (float)rightPot;

y = map(y, 0, 1023, 0, height); //map to height and width of canvas
x = map(x, 0, 1023, 0, width);

//print out the new values for good measure
println("X: " + x + " " + "Y: " + y);
```

7. Great. We have sensor values that map to the height and width of our canvas, so now we just have to draw a point (or a small circle) wherever our sensors tell us to go:

```
ellipse(x, y, 5, 5);
```

Yup. Simple as that.

8. Next, we're going to add in some functionality for changing colors based on the button pushes from Simon, as well as an 'erase' mode if we push two buttons at once. To do this, we can just check the size of the sensor value array – if we get 3 values, the third corresponds to the button we should change colors to. If we can 4 values, then 2 buttons are being pushed at once, which is our cue to go into 'erase' mode.

```
//if there's a third value, a button has been pressed, so change the color

if (sensors.length > 2) {
  int colorC = sensors[2];
  changeColor(colorC);
//this is passing the value to our 'changeColor' function (see below)
}

if (sensors.length > 3) {  //if you press 2 buttons, set to erase mode
        fill(255);
        stroke(255);
}
```

Before this code will work – we need to write our changeColor method that we use above. Put this after your serialEvent method at the end of your sketch:

```
void changeColor(int colorC) {
  //match the color to the button press
  switch (colorC) {
  case 2: //yellow
    stroke(255, 255, 0);
    fill(255, 255, 0);
    break;
  case 6: //red
    stroke(255, 0, 0);
    fill(255, 0, 0);
    break;
  case 9: //green
    stroke(0, 255, 0);
    fill(0, 255, 0);
    break;
  case 12: //blue
    stroke(0, 0, 255);
    fill(0, 0, 255);
    break;
  default:
    stroke(0, 0, 255);
  }
}
```

That's it! You've hacked your Simon! Try running your sketch. You'll have to push a button to get drawing, then control the drawing with the two potentiometers, just like an Etch-A-Sketch. Try changing colors using the buttons, and make sure your erase mode works. Get painting!

Here's the full sketch in case you get stumped:

```
/* SimonSketch - A Simon Tweak from Sparkfun
 * Use the buttons on your simon to pick colors,
 * the trimpots to move your cursor
 * 2 = yellow
 * 6 = red
 * 9 = green
 * 12 = blue
 * incoming string = leftPot;rightPot;buttonPress
 * if no button press, then just leftPot;rightPot
 */

import processing.serial.*;

Serial myPort;
int leftPot;
int rightPot;

float x = width/2;
float y = height/2;

// Whether we've heard from the microcontroller
boolean firstContact = false;
```

```
void setup() {

  size(1000, 750);
  background(255);
  stroke(255);
  smooth();
  println(Serial.list()); //list serial ports
  //init serial object (picks 1st port available)
  myPort = new Serial(this, Serial.list()[0], 9600);
  myPort.clear();
}

void draw() {
}

void serialEvent(Serial myPort) {

  String myString = myPort.readStringUntil('\n');
  // if you got any bytes other than the linefeed:
  if (myString != null) {

    myString = trim(myString);
    //println(myString);

    if (firstContact == false) {
      if (myString.equals("hello")) {
        myPort.clear();
        firstContact = true;
        myPort.write('A');
        println("contact");
      }
    }
    else {
      int sensors[] = int(split(myString, ';'));
      // print out the values you got:
      for (int sensorNum = 0; sensorNum < sensors.length; sensorNum++){
        println("Sensor " + sensorNum + ": " + sensors[sensorNum]);
        println(sensors.length);
      }

      leftPot = sensors[0];
      rightPot = sensors[1];

      y = (float)leftPot;
      x = (float)rightPot;

      y = map(y, 0, 1023, 0, height);
      x = map(x, 0, 1023, 0, width);
      println("X: " + x + " " + "Y: " + y);

      ellipse(x, y, 5, 5);
//if there's a third value, a button has been pressed, change the color
      if (sensors.length > 2) {
        int colorC = sensors[2];
        changeColor(colorC);
      }

      if (sensors.length > 3) { //got 2 buttons, set to erase mode
        fill(255);
        stroke(255);
      }
      // when you've parsed the data you have, ask for more:
      myPort.write("A");
    }
  }
}

void changeColor(int colorC) {
  //match the color to the button press
  switch (colorC) {
  case 2: //yellow
    stroke(255, 255, 0);
    fill(255, 255, 0);
    break;
  case 6: //red
    stroke(255, 0, 0);
    fill(255, 0, 0);
    break;
  case 9: //green
    stroke(0, 255, 0);
    fill(0, 255, 0);
    break;
  case 12: //blue
    stroke(0, 0, 255);
    fill(0, 0, 255);
    break;
  default:
    stroke(0, 0, 255);
  }
}
```

Going Further

The Simon has 6 Analog Pins broken out for us to use, so why stop after only using 2? Let’s push it a little further. In addition, we’re going to write a class called *SimonCursor* to control how our cursor behaves while we draw.

The following is just one example, but feel free to improve and improvise using other components.

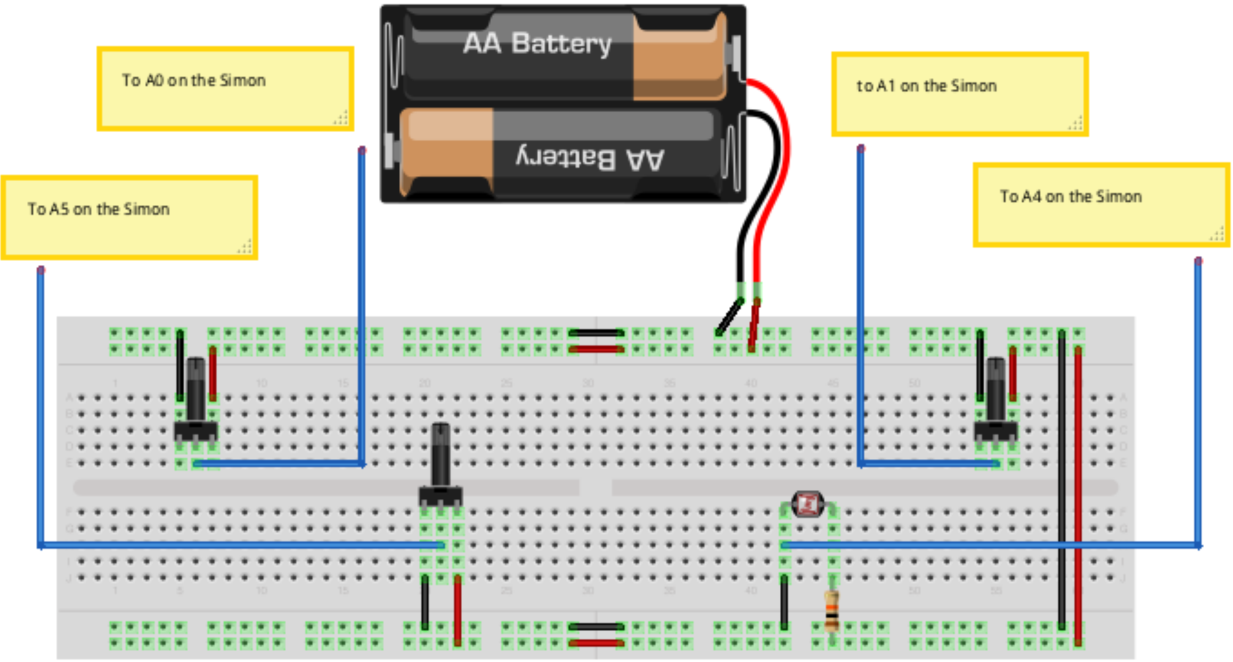
SimonSketchPro with Size and Transparency Control

Step 1: Hardware

• A another Trim Pot to your board, and connect it to Pin A5 on the Simon

• Add a photo-resistor to your board (don’t forget the 10K resistor), and connect it to Pin A4 on the Simon

It should look something like this when you’re done:



Step 2: Arduino

Since we added more sensors, we're going to have to change our Arduino code to read those sensors and send them out to Processing.

1. Copy and Paste your 'SimonSketch' code into a new sketch called 'SimonSketchPro' (or whatever you like).

2. Add these lines at the beginning of your sketch for Pin declarations (new code is **Bold**):

```
int leftPot = A0;
int rightPot = A1;
int lightSensor = A4;
int sizePot = A5;
```

3. In our loop() method, we'll have to read, format, and print the new sensor values:

```
int leftPotVal = analogRead(leftPot);
Serial.print(leftPotVal, DEC);
Serial.print(";");
int rightPotVal = analogRead(rightPot);
Serial.print(rightPotVal, DEC);

Serial.print(";");   // <- new
int sizePotVal = analogRead(sizePot);
Serial.print(sizePotVal, DEC);
Serial.print(";");
int lightSensorVal = analogRead(lightSensor);
Serial.print(lightSensorVal, DEC);
```

4. Before you upload, comment out the lines for the handshake and check your serial monitor to make sure the sensor values are being sent correctly.

IMPORTANT: Every photocell is a little different and gets different values depending on the conditions (sunny vs. cloudy, indoor vs. outdoor, etc.). As you look at the serial monitor, make a note of the high and low values you're getting – we'll need this later in the Processing sketch.

Make sure to uncomment the handshake code and re-upload your code before moving on.

That's it for the Arduino part! On to our Processing code!

Step 3: Processing

Now we have to change a few things in our Processing code. We'll have the extra trim pot control the size of our stylus, and the photocell will control the transparency.

1. Copy and Paste your 'SimonSketch' Processing code into a new sketch called 'SimonSketchPro' (or whatever you like).

2. Let's start writing our class. You can start a class at the bottom of your current code, but you may find it cleaner to create a new tab (under the tab menu) and name it after your class, in this case SimonCursor.

3. In this case, our class is going to take care of the values from our four sensors to control the X position, Y position, size of the cursor, and its transparency. So let's declare our class and its attributes:

```
class SimonCursor {

 float x;
 float y;
 float siz;
 float light;

}
```

4. After the 'float light;' line, we need to add a *constructor*. This is a statement that prepares the object for use upon its creation, by setting up the parameters that align with the variables we just declared. Basically it's a method with the same name as your class, and a dummy variable for each attribute your object possesses. It may look a little strange, but it's necessary:

```
SimonCursor (float _x, float _y, float _siz, float _light) {
  x = _x;
  y = _y;
  siz = _siz;
  light = _light;
 }
```

Notice the underscores – these differentiate the variables in our constructor method from those in our class (even though we're assigning them to each other).

5. Next we're going to crate a render() method to draw our cursor with the position and size that we've passed in:

```
  void render() {
    ellipse(x,y,siz,siz);
  }
```

6. Finally, we're going to move our entire changeColor method into our new class – just copy and paste the entire method in, after the render() method. Don't forget the last curly bracket (}) to close out our class.

7. We're done with writing our class – now we need to go back into the main Processing sketch and change our code to make use of our new class and to read in the new values from our two new sensors.

8. Back in our SimonSketchPro tab,we need to declare our SimonCursor object from our new class and create a few more global variables for our new sensor values coming in (new code in **bold**):

```
SimonCursor myCursor;
int leftPot;
int rightPot;
float sizePot = 0;
int lightSensor = 0;
```

9. We also need to define variables to keep track of the new dimensions we want to control: size and light (new code in **bold**):

```
float x = width/2;
float y = height/2;
float siz; // (size is a protected term in processing, hence 'siz')
float light;
```

10. Now's the time to put in the high and low values you recorded from your photocell:

```
// our high and low photocell values
int high = 550;
int low = 150;
```

11. Next, skip down to your serialEvent() method. We'll need to find a place for the extra sensor values coming in (new code in **bold**):

```
leftPot = sensors[0];
rightPot = sensors[1];
sizePot = sensors[2];
lightSensor = sensors[3];
```

12. As before, we'll have to convert the sensor values to floats so we can map them to the range of values we want (new code in **bold**):

```
y = (float)leftPot;
x = (float)rightPot;
siz = (float)sizePot;
light =(float)lightSensor;
y = map(y, 0, 1023, 0, height);
x = map(x, 0, 1023, 0, width);
siz = map(siz, 0, 1023, 2, 50);
light = map(light, low, high, 0, 255); //note the high and low values
println("X: " + x + " " + "Y: " + y + " " + "Size: " + siz + " " + "Light: "
+ light);
```

13. Now we get to make use of our new cursor object – we'll create a new SimonCursor object called 'myCursor', and pass it in all the variables it expects (x,y,size, light):

```
myCursor = new SimonCursor(x,y,siz,light);
```

14. We've also got to tell it to draw itself now that it has all the values it needs:

```
myCursor.render();
```

15. Since our normal sensor array size is going to be bigger (always reading 4 sensors instead of 2), we have to adjust our code for determining when to change colors and when to enter erase mode – and since we moved our changeColor method into our new class we've got to call that method as part of our myCursor object:

```
if (sensors.length > 4) {
      int colorC = sensors[4];
      myCursor.changeColor(colorC);
}

if (sensors.length > 5) {  //if you press 2 buttons, set to erase mode
      fill(255);
      stroke(255);
}
```

9. The last change is incorporating the photocell value into our changeColor() method back in our Class tab, to control the alpha value (transparency) of our colors (new code in **Bold**):

```
void changeColor(int colorC) {
  //match the color to the button press
  switch (colorC) {
  case 2: //yellow
    stroke(255, 255, 0, light);
    fill(255, 255, 0, light);
    break;
  case 6: //red
    stroke(255, 0, 0, light);
    fill(255, 0, 0, light);
    break;
  case 9: //green
    stroke(0, 255, 0, light);
    fill(0, 255, 0, light);
    break;
  case 12: //blue
    stroke(0, 0, 255, light);
    fill(0, 0, 255, light);
    break;
  default:
    stroke(0, 0, 255, light);
  }
}
```

You're finished! Try running your new Simon Sketch code – note that the more you cover the photocell when changing colors (that's when we send the photocell value to the changeColor method), the more transparent your drawing will be.

Creating a Schematic

Step-by-Step Guide to creating a USB to Serial Interface using the FT232

1. With Eagle open, go to File->New and select Project. Give the project a name, but no version number: USB Converter
2. Right click on the USB Converter project (now located in the projects folder) and select New->Schematic.
3. Click the 'Add' button in the toolbar and then find the part named “FRAME-LETTER” in the list of parts. (Tip: The 'search' box in the bottom of the parts window doesn't work very well. It's better to manually find the part.)
4. Double click the part to select it. This will take you back to the schematic screen. Place the screen at the origin. There's a dotted cross-hair that identifies the origin.
5. Save the schematic. Be smart about saving schematics; use the project name followed by a version number. For this schematic use the name 'USB Converter-v10.' That stands for major version 1, minor version 0.
6. Now add all of the parts that are in the reference design. Normally you have to search through the parts to find what you need, and after a while you become accustomed to the available parts. Here's what you'll need for this project though:

Part Name	Quantity
FT232RLSSOP	1
INDUCTOR (0603)	1
CAP (0603)	4
USB (USBPTH)	1

*Tip: You can press the Escape key after adding your part to the schematic to go back to the Parts List

7. Use the 'Move' command (Shortcut Key 'F7') to arrange the components on the schematic so that it looks similar to the sample circuit in the datasheet.

8. Assign values to the components that require them (caps and ferrite bead). Use the 'Value' command to do this (Shortcut Key 'F5'). Use the values from the sample circuit in the datasheet.
9. Check out the name of the inductor: U$1. This is just ugly! The proper prefix for an inductor is the letter 'L.' Use the 'Name' command (Shortcut Key 'Alt+F5') to rename the inductor L1.
10. Some of the labels for the part names and values may not be optimally placed; perhaps they overlap each other or you just don't like their locations. We can move the labels indepently of the part by first 'Smashing' the labels from the part, then 'Moving' the labels. Use the 'Smash' command (Shortcut Key 'F6') to smash the labels of all the parts; after smashing a part you'll notice that a crosshair appears on each label. Use the 'Move' command and move the labels by clicking and dragging their corresponding crosshair.
11. Add Power and Ground to the schematic by finding and adding the part named 'VCC' and the part named 'GND' to the schematic.
12. In a schematic we connect parts together using 'Nets' (think of them as wires). Use the 'Net' command (Shortcute Key 'F9') to connect the VBUS pin of the USB connector to one end of the Ferrite Bead.
    1. Start a net at the endpoint of a pin by clicking the pin. Pins are red!
    2. While running a Net from one pin to another you will have to add in some angles to the Net to reach from place to place. Only use Right Angles. You can change the direction of the angle by holding 'Ctrl' and clicking the right mouse button while running the trace.
    3. End a Net by clicking on either another net or another pin.
13. Before moving on to the next net, check to make sure that the net has been connected properly by using the 'Show' command (Shortcut Key 'F1'). Select the 'Show' command and then click the Net; the entire signal will be highlighted. Make sure the green wire and the red pins of the signal are highlighted. If the pins aren't highlighted, the net is not connected to them.

14. Now connect the opposite end of the Ferrite Bead to the VCC pin of the FT232 part using a net. Also connect the VCC part that was added to the schematic to this net.
    1. You will create two nets that will merge into one signal. When the Net is added from the bead to the FT232 module it will be just like the first net you added.
    2. Next run a net from VCC to the net you just created. A couple things might happen, first you will be given a prompt that reads "Merge segment net N$X into supply net VCC?" This is Eagles way of verifying that you want to connect two signals. Click 'Yes' to accept.
    3. Now the Net should end; however when you move your mouse the net may still be going. Press Escape to end the net.
        1. Go to Options in the menu bar, and open the Set... window. Go to the 'Misc' tab and enable the option 'Auto end net and bus.' This will stop a net after you've connected it to a new signal.
    4. Now there should be a 'T' shaped signal that connects the VCC to the FT232 module, the Ferrite Bead, and the VCC part. We want a better indicator that the signals are connected at the intersection of the 'T,' so we'll add a Junction. Use the Junction command (Find the Dot in the Toolbar) and add a junction at the intersection of the T.
        1. Go to Options in the menu bar and open the Set... window. Go to the 'Misc' tab and enable the option 'Auto set junction.' This will automatically create junctions when signals are merged.
    5. Verify the new VCC net by using the 'Show' command and make sure the proper signals are all highlighted.
15. Run a net from the VCCIO pin to a pin on one of the 100 nF capacitors (if the capacitor isn't nearby, move one so that it is).
    1. Oops! We connected the wrong pin. Check the sample circuit again, it is actually the 3V3OUT pin that needs to be connected to the capacitor. Use the 'Delete' command (Shortcut Key 'F3') to delete the net; each segment of the net must be deleted individually.
    2. Now add the correct net from the 3V3OUT pin to the capacitor and double check it with the 'Show' command.
16. Move the GND part under the capacitor that was just connected. Connect the opposite end of this capacitor to the GND part.
17. Run a net from the VCCIO pin to the VCC net created earlier and double check the connection.
18. Move the 10 nF capacitor so that it is located beneath and to the right of the USB part. Connect one end of the Capacitor to the VBUS net created earlier.

19. The other end of the capacitor needs to be connected to ground, but there's a better way to do this than to drag a net all the way over to our existing ground. A Net doesn't actually have to be *physically* connected in the schematic to connect two parts. If nets that aren't connected have the same *name* then they will be connected in the design. So we just need another net with the same name.
    1. We can do this three different ways: add another GND part from the parts list, copy the existing GND part and place the new one where we want it, or just add a 'stub' net to the end of the capacitor and name it GND.
    2. Start by adding a net to the end of the capacitor; drag it down from the end a bit and then click add the stub. Press Escape to end the net without connecting it to anything.
    3. Use the 'Name' command (Shortcut Key 'Alt+F5') and click the stub to change the name. Assign the name GND and click OK. You'll be prompted to check if you really want to connect the new net to GND, confirm this.
    4. Now use the 'Show' command and select your GND stub; notice that the other GND net that was created earlier is also highlighted. This verifies that the two signals are connected in the design even though we don't see a physical connection in the schematic.
    5. It's bad design creation to leave a GND net without a Ground symbol, though, so we'll also copy the GND part. Use the 'Copy' command (Shortcut Key 'F8') and click the GND part by the FT232. Now just place the GND part on the end of the GND stub just created. Make sure the pin doesn't overlap the net, the pin and the net should only touch at their endpoints.
20. Connect the GND pin of the USB connector to the GND net just created.
21. Copy the GND part once more and place it beneath the two remaining unconnected capacitors. Run a net from one end of each of these capacitors to the new GND part. This time there won't be a prompt to connect the signals, so verify the connections with the 'Show' command. Make sure the new nets are connected to all the other GND nets in the schematic.
22. Repeat step 20 with the VCC part and the opposite ends of the two capacitors.
23. Run a net from D- on the USB connector to the USBDM pin on the FT232. You'll have to cross a net that's already been created. The nets won't merge unless you click on a previous net. Crossing nets is fine, but avoid placing nets on top of each other.
24. Run a net from D+ on the USB connector to the USBDP pin on the FT232.
25. Connect the 4 ground pins (AGND, GND7, GND18, GND21) of the FT232 to a GND net.
26. Connect the TEST pin of the FT232 to a GND net.

27. The design is about finished, but it seems there's something missing. The purpose of the design is to create a module that will convert a USB signal to a Serial signal. We have a place where the USB signal comes (the USB connector), but we have nowhere for the Serial signal to go.
    1. Add a serial header to the design. A serial header has 4 pins (VCC, GND, Tx, Rx).
    2. Use the 'Add' command in the Toolbar and find the part M04. Select the M04PTH footprint.
28. Run a net from the TXD pin on the FT232 to pin 3 of the header.
29. Run a net from the RXD pin on the FT232 to pin 4 of the header.
30. Connect pin 1 of the header to VCC.
31. Connect pin 2 of the header to GND.
32. There's no name for the 4 pin header. Add a note to the schematic so we know what it is.
    1. Use the 'Text' command in the Toolbar to add a note. Assign the text Serial Header.
    2. We can place the text in different layers. By default the text is probably green and assigned to the Nets layer. Click the middle mouse button and choose the Info layer. Place the text near the 4 pin header.
33. For vanities sake the schematic should be centered on the frame. Use the 'Select' command (Shortcut Key 'Alt+F7') and select the entire design. Moving a group of parts is a bit different than moving individual parts; hold the 'Ctrl' key and right-click near the center of the group. While holding the 'Ctrl' key move the group to the desired location. Click to finish the move.
34. The schematic is finished, but we need to check the design before moving on. Run the ERC (Electrical Rule Check) to check for errors on the board. This won't check the validity of the design (you have to do that!), it's more like a spell-check for a schematic. The ERC will check for unconnected nets, warn if pins may look connected but aren't, check if a pin isn't the right type, etc...
    1. There will *likely* be errors reported in the DRC. This doen't mean you can't move on; just read all the errors and make sure they are intended mistakes.
35. The design is finished! Save the schematic and move on to the PCB Layout.

Laying out a PCB

Step-by-Step Guide to laying out the PCB for the FT232 USB to Serial Converter

1. Before starting the layout we need to create the board file. Open the schematic and press the 'Generate Board from Schematic' button located along the top of the schematic window.
2. A prompt will appear with the notification that a board doesn't exist and asking if it should be created. Click 'Yes.' In the new window you'll notice a thin white rectangle and 8 smaller parts. The white rectangle is the physical outline of the board, and the 8 parts correspond to the parts added in the schematic. All of the yellowish colored lines are called Airwires; these correspond to unrouted traces and will disappear as we route the board.
3. Start by redrawing the board shape. Use the 'Delete' tool (Shortcut Key F3) to delete the white rectangle. Each segment will have to be deleted individually.
4. Now redraw a more appropriately sized board. Select the 'Line' tool in the toolbar on the left. Starting at the origin of the board (the dotted crosshair), draw a rectangle that will be big enough to fit all 8 components of the board.
    1. The rectangle needs to be placed on the 'Dimension' layer. To do this, click the layer drop-down box at the top of the window and select 'Dimension.'

*Hint: You can also access the layers by clicking the middle mouse button

1. While drawing the rectangle, the angle might not be set to 90 degrees; to change this just right-click while drawing the wire until the appropriate angle is shown.
2. Press the escape key to stop drawing.

2. Click the layer drop-down box again. Notice at the top of the list there are a bunch of layers named 'Route X.' These are rarely used and end up causing a lot of annoyance. To get rid of them:
    1. Open the DRC
    2. Erase the parenthesis around 1*16 in the setup box of the Layers tab.
    3. Press 'Check.'
    4. Open the DRC again.
    5. Put the parenthesis back around 1*16 in the setup box of the Layers tab.
    6. Press 'Check' again.

Now the erroneous layers are gone!

3. Before we start moving components around, it would be nice to see which component is which (though it might be obvious). Click the 'Layer' tool in the toolbar (this is different than the drop-down box), and scroll down to find the layer name tNames. Click the number 25 next to the name so that the number is highlighted in blue, this turns the names layer on. Press OK to confirm.
4. Ah, now things are more clear! The names on the parts in the PCB correspond to the names on the parts in the schematic. We can start moving things now. Start with the FT232 module. If we check the schematic we can see that the module is named 'IC1.' Use the 'Move' tool (Shortcut Key F7) to move this part to the center of the board.
5. Next move the USB connector onto the board, the USB connector is named X1. Notice the yellow outline around one end of the connector. This indicates the physical dimension of the part; the yellow part of the connector needs to stick off the end of the board so that a USB cable can be plugged in without running into the board. Move the USB connector onto the board but make sure that the yellow outline is hanging outside the boards edge.
6. As a side note, notice the bright white shapes around the parts. These are silk-screen markings that will be printed on the PCB. They are used to help identify how a part should be placed on a board when the board is populated.
7. Now move the 4 pin header onto the board. It should go on the opposite side of the board as the USB connector so that we can connect to the header while a USB cable is plugged into the board.
8. Move the rest of the components onto the board one at a time. Make sure none of the 'pads' are touching one another. Try placing the components so that the airwires are as short as possible. A part can be rotated while being moved by right-clicking. Try positioning the parts to keep the airwires from getting tangled up; keep the airwires as short as possible.
9. Often times when parts are moved the airwires need to be recalculated. To do this the Ratsnest command must be used. Press F8, or click the Ratsnest icon on the toolbar.
10. Before routing the board, add a ground plane. To do this, select the 'Polygon' tool from the toolbar (make sure not to use the rectangle tool). Next, select the 'Top' layer from the layer drop-down menu (or middle click and select the layer). Draw a rectangle around the board.
11. Repeat step 13, but place the rectangle on the bottom layer.
12. Now rename the two rectangles so that they are connected to ground. Use the 'Name' command (Shortcut Key 'Alt + F5') to rename the red and blue rectangles as GND.
13. Run the Ratsnest command again. Whoah! The whole board changed color. This is because the rectangles named ground fill in and connect to any pin in the design with the same name.

14. Now we're ready to start routing the board. In the schematic we created 'Nets' to connect pins; in the board there is an airwire everywhere where a corresponding net is in the schematic. To connect the pins in the board we must create a *trace* that connects the pins. Connecting pins along an airwire will cause the airwire to disappear; however the airwire can be used as a guide while routing the trace to find the destination. Here are some general guidelines for routing:
    1. Start with the shortest airwires and progress to the longest ones.
    2. Before starting a trace, use the 'Show' tool and click on the airwire you're attempting to trace. This will highlight the airwire and the pins that need to be connected. This will help you plan a path for the route.
    3. Never have two routes from different signals intersect.
15. Select the 'Route' tool (Shortcut Key F9) and click on a pin to route. Drag the trace to the another highlighted pin. If the trace is not bending at 45 degree angles, right-click until it is. Finish the trace by clicking again. The trace will automatically end when connected properly. Here are some notes to help while routing:
    1. A trace can be placed on either the Top or the Bottom layer, but since all of the components are on the top a trace must go from the top to the bottom, then back to the top if this is to happen. To go from the Top to the Bottom with a trace, press the middle button while routing the trace to create a *via.* A via is like a tunnel for the trace.
    2. Traces on the top are red, traces on the bottom are blue.
    3. If airwires aren't disappearing when you think they should, try running the Ratsnest again.
    4. Notice that right after the Ratsnest is run, the number of remaining airwires is shown in the bottom left corner of the screen.
    5. The USB and 4 pin header have holes instead of pins. A hole can connect to a trace from the top or the bottom; a pin, like on the capacitors and FT232, can only connect on the top.
    6. Don't overlap any pins with a trace(except the ones being routed to).
    7. Hold the 'Alt' key while running a trace to route on a finer grid.
    8. The 'Delete' command won't get rid of a trace. Instead, if you need to re-route a trace you must first 'Ripup' the original trace. Use the shortcut 'Alt+F9' to select the 'Ripup' tool.
    9. Use the scroll wheel to zoom in and out while routing.
    10. Press and hold the middle mouse button while routing to pan around the board without modifying the trace.
    11. When you think you've routed all of the traces, run the ratsnest. Make sure the status bar at the bottom left of the window reads: “Ratsnest: Nothing to do!”

16. When the board is done being routed, there's still some work to do. First off, what the heck is this thing? Once you start getting multiple boards out at the same time, it's easy to forget which board is which when they start arriving. We need to add labels. It's also very smart to label any pins or switches on the board to make it easier to connect to. If you fail to label the pins, you'll find you have to refer back to your schematic much more often.
    1. Add a title to the board. Use the Text tool from the toolbar and label the board (USB to Serial Converter). Place the text on the tPlace layer.
    2. Is the text too big? Go to the 'Change' tool in the toolbar, then go to 'size' and select a smaller size. Try not to go smaller than .04 with the text.
    3. Label each pin of the 4 pin header. The labels should be VCC, GND, Tx and Rx. Make sure to label them correctly.
17. Add a version number somewhere as well (v10). This helps identify the board in case multiple version are made with very minor changes.
18. Before we call the design complete we must run the DRC, or Design Rule Check. Click the DRC tool.
    1. Click 'load' and then select the SparkFun DRC. This will ensure that the board can be built by SparkFun PCB manufacturer. Different PCB manufacturerers will have different DRC rules that must be followed.
    2. Click 'Check.'
    3. If there are any errors with the board layout, they will be listed. Some of them may be able to be ignored, this must be determined by the designer; but it's always good to be very aware of the PCB fabrication house rules.
19. The design is now finished! Next the gerber files need to be created. The gerbers are a collection of files the fabrication house uses to create the PCB. Start by clicking the 'Gerber' tool near the top of the window.
20. In the new window, go to File->Open->Job... and select sfe-gerb274x.cam.
21. Click “Process Job.” The gerbers are created and saved in the same directory as the project.
22. Before we're ready to send the gerbers off to the fabrication house we need to double check that the gerbers were created properly. We use a program called a 'Gerber Viewer' to do this. Open the Pentalogix ViewMate software.
    1. Go to file->import->gerber...
    2. Navigate to the directory where your project was stored and select all of the files with the extension .GXX. There are 7 files.
    3. Inspect the gerbers. You can turn different layers on and off to ensure everything is correct.
23. Once the gerbers have been verified the design is ready to be sent off. Usually this requires just zipping the the gerber and drill files into a folder and submitting them online. Typically just include all of the files except the schematic and board file.

sparkfun™
ELECTRONICS

WEBSITE: sparkfun.com
6175 LONGBOW DRIVE, SUITE 200
BOULDER. COLORADO USA
ZIP CODE: 80301
[303] 284.0979 [GENERAL] P
443.0048 F

Eagle Schematic Reference

SparkFun Electronics Summer Semester

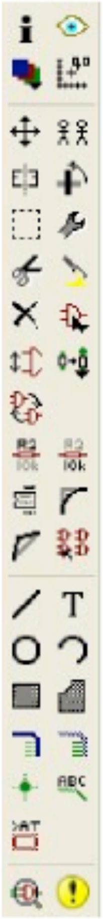
SCHEMATIC SHORTCUTS:

KEY	ICON	COMMAND	DESCRIPTION
F3	✕	Delete	Deletes any object from the schematic (Warning: Deleting a trace or a component will cause the corresponding trace/component on the PCB to be deleted.)
F4	R2 10k	Name	Allow the renaming of components or 'Nets' (Signal)
F5	R2 10k	Value	Used to set the value of a component
F6		Smash	Allows the component name and value to be repositioned independent of the component
F7	✥	Move	Moves a component (Hint: Hold 'Control' while moving to move a selected group)
Alt +F7	⬚	Group	Use this to select a group of objects
F8		Copy	Copies a component or trace (Warning: Won't copy a group!)
F9		Net	Use this to create a 'wire' connecting different signals
Alt +F9	ABC	Label	Puts a text label with a nets name on the schematic
F10		Set Small Grid	Changes the grid to 0.05"
Alt +F10		Set Metric Grid	Changes the grid to 1mm
		Add	Adds a part to the schematic
		ERC	Run the Electrical Rule Check
		Grid	Opens the grid setting window
Alt +F1	i	Info	Show the attributes of a component
	●	J unction	Adds a junction node to the intersection of two signals
		Layer	Show the Layer options for the schematic
	✂	Cut	Copies a group of parts in the schematic
		Paste	Pastes a copied group in the schematic (Not used with Copy)
F1	👁	Show	Shows the all of the connected signals in a specific net
		Create Board	Creates a PCB design file and adds all of the parts from the schematic to the board.

Eagle Schematic Reference

SparkFun Electronics Summer Semester

SCHEMATIC TOOLBAR



Info	Show
Layers	Mark
Move	Copy
Mirror	Rotate
Group	Change
Cut	Paste
Delete	Add
Swap	Replace
Gateswap	
Name	Value
Smash	Miter
Bend	Activate
Line	Text
Circle	Arc
Rectangle	Polygon
Bus	Net
Junction	Label
Attribute	
ERC	Show Errors

SCHEMATIC TIPS AND TRICKS

Always use a frame in a schematic, and keep all the parts within the frame. If more space is needed, add a new page and frame to the design.

Change shortcut assignments by going to the Assign... window in the Options menu.

Copy, Cut and Paste don't work like you'd think they would. Be careful to understand exactly what you're doing before using these tools.

The normal commands (Ctrl+C, Ctrl+X and Ctrl+V) don't work in Eagle.

If a trace is copied and pasted, go back and double check it's connections using the 'show' command to ensure it's been connected in the intended spot.

The 'Cut' command is used to copy a group of components and nets. It won't remove the group that is 'cut.'

Name your nets. This will make routing the board easier.

Add text labels to sections of the board to make it easier to understand from a third persons view.

Always remember to run the ERC before moving onto the PCB layout.

To move a group of components first select the group, then with the 'Move' tool selected hold the 'Ctrl' key and right-click near the center of the group. Keep holding the 'Ctrl' key and move the group to the desired location; click again to finish the move.

Press and hold the middle mouse button and drag the mouse around to pan around the window.

Eagle Layout Reference

SparkFun Electronics Summer Semester

PCB LAYOUT SETTINGS:

0.05" Grid turned ON

0.02" Round Vias

0.01" Trace width

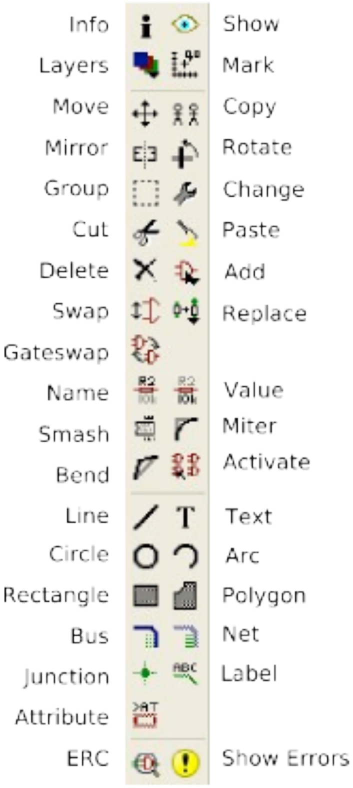
15% Ratio for Text

PCB SHORTCUTS:

KEY	ICON	COMMAND	DESCRIPTION
F1	👁	Show	Highlights the selected object and all similar.
Alt + F1	i	Info	Shows the attributes of an object.
Alt + F2	🔍	Zoom to fit	Redraws the window to fit all of the objects in the layout.
F3	✕	Delete	Deletes an object from the PCB
F4	R2	Name	Allow the renaming of components or 'Nets' (Signal)
F5	R2 10k	Value	Used to set the value of a component
F6	▣	Smash	Allows the component name and value to be repositioned independent of the component
F7	✥	Move	Moves a component (Hint: Hold 'Control' while moving to move a selected group)
Alt + F7	⬚	Group	Use this to select a group of objects
F8	✕	Ratsnest	Finds all the 'airwires' in the design (unconnected signals!)
F9	⌐	Route	Use this to create a 'wire' connecting different signals
Alt + F9	⌐	Ripup	Deletes a trace
F10	⠿	Set Small Grid	Changes the grid to 0.05"
Alt + F10	⠿	Set Metric Grid	Changes the grid to 1mm
F11	■	Change Layers 1	Shows the 'Routing' layers
Alt + F11	■	Change Layers 2	Shows the layers to create an assembly sheet
F12		Turn on polygons	When the ratsnest is processed, polygons will be included (often takes much longer, but required before the design can be finished)

Alt +F12		Turn of polygons	Polygons not included in ratsnest. Faster, but won't connect signals to the signal planes on the board (false airwires will appear).
		Copy	Copies a component (Like text or polygons).
		Cut	Copies a group of components.
		Paste	Pastes a group of components that have been cut.
		Generate Gerbers	Generates the design files for the PCB.
		DRC	Runs the Design Rule Check.
	T	Text	Adds text to the PCB
	●	Via	Adds a Via to the board (can be used to transfer a route from the top layer to the bottom layer).
Alt +Backspace		Undo	

SCHEMATIC TOOLBAR



Eagle Layout Reference

SparkFun Electronics Summer Semester

TIPS AND TRICKS

Try placing all components on a 0.1" grid

For signals use at least a .008" (8mil) trace width

For power traces use at least .01" (10mil) trace width, but if there's room to make it bigger then do so.

LABEL, LABEL, LABEL. Label switches (and their states), LEDs, buttons, and headers. Use the silkscreen to designate the purpose of these items.

If a label interferes with a via, either move the via or make sure that the via will be tented.

Put a Date Code on the back of the board. This will help when you've got multiple versions of the same board.

Create Ground Pours on the top and bottom layer

Change the Isolation of the Ground Pour to 12 mil. This prevents manufacturing errors

Group components on the PCB according to how they are grouped in the schematic.

Keep at least 8mil between traces

Only use 45 degree angles on the traces. Never put a 90 degree angle on a trace.

Traces on the same layer (same color) must not intersect unless they are the same signal!

Route traces from the middle of the pad.

While routing a trace, press the middle mouse button to insert a via and transfer the route from one layer to the other.

Use the 'Ratsnest' to find out if there are airwires(unconnected signals).

If you can't find the airwires, open the 'layers' window and click 'none' to turn off all of the layers. Then find the 'Unrouted' layer in the window and turn that layer on. This will show you where the airwires are. After you find them, press F11 to turn all of the layers back on.

When a board is first created there are 16 layers; typically only 2 layers are used on a board. To get rid of the extra layers (a very annoying bug in Eagle):

1. Open the DRC

2. Go to the 'Layers' tab

3. In the 'Setup' text box erase the parenthesis around 1*16.

4. Press 'Check'.

5. Open the DRC again.

6. Put the parenthesis back around 1*16 in the Setup box of the layers tab.

7. Press check again

Hold the 'Alt' key while routing a trace to place the route on a fine grid. (This also works while moving components, but the components should stay on the major grid when possible).

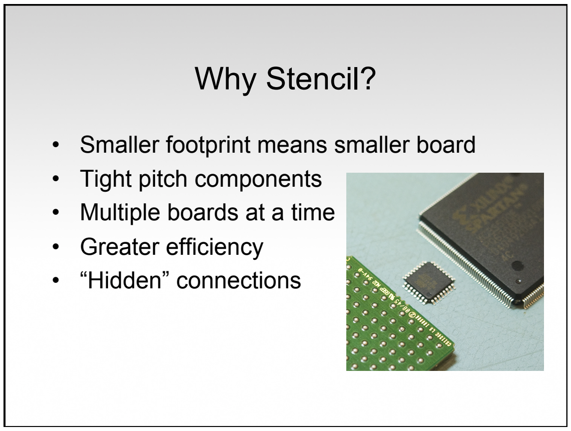
Use the DRC to check the board for design errors.

ALWAYS ALWAYS ALWAYS print a 1:1 copy of the layout and make sure that the parts fit the way they should.

sparkfun ELECTRONICS

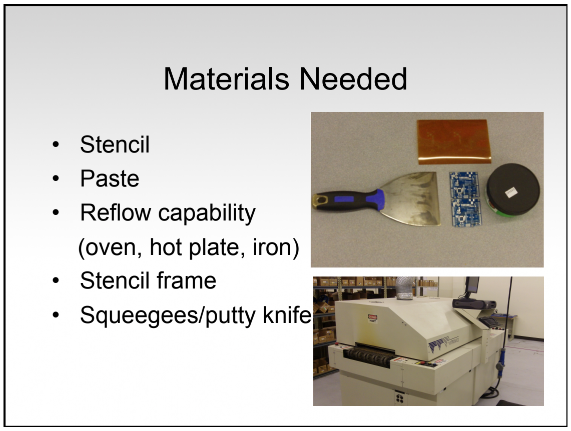
SparkFun Electronics Stenciling

Materials by Bob Hunke & Linz Craig



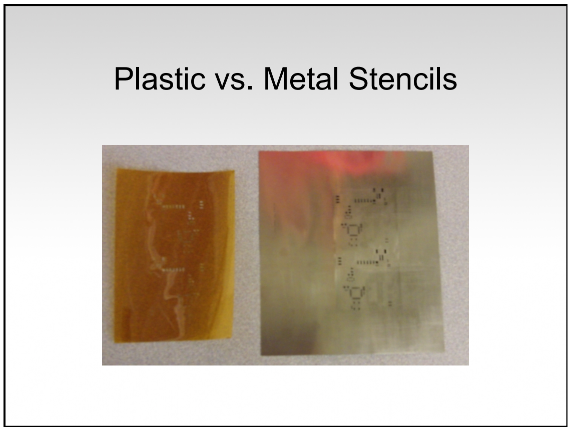
Why Stencil?

- Smaller footprint means smaller board
- Tight pitch components
- Multiple boards at a time
- Greater efficiency
- "Hidden" connections

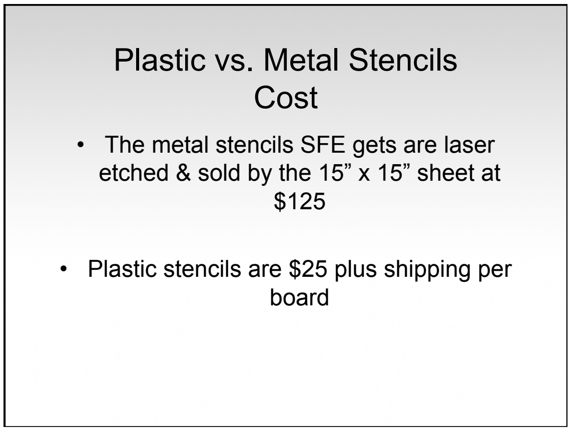


Materials Needed

- Stencil
- Paste
- Reflow capability (oven, hot plate, iron)
- Stencil frame
- Squeegees/putty knife

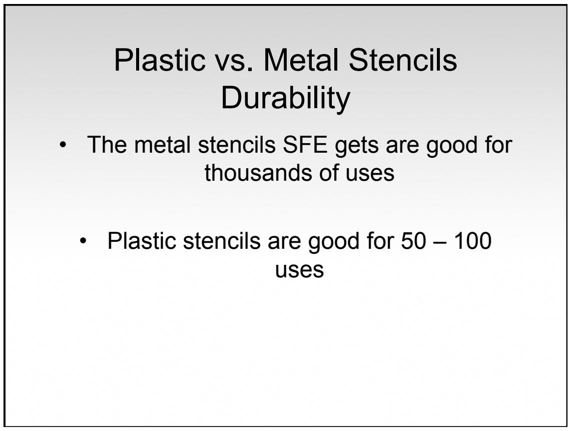


Plastic vs. Metal Stencils



Plastic vs. Metal Stencils Cost

- The metal stencils SFE gets are laser etched & sold by the 15" x 15" sheet at $125
- Plastic stencils are $25 plus shipping per board



Plastic vs. Metal Stencils Durability

- The metal stencils SFE gets are good for thousands of uses
- Plastic stencils are good for 50 – 100 uses

Plastic vs. Metal Stencils
Ease of Use

- Make sure that your stencil extends over the end of your frame or framing scrap PCBs
- Stiffer stencils are easier to pick up off of the board, hence metal is easier to work with

Plastic vs. Metal Stencils
DIY

- Cut your own plastic stencils fairly easily
- Chemically etch your own metal stencils not so easily
- Machine cut your own stencil if you've got the hardware and software (this is becoming more and more common)

Plastic vs. Metal Stencils
Quality of work

- Metal stencils tend to make for better paste placement

Paste Types

- Leaded vs. Lead free
- Flux: Clean vs. no clean
- Size: SFE uses Type 3, average solder sphere of 36 microns
- Five-ball rule for choosing type:

Smallest component footprint should be no smaller than five times the width of average solder sphere

Type 3 Paste Cost

- SFE, leaded: $9.95 for 50g

SFE, leaded

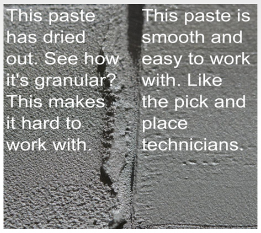
Lead free, $63

Syringe style

- Larger quantites: 500g $100 - $150

Paste Storage and Handling

- Store your paste in a cool place such as a refrigerator
- For short term storage room temperature is okay


This paste has dried out. See how it's granular? This makes it hard to work with.

This paste is smooth and easy to work with. Like the pick and place technicians.

Arrays

If variables can be thought of as buckets that hold a single piece of information, then arrays can be thought of as a collection of buckets, or a big bucket with a lot of little buckets inside. Arrays are extremely useful for a lot of different programs – basically, any time you want to perform a similar operation on several variables (of the same type) – you should consider putting the variables in an array. For example, if I want to blink eight LED's at the same time, I could put them in an array, and then use a for loop to iterate over the array, like so:

```
/* this is the array that holds the pin numbers our LED's would
be connected to */

int ledPins[] = {2,3,4,5,6,7,8,9};


// in setup()we can set all the pins to output with a simple for
loop

// 8, because we have 8 elements in the array

for( int i = 0; i < 8; i++) {

// sets each ledPin in our array to OUTPUT

 pinMode(ledPins[i], OUTPUT);

}
```

The `ledPins[i]` part is important; it allows us to reference each element in our array by its place in the array, starting with 0 (which can be confusing). So, in our example above ledPins[0] == 2, since 2 is the 1st element we put into the array. This means that ledPins[1] == 3, and ledPins[7] == 9, and is the last element in our array.

See http://arduino.cc/en/Reference/Array for further explanation.

Data Types

In addition to the data types already covered like integers, booleans, and characters, there are some more data types that may prove useful for your specific application.

Float: Data type for floating point numbers (those with a decimal point). They can range from 3.4028235E+38 down to -3.4028235E+38. Stored as 32 bits (4 bytes). A word of advice: floating point arithmetic is notoriously unpredictable (e.g. 5.0 / 2.0 may not always come out to 2.5), and much slower than integer operations, so use with caution. Some readers may be familiar with the '**Double**' data type – currently, the Arduino implementation of Double is exactly the same as Float, so if you're importing code that uses doubles make sure the implied functionality is compatible with floats.

Long: Data type for larger numbers, from -2,147,483,648 to 2,147,483,647, and store 32 bits (4 bytes) of information.

String: On the Arduino, there are really two kinds of strings: strings (with a lower case 's') can be created as an array of characters (of type `char`). String (with a capital 'S'), is a String type object. The difference is illustrated in code:

```
Char stringArray[10] = “SparkFun”;

String stringObject = String(“SparkFun”);
```

The advantage of the second method (using the String object) is that it allows you to use a number of built-in methods, such as length(), replace(), and equals().

More methods can be found here: http://arduino.cc/en/Reference/StringObject

Pointers

Pointers seem to be one of the concepts that gives some people a bit of trouble. In the simplest terms, a pointer is a memory address. In C-based languages (including Arduino), when a variable is created, like so:

```
int foo;
```

A certain chunk of memory (how big a chunk is determined by the data type) is allocated to storing the value of that variable. That chunk of memory also has an address. Most of the time we, as programmers don't have to deal with the address, but sometimes dealing directly with a pointer to an address is more efficient. To declare a variable to be a pointer, use an asterisk (*) before the variable name like so:

```
int *pnt;
```

Typically, we would say that `pnt` is declared as a pointer to int. If we want to assign our pointer to the memory address of foo, we can use the & operator, (usually called the *reference* or *address-of* operator), like so:

```
int *pnt = &foo;
```

This tells our pointer to point to the address of our foo variable (not the value of it).

To get or assign a value with a pointer, you need to use the dereference operator (*). Assigning the value from our pointer to a regular integer looks like this:

```
int foo2 = *pnt;
```

Assigning a number value to our pointer can be done like so:

```
*pnt = 100; //sets whatever pnt is pointing to a value of 100
```

Note that we cannot simply say:

```
pnt = 100;
```

This would change the memory address of the pointer to 100 – and since there's probably nothing at memory address 100, we would get an error.

There's plenty more on pointers on the internet, mostly within the C language.

Note that in Arduino programming, you may never have to use a pointer, although for certain operations it will simplify your code – but only if you know what you're doing.

Sending and Receiving data in different formats

One of the common mistakes when attempting to get two devices to communicate with each other is the improper formatting of data. While data formatting is a very broad discussion, we have a few specific tips for the Arduino environment to make sure that you don't make this mistake.

Normally, when we do a print statement in Arduino we just say `Serial.print(foo)` or `Serial.println(foo)`, which sends human-readable ASCII text over the serial port. However, the Arduino print commands have a second argument that allows you specify how you want to encode the data. So, for example (from the Arduino reference):

```
Serial.print(78, BYTE) // gives "N"

Serial.print(78, BIN) // gives "1001110"

Serial.print(78, OCT) // gives "116"

Serial.print(78, DEC) // gives "78"

Serial.print(78, HEX) // gives "4E"
```

For floating point numbers, the second parameter specifies the number of digits after the decimal place to include when sending – note that the default for sending floating point numbers cuts off at 2 places past the decimal point:

```
Serial.println(1.23456, 0) // gives "1"

Serial.println(1.23456, 2) // gives "1.23"

Serial.println(1.23456, 4) // gives "1.2345"
```

Switch/Case statements

Sometimes you've got a bunch of if statements that all check the same variable. It is possible to combine all these if statements into one statement called a Switch/Case statement. The Switch/Case statement will check the value of a given variable against the values specified in the case statements and execute code if the values match. So for example (from the Arduino Reference):

```
switch (var) {
      case 1:
      //do something when var equals 1
     break;
     case 2:
      //do something when var equals 2
     break;
     default:
      // if nothing else matches, do the default
      // default is optional
}
```

The example above uses an integer variable called var and has cases that check if var is equal to 1 or 2 with a default to handle all the other values it could have. Here is some pseudo-code to explain the syntax:

```
switch (var) {
      case label:
      // code that executes when var is equal to label
      break;
      case label:
      // code that executes when var is equal to label
      break;
      default:
      // code that executes when var is equal to label
}
```

Break indicates that the computer should exit the entire switch/case statement and continue with the rest of the code in the sketch. Break is not strictly necessary when writing a switch/case statement, but if you do omit it the computer will go on to check the rest of the cases until it encounters a break or checks every single case. In this example var is the only text that you might change so that it matches the variable you are checking against. Default is optional, so you can omit it, but usual you will want to use this as a catch all in case your variable winds up with a value you have not listed in the various cases.

Constants and Define statements

If we know that the value of a variable is never going to change (or we know we don't want it to change), we have the option of declaring it a *constant.* As in, its value remains constant throughout the program.

There are a two common ways to do this: #define or const.

A define statement appears at the beginning of your program, and looks like this:

```
#define ledPin 3
```

Notice the '#', the lack of an assignment operator (no = sign), and no semicolon (;). Omitting the #, or introducing = or ; into the statement are all errors. The main advantage to using a define statement is that the constant doesn't take up any extra memory space on the chip – the compiler simply goes through the program and replaces every instance of your variable (`ledPin` in this case) with the value you defined for it (`3` in this case).

There is a downside to using define statements, however: if for example, a constant name that you #defined is included in some other constant or variable name, the text would be replaced by the #defined number (or text). Bad news.

For this reason, many programmers prefer using the `const` keyword. You can think of the `const` keyword as a modifier before declaring any variable, so the syntax looks like:

```
const int ledPin = 3;
```

This declares a constant integer value of 3 for our ledPin variable. If we try to modify it like so:

```
ledPin = 4;
```

We will get an error. However, you can use the constant value in math:

```
X = ledPin * 4;
```

Generally, define statements are fine, but using `const` is considered the preferred, safer method for declaring constants if memory space is not a concern.

Bitwise Operations and Bit Shifting

Sometimes it becomes necessary to work on the binary level, and affect changes through bitwise operations and bit shifting (for an example, check out the multiplexor code (circuit 5) from the SparkFun Inventors Kit http://www.oomlout.com/a/products/ardx/circ-05).

Note that the notations for bitwise operator are different than their logical counterparts (& instead of &&, for example).

There are four bitwise operators: NOT, AND, OR, and XOR.

The NOT operator (~) simply flips the value of each bit:

```
NOT 1010

  = 0101
```

The AND operator (&) takes two binary representations (of equal length) and preforms a logical AND operation on each corresponding pair of bits. If the first bit is 1 AND the second, corresponding bit is 1, then the result is 1 – otherwise, the result is 0. So:

```
    0101

AND 0011

  = 0001
```

The OR operator (a single pipe, l) takes two binary representations and performs a logical OR operation on each pair of corresponding bits. So, if the 1st bit OR the 2nd bit is a 1, the result is 1. For example:

```
   0101

OR 0011

 = 0111
```

The exclusive or operator XOR (^) performs the logical XOR operation on two binary representations of equal length. The result is 1 if *only* the first bit or *only* the second bit is 1, but returns 0 if *both* bits are 1 or *both* bits are 0. For example:

```
    0101
XOR 0011
  = 0110
```

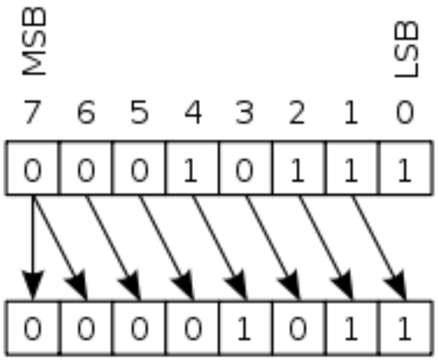
Bit Shifts, as opposed to bitwise operators, occur on only a single binary representation (not a pair). Several concepts become important when introducing bit shifting: the most significant bit (MSB) and least significant bit (LSB). The most significant bit is the bit position in the binary number that has the greatest value (typically the bit furthest to the left). The least significant bit is that bit with the lowest numerical value.

There are four typical types of bit shifts that you may encounter: Arithmetic Shift, Logical Shift, Rotate No Carry, and Rotate Through Carry. Each type of shift can either be shift-right or shift-left, indicating which direction the bits are being shifted.

Arithmetic Shifts

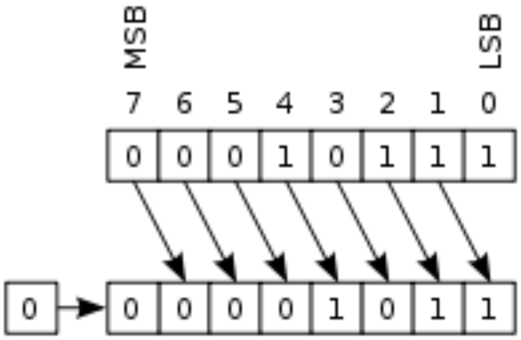
Arithmetic Shifts shift out the bit at either end and discard it. In a left shift, all the bit values shift to the left, and a 0 is inserted into the new place (on the far right). In a right shift, the sign bit is shifted in on the right (the sign bit is usually a copy of the MSB at the time, also defining the sign of the number in some cases).

The following is an arithmetic shift right:

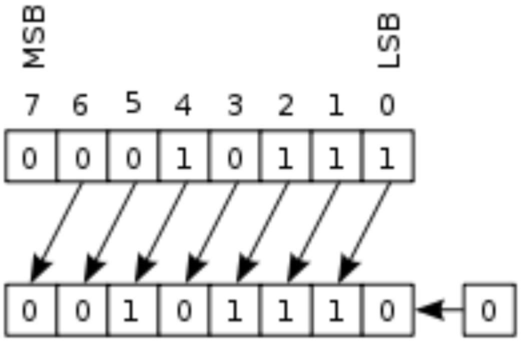


Logical shifts are very similar to arithmetic shifts, with bits shifted out being replaced by 0. In the case of a left-shift, arithmetic and logical shifts are exactly the same. In the right shift however, a 0 is always inserted with no attention paid to the sign bit.

Notice the difference in a right logical shift:

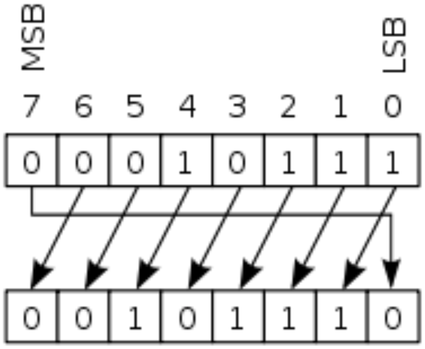


And here is a left logical shift:



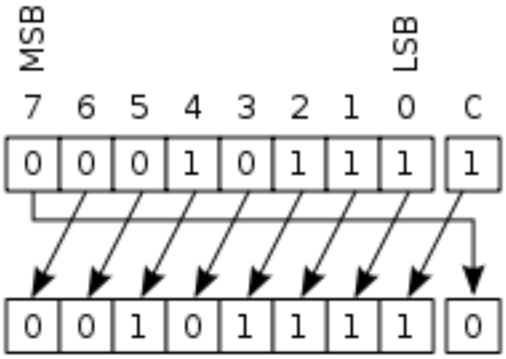
Next we have Rotate No Carry (also called Circular) and Rotate Through Carry shifts. In a Rotate No Carry shift, the bits are rotated as if they were joined at either end – the bit that gets shifted off gets moved around to the other side instead of being discarded.

Pictured is a rotate left bit shift (no carry):



The difference between rotate through carry and rotate no-carry refers to the presence (or absence) of a carry flag. The carry flag is basically a holder for another bit, and the holder can be thought of as in between the two ends of the register. This means that whatever bit gets shifted out get placed into the carry flag, and the old value that was in the carry flag gets shifted in on the appropriate side (depending on left or right shift).

Here’s a diagram of a left rotate through carry:



Notice how the carry bit (labeled ‘C’) gets shifted in and replaced by the bit being shifted out.

Fortunately, in C++ (and in Arduino) there are only two bit shifting operations, bitshift left (<<<) and bitshift right (>>>).

Luckily, these operations are equivalent to the ‘arithmetic’ shift mentioned above. For example:

```
int a = 5;         // binary: 0000000000000101
int b = a << 3;    // binary: 0000000000101000, or 40 in decimal
int c = b >> 3;    // binary: 0000000000000101, or back to 5
like we started with
```

So there’s your (not so) brief introduction to bit wise operations and bit shifting.

Casting

Say you want to map a sensor value from it's normal range (0 to 1023) to a new range (0 to 100). You want to use the map function because it does all the math for you (yay!), but you can't because your sensor value is an integer and the map function requires a float (in Arduino it doesn't, but in Processing it does). Are you stuck? No! Although you could go back and change your code to make all the integers into floats, you can also perform a *cast* – translating one variable type into another.

It looks something like this:

```
int sensorValue = analogRead(A0); //our integer variable

float newValue = (float) sensorValue;  //new float variable
takes the int and casts it into a float – neat!
```

Basically, the rule is to put the data type that you want your variable to become in parentheses when assigning it to a new variable.

For our earlier example, we could save a step by casting our variable directly in the map statement, like so:

```
int sensorValue = analogRead(A0);

sensorValue = map((float)sensorValue, 0, 1023, 0, 100);
```

Note that there are sometimes consequences for casting. For instance, when casting from a float to an integer, the decimal places will be lost (not rounded) – so 3.7 becomes 3, as does 3.1.

Bootloaders and ICSP

Every Arduino board that you buy comes with a small program – called a bootloader - already pre-programmed onto the chip. The Arduino bootloader (there's a few different ones depending on the chip) allows you to write and upload programs onto the chip from the Arduino software environment.

ICSP (In-Circuit Serial Programming) is simply the method by which you can (if you wish) directly access the chip that is the 'brain' of the Arduino board (Usually an ATmega of some kind). This is handy to know about for a few reasons: if you want to replace the ATmega chip in your Arduino, you can simply buy a new chip, put it in the Arduino board and burn the Arduino bootloader onto it (instead of buying a whole new board - much more cost effective). Also, if you need to clear 1-2KB from the flash memory of your chip to fit your program on, removing the bootloader is one way to do it.

In order to make use of external programmers using ICSP, you'll need do to one of three things:

- Get an AVR-ISP (http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2726)
- Get a USBtinyISP (http://www.ladyada.net/make/usbtinyisp/)
- Build a Parallel Programmer (http://arduino.cc/en/Hacking/ParallelProgrammer)

A great page to started with the bootloader and ICSP with Arduino: http://arduino.cc/en/Hacking/Bootloader

Enjoy!

Wireless Overview

SparkFun Electronics Summer Semester

WIRELESS OVERVIEW

There's a lot of different ways to get data from a remote unit to somewhere else. In this guide, we attempt to show you different technologies and the tradeoffs between those technologies. Remember, we are completely biased. We've used a lot of these modules, and we apologize to the companies that spent a lot of effort designing them. This is just what we think!

	Power	Distance	Data Rate	Data Delivery	Cost	Learning Curve	Size
Nordic	5	1	2	2	5	3	5
Cellular	1	5	4	5	1	5	1
Bluetooth	3	3	4	4	3	5	3
Zigbee	4	1	2	2	4	2	3
XBee	3	3	3	3	4	4	3
WiFI	2	3	5	5	2	3	3
FM	4	4	*	*	4	4	4
General	4	4	1	1	5	1	4

KEY	5 = Lowest	5 = Longest	5 = Highest	5 = Guaranteed Delivery	5 = Cheapest	5 = Shortest time learning	5 = Smallest
	1 = Highest	1 = Shortest	1 = Lowest	1 = May Not Get There	1 = Most Expensive	1 = Lots of time learning	1 = Largest

Pictured above is our quick and dirty breakdown. Now let us explain ourselves…

Before you get started, think about your project goals. Then answer these questions:

- Is battery life a concern?
- How far does the wireless link need to transmit?
- How much data does your system need to transmit?
- Is it ok if some data gets lost? Or do all packets abslutely have to arrive at the base?
- How much are you able to spend?
- How much time do you have to play?
- How small does it need to be?

Answering these questions will help you significantly narrow down the search for the RF solution that best fits your project.

Features that are addressed by the questions:

- Power
- Distance
- Data rate
- Data delivery
- Cost
- Learning curve
- Size

Potential RF Solutions:

- Nordic
- Cellular
- Bluetooth
- Zigbee
- XBee
- WiFi
- FM
- General

Power:

Depending on the type of solution, power consumption can range from micro amps to amps (6 orders of magnitude). Historically, RF requires a lot of power. If you're trying to broadcast pirate radio, expect to not be able to run your station off batteries. However, if you need to chirp some data every few minutes or on the hour, your can significantly reduce your power and do some fun sleep mode tricks.

Connecting a device to the power grid will open up many RF options. On the other hand, if you're planning on making your RF device portable, consider how much mAh (mili-amp hour) time your battery has and how often you need to transmit your data. If you've got a 50mAh coin cell and your RF device uses 12mA when receiving, you've got 50/12 = ~4 hours of run time. Now if you power cycle the RF device (for example, in receive mode 5% of the time), you can significantly increase your battery life (50/0.6 = ~83 hours = ~3.5 days run time).

Wireless Overview

SparkFun Electronics Summer Semester

Distance:

Distance is related to power. If you need to transmit ~10-50 meters (32 to 164ft), there are lots of options open. If you need to broadcast over the African Sahara, you may need a bit more umph. Some of the XBees claim miles of range, but this is usually with very directional antennas and with a bit of power behind it (210mA at 3.3V). In embedded electronics land, we deal with mW (mili-Watts). If you need watts of power for your pirate radio station, you are probably talking about miles of transmission range, but you are not going to get it with a coincell battery.

Data Rate:

Do you need to download torrents? Or do you need to receive a 4-byte analog to digital conversion? Data rate is everything. Some of the simple data links are limited to ~300 bytes per second (2400 bps). You can much higher datarates (wifi being the forerunner) but this adds overhead, complexity, and power consumption. Low power go s hand in hand with low bandwidth. A small battery usually means you will be able to broadcast a handle of bytes (hundreds) every handle of seconds. If you're looking to stream audio data off Pandora, you are probably looking at the wrong tutorial.

Data Delivery:

Do I really care if my data gets to where it needs to be? Sounds funny, no? Well it's actually quite applicable. RF is a finicky. thing. One second I can get 100 bytes where it needs to be and the next minute one of the bytes get corrupted.

Because we cannot see high frequency RF do s not mean it is not there. There is a lot of radiation coming from normal house hold wiring, computer monitors, the sun, even static electricity. There are a ton of EMI (electro-magnetic interference) sources. And we really cannot predict where the interference will come from. Therefore, when we broadcast 'Hello world', it may get garbled on its way to the lunar lander. One of the ways to protect against this is verifying that the data was delivered correctly. Basically we wait for the receiver of the information to confirm 'yes, I heard you'. This systems of:

- Broadcast
- Wait for the receiver to say 'I got it!', or if too much time passes
- Re-broadcast

can add a lot of complexity to the overhead of the protocol (more power, less data bandwidth, etc). Do you really care if your garage door opener fails to open your garage door? No, you just hit the button again. Do you care if your cardiotocograph gets recorded correctly to the nurses' station - um, ya.

Cost:

This correlates with power and complexity of use. If you've got a bunch of free time, you can probably spend a few days/weeks hacking at the really cheap RF links, creating your own protocol, and verifying checksums. However, if you simple need to plug it in and have it work - now!, expect to spend more money to have something work out of the box (Bluetooth and Cellular are the simplest solutions for unseasoned users).

Learning Curve:

This characteristic ties closely with cost - if you've got no money, expect to spend a bit more time wading through example code and datasheets. Once you get a basic RF link working, it's awesome! But it may take a few ferocious days of beating your head against the wall. On the other hand, Bluetooth may seem exorbitantly expensive, but you can open a serial pipe with a few mouse clicks. No matter which solution you choose, expect to spend many hours pouring over datasheets and/or example code. But once you really wrap your head around the wireless solution, it will seem trivial.

Size:

Regardless of what you may have seen in the movies, slipping someone an RF pill won't allow you to track them across a country. Help us with a good movie example! RF means a certain amount of size simply for the antenna and power source. There are very sneaky ways of miniaturizing the antenna but every time you fold the antenna smaller, you generally loose transmission/reception range (the smaller the antenna, the less distance of coverage you get). As the frequencies get higher (think 2.4GHz, 5.8GHz, etc), the antennas do get smaller. However, the penetration of walls and barriers gets worse. 5.8GHz likes to bounce of household walls, 2.4GHz might make it to the neighbors, 900MHz can get through walls, trees, kids, even livestock, 125kHz can work under water. Do s GPS (1.5752GHz) work under water? No - because water is an awesome insulator (do sn't allow energy to pass through). Then how do s my dive computer work under water? That's because a wireless dive computer works in the kHz range (much lower than MHz or GHz).

It's a trade off: the longer range and better reception (more penetration) you need, the lower the frequency (think kHz, not GHz), the lower the data rate, the bigger the antenna (2.4GHz = 3cm, 900MHz = 9cm, 32kHz = 243,212cm). If you need a 32kHz antenna (giant 768ft antenna) to transmit 1Mbit/s of data, to power off a coincell battery, you may run into problems.

Antenna selection is not trivial. You could have an amazingly powerful RF engine, but a crappy antenna and all of the features will suffer. A chip antenna might seem alluring, but it may not be the best fit with your project. If you need range, use a directional biquad antenna. If you need tiny size and only feet of range, use a trace or wire antenna. Chip antennas are a little better. Bigger duck antennas are incrementally better. It all depends on your application.

Nordic - how I love you so.

Pros:

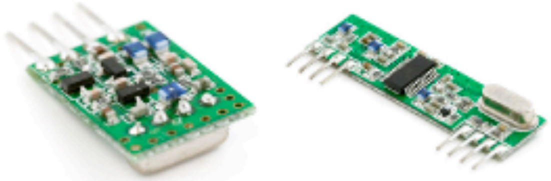
- Low cost - ~$2
- Low power - down to micro-amps
- Super flexible
- Great way to get a very small amount of data from one point to another
- Small footprint

Cons:

- You write your own protocol
- Shorter range (~50 to 100m)

Wireless Overview

SparkFun Electronics Summer Semester



General TX/RX

Pros:

- Cheap
- Basic interface
- Reasonable range (500ft)
- Large voltage range for transmitter (2-12V)

Cons:

- Big/long learning curve to get units functioning correctly
- Limited data rate (2400-4800bps)
- No data filtering or packetization
- Not directly FCC certified

Other:

The general RF transmitter and receiver pairs are commonly used for low-data rate consumer applications - like in your garage door, remote weather station, wireless window remote, the list goes on and on. These units are made cheap, and therefor have no brains. This leads to a headache from the users' perspective. It is possible to get bytes across this link, but it requires quite a bit of hacking, protocol creation, error checking, and patience.

So that's our take on the RF selection at SparkFun. We hope this gives you a bit of insight into what is, and what is not possible when using wireless devices. There are more options than what is listed here. If you know of a good solution, let us know!

Other:

Bluetooth is pretty good. I really liked using it on the popular BlueSMiRF. It has been around for a few years now and the Serial Port Profile (SPP) works great for opening up a serial pipe between two points (what you pass in the RX pin, you get out at the TX pin on the other side of the wireless connection). Anything Bluetooth can pretty much talk to anything else Bluetooth. This makes it pretty flexible and easy to use, but you pay for it in cost, and in power consumption. Bluetooth modules with audio are one of the very few technologies that we've played with that support audio transmission.

XBee rocks. Don't let me confuse you! XBee sounds a lot like Zigbee. That's because this company called Digi (well, Maxstream before they were bought) took all the good stuff about Zigbee, and wrapped it up into an easy to use AT command set. I don't care how you get the data from point A to point B, XBee takes care of it all for me.

Pros:

- Simple to use interface
- Creates serial pipe between point A and point B out of the box
- Comes in many different footprints to support different frequencies, distances, and power ranges.
- Short range (100m) to long range (16 miles!)
- Same AT commands for all the different flavors (awe some)
- Great configuration software (windows only, sorry!)

Cons:

- Uses a slightly hard-to-find 2mm pitch connector. So we designed a bunch of breakout boards (basic, USB, RS232).
- Not low power (50mA)
- Series 2.5 supports multi-point and mesh-node-net working but does it with some really dense datasheets and nomenclature
- Not cheap. ~$25(min) per radio.
- Not all that small. Pretty small, but not really small.

Other:

I like XBee. I wasn't a believer until I started using it for all my wireless bootloading. Now I love it. If power is not an issue, these work great for beginners to get data from one place to another with the least hassle, learning curve, and setup.

WiFi

Pros:

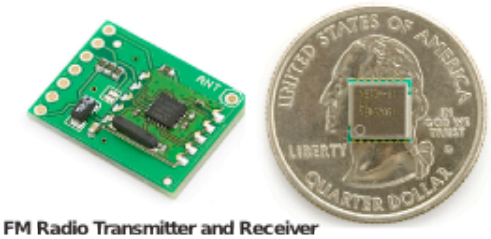
- Super common standard - pipe data to any router any where
- Good for moving large amounts of data (1Mb+)

Cons:

- Not low power - ~100mA
- Requires some serious processing power to handle 802.11 stack

Other:

I have not yet used WiFi. I usually don't need to move more than a few bytes over a wireless link so I haven't found a need for it, but many people have. There are some decent WiFi modules available that give us UART or SPI interface to WiFi, but it's still pretty complex in my opinion.



FM Radio Transmitter and Receiver

Pros:

- A great way to get audio from one device to another
- Easy interface/control
- Small size

Cons:

- Only audio, no data
- Not directly FCC certified

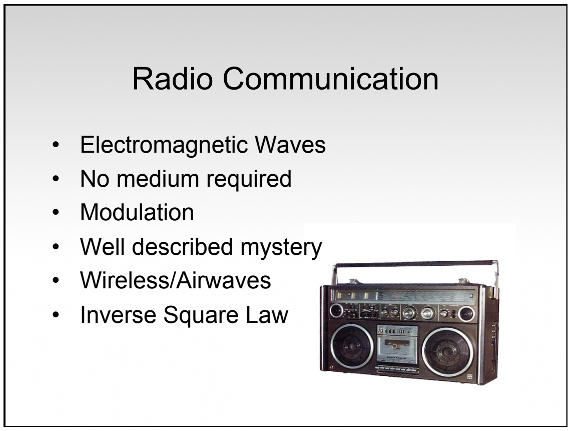
Other:

The FM Transmitter module is easy enough to use, and it works surprisingly well, but it's limited to audio paths only. You can't transmit data (without being sneaky). The module can be FCC certified, but it is not certified off-the-shelf so you have to use it under the 'research only' umbrella. The FM receiver is very handy as well but with Pandora, iTunes, and the proliferation of MP3 - who listens to the radio anymore?

sparkfun ELECTRONICS

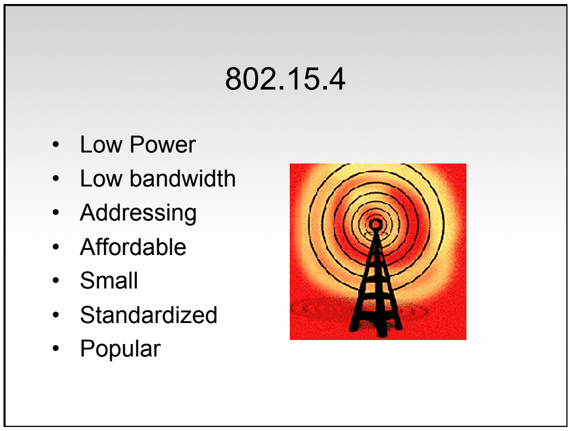
XBee Wireless

Michelle Shorter



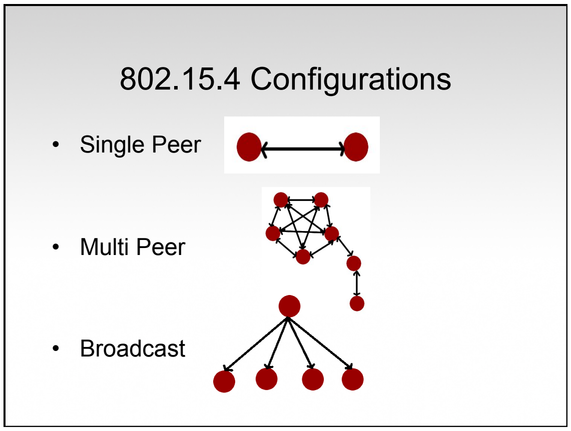
Radio Communication

- Electromagnetic Waves
- No medium required
- Modulation
- Well described mystery
- Wireless/Airwaves
- Inverse Square Law



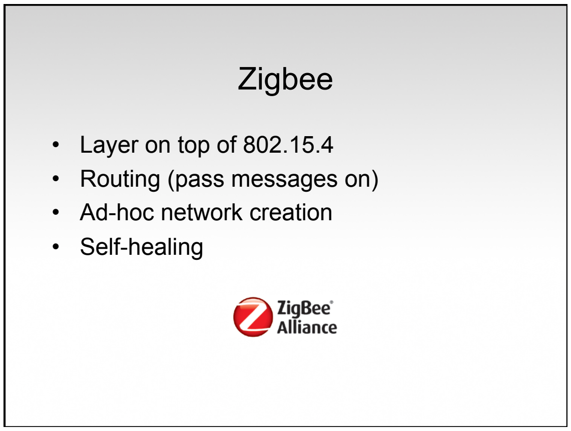
802.15.4

- Low Power
- Low bandwidth
- Addressing
- Affordable
- Small
- Standardized
- Popular



802.15.4 Configurations

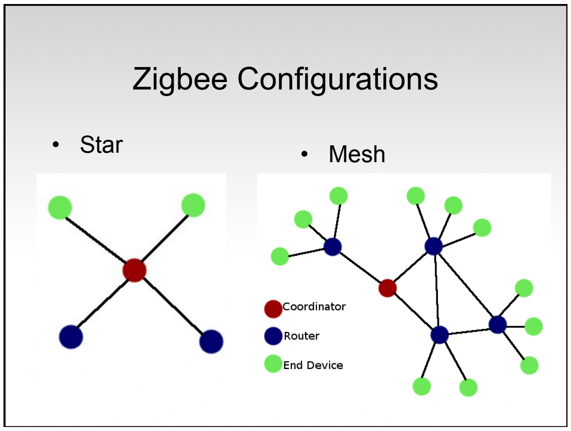
- Single Peer
- Multi Peer
- Broadcast



Zigbee

- Layer on top of 802.15.4
- Routing (pass messages on)
- Ad-hoc network creation
- Self-healing

ZigBee Alliance



Zigbee Configurations

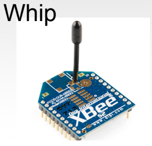
- Star
- Mesh

● Coordinator
● Router
● End Device

Antennas

- Chip

- Whip



- u.FL

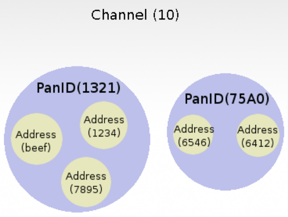
- RPSMA

Regular vs. Pro

- 1-2mW
- Smaller
- Shorter range (100m)
- Cheaper

- 50-60mW
- Longer
- Longer range (300m)
- More expensive

Addressing



Addressing

- Channels
- PAN
- 64-bit address
- High -0013A200 same for all XBees
- Low – each XBee has its own address
- 16-bit address (configurable on Series 1)

Coordinators

- Each network has 1 coordinator
- Coordinator selects channel and PAN ID
- Other devices then join the PAN
- Usually powered by something stable
- 16-bit address is always 0
- Assigns 16-bit address for the router and end devices

Routers

- Optional
- Often powered by something stable
- Can have as many as you want
- Issues a request on startup to find a coordinator/network it can join
- Can talk to any device
- If an end device is sleeping it stores its data
- Coordinator can act as a “super router”

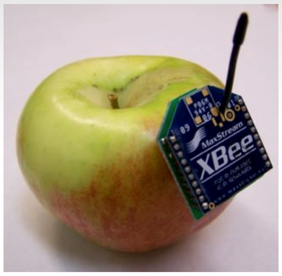
End Devices

- Optional
- Usually battery powered
- Can have as many as you want
- Issues a request on startup to find a network it can join and a parent device (router or coordinator)
- Can only communicate with its parent

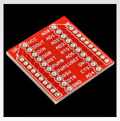
Firmware

- Must upload with X-CTU (on Windows)
- AT firmware vs API firmware
- Coordinator, Router, End Device
- Other
- Each Firmware has different settings

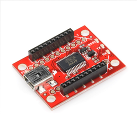
Contrary to this picture X-CTU will not work on your Mac

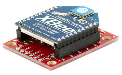


How to Hookup your XBee

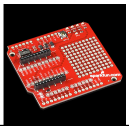


- Breakout Board
- Xbee Explorer





- Xbee Explorer Regulated
- Xbee Shield



Terminal Windows

- X-CTU
- Hyperterm (doesn't come with Windows 7)
- Coolterm (Windows, Mac, Linux)
- Unix/Linux terminal window
- Plenty of others

9600-8-N-1

Getting into Command Mode

- +++ gets you into command mode
- 1 second delay on either side
- No <enter>
- Should get “OK” back
- Times out after 10 seconds

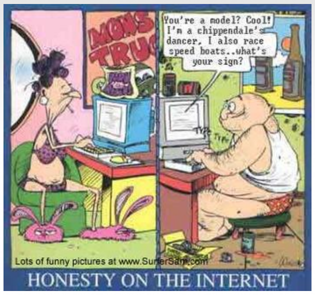
AT Commands

- AT – just returns an “OK”
- ATMY – 16- bit address (Series 1 only)
- ATDH – 64-bit destination address high bits
- ATDL – 64-bit destination address low bits
- ATID – PAN ID
- ATCN – end command mode
- ATRE – reset all settings
- ATWR – write settings to flash

Sending Commands

- Just typing the AT command will give you the setting
- Typing the AT command followed by a value sets the value
- Commands use Hexadecimals
- Always Press Enter
  - >ATID 1111
  - OK
  - >ATID
  - 1111
  - >ATWR
  - OK

Chat Program


MONS TRU
You're a model? Cool! I'm a chippendale's dancer, I also race speed boats..what's your sign?
TYPE TYPE
Lots of funny pictures at www.SurferSam.com
HONESTY ON THE INTERNET

I/O Series 1 vs Series 2

- 8 Digital I/Os
- 7 Analog Inputs
- 2 Analog Outputs (PWM)
- Can't use these all at once
- Straight through I/Os
- Must use Vref

- 10 Digital I/Os
- 4 Analog inputs
- No Analog outputs
- Can't use these all at once
- I/O pins are 1.2V only

To use or not to use

- Saves space
- Save power
- Save weight
- Save money
- Reduce complication

- Limited I/Os
- No logic
- No analog output
- Added Complexity

I/O Commands

- ATD0...D7 configure pins 0-7
- ATP0...P1 configure pins 10,11
- ATIR set the sample rate (in ms)

- Always sends 1 sample per transmission
- Data is sent to destination address

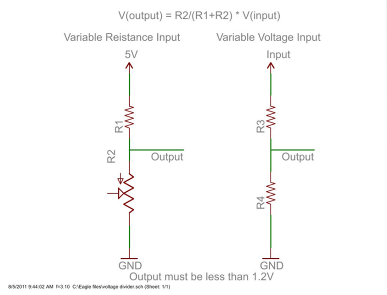
Commands for Series 1 Only

- ATIT samples before transmit (1 for Series 2, configurable for Series 1)
- ATIA – I/O addresses (who can play with my pins)

ATDx Command Options

- 0 - Disabled
- 1 – Built-in function (sometimes)
- 2 – Analog Input (sometimes)
- 3 – Digital Input
- 4 - Digital Output (low to start)
- 5 – Digital Output (high to start)

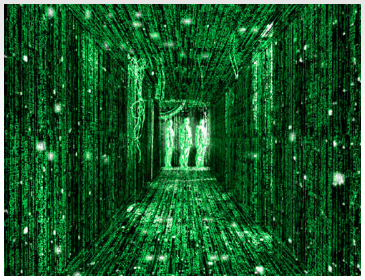
Voltage Divider


V(output) = R2/(R1+R2) * V(input)

Output must be less than 1.2V

8/5/2011 9:44:02 AM f=3.10 C:\Eagle files\voltage divider.sch (Sheet: 1/1)

Serial Sensor Project



Why use a Microcontroller

- Local logic
- Fast prototyping
- I2C, PWM, SPI
- More I/Os
- Xbee Series 2 only allows 1 side of I/O

Arduino HW Serial vs NewSoftSerial

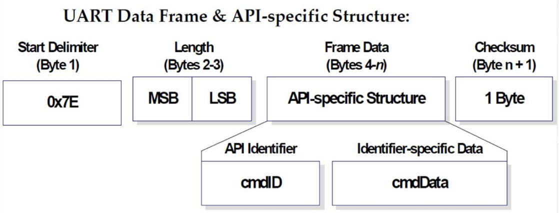
- 1 Serial port on Uno
- 4 Serial ports on Mega
- Connected to USB port

- Unlimited serial ports*
- Requires more processing power
- More likely to loose data
- Higher baud rates less likely to work
- *as many as the processor can handle

API Mode?

- Application Programing Interface
- For computers to talk to other computers
- Structured
- Predictable
- Reliable
- Frames of data
- Radio must be in API mode
- ATAP 1 for Series 1 (ATAP 0 to turn off)
- API firmware for Series 2

API Frame Data



Color Project

Gateways

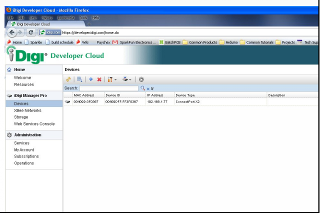
- Connect your Xbee to something else
- Bluetooth
- Ethernet
- Cell Modules
- WiFi
- RFID
- Many others

Digi's Connect Ports

- X8 – Ethernet, Wifi, Cell, USB, serial... $1000
- X5 – Satellite Radio, cell, WiFi, GPS... $1000
- X4 – Ethernet/Wifi, cell $700
- X3 – GSM/FPRS... $250
- X2 – Ethernet or WiFi
  - $100-$200
  - Sparkfun WRL-10569
    - (Ethernet version)
    - $140

iDigi

- Free account with up to 5 Connect Ports
- Remote access your connect port
  - Update firmware on other Xbees in the network
  - Firmware updates
  - Remote Reboot

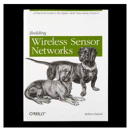


MAC Address	Device ID	IP Address	Device Type	Description
00409D:3F0367	00409DFF-FF3F0367	192.168.1.77	ConnectPort X2	

X2 Example

Troubleshooting

- Only use 3.3V, more than 7 will release magic smoke
- Use decoupling capacitors with a voltage regulator
- TX->RX RX->TX
- Don't overwhelm them, try putting in a small delay



Questions?

sparkfun™
ELECTRONICS

www.sparkfun.com
6175 Longbow Drive, Suite 200
Boulder, Colorado 80301

Simple Sensor Network Activity

* This section borrows heavily from Rob Faludi's *Wireless Sensor Networks, Chapter 5: API and a Sensor Network – thanks Rob!*

This project can serve as a starting point for almost any wireless sensor network you'd like to build. Each member of the class will set up a sensor that sends three analog sensor values to generate Red, Green, and Blue (RGB) values on the screen. When we're done, we'll play a little color matching game.

Each of your sensor boards will act as router, talking to a common base station (acting as the coordinator) connected to a computer running Processing. This base station will parse out the sensor values and display them on screen.

Parts

For each sensor board:

- 1 Solderless Breadboard
- 1 XBee Series 2 Wireless Radio (configured as a ZigBee Router AT mode)
- Jumper wire / Hookup wire
- Power source (3.3v line from your Arduino, or 2AA battery pack)
- 3 Trim Pots (or other analog sensors, though the pots tend to work well for this particular exercise)
- 1 XBee breakout board

For the coordinator radio: (if you want to do this at home)

- 1 Xbee Series 2 Wireless Radio (configured as a ZigBee Coordinator API mode)
- XBee USB Explorer Board (connected to a computer running Processing)

Simple Sensor Network

SparkFun Electronics Summer Semester

Step 1: Prepare Your Coordinator Radio (if doing this at home)

Using X-CTU and your XBee Explorer, configure your Coordinator radio to be a ZigBee Coordinator in API mode (refer to the earlier handout if you get stuck). The coordinator must be in API mode for this exercise because i/o data is only delivered in API mode.

Step 2: Prepare your Router Radio

Using X-CTU and your XBee Explorer, configure your Router radio to be a ZigBee Router in AT mode (refer to the earlier handout if you get stuck).

In X-CTU or CoolTerm, make sure you have the following settings on your router radio:

PAN ID: **7777** (since all the router radios have to be on the same PAN ID as the coordinator)
ATDH: **0** (the 16-bit network address of the coordinator is always 0)

ATDL : **0** (also a 0, because that's the fixed address for the coordinator)

ATJV1: To ensure your router attempts to rejoin the network on startup

ATD02, **ATD12**, **ATD22**: These commands put Pins 0, 1, and 2 in Analog mode

ATIR3E8: Sets the sample rate at 1000 milliseconds (hex 3E8)

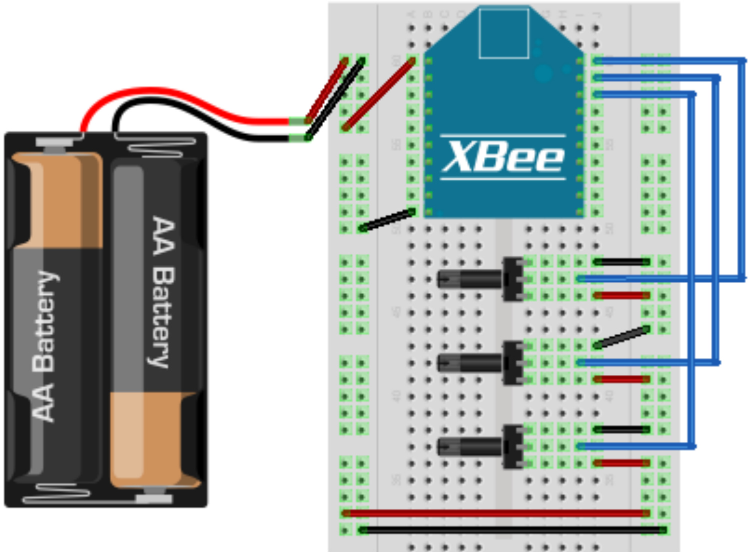
ATWR: Write your settings to non-volatile memory and sets them as the default (otherwise, they'll be reset when you unplug from the USB Explorer)

• It's never a bad idea to check that your settings are correct. To check the current value of a setting on the radio, enter the AT command without the setting number (e.g. ATDH instead of ATDH 0, ATD0 instead of ATD02, ATID instead of ATID7777, etc.) – you should get back a number that corresponds to the setting you entered.

Step 3: Prepare your sensor board

1. Place your configured Router radio in an XBee breakout shield, and place the breakout shield in your breadboard, across the middle divider.

2. Place the three potentiometers (or other analog sensors) on the breadboard, and hook them up to analog pins 0, 1, and 2 on the Xbee (see the wiring diagram below for help):



Remember the pin numbering on the Xbee (Pin 1 is on the upper left, going down the left side, pin 11 is on the bottom right, with numbers going up the right side). Don't forget to hook up power (pin1) and ground (pin10).

Note: if you're using different analog sensors, remember to hook them up properly (don't forget necessary power requirements, capacitors, resistors, voltage regulators, etc.)

Great! You should be ready to start sending your values to the controller!

Step 4 (Optional): Programming your Processing Sketch (for at-home use)

The Processing sketch for the controller XBee is a bit more complex than normal in terms of interfacing with sensors, since we have to deal with receiving them wirelessly from the router XBee's, not just through the Serial port from an Arduino. If you want to run this code at home, you'll also need the Xbee-Api Java Libraries from Andrew Rapp, which you can download here: http://code.google.com/p/xbee-api/downloads/list

You'll want to put the log4j.jar and xbee-api-version#.jar files in a 'code' folder inside your Processing sketch folder (further instructions in the comments at the beginning of the code).

```
/*
 * Matching color game from sets of Analog sensors
 * by Ben Leduc-Mills
 * based off code by Rob Faludi http://faludi.com
 */

// used for communication via xbee api
import processing.serial.*;

// xbee api libraries available at http://code.google.com/p/xbee-api/
// Download the zip file, extract it, and copy the xbee-api jar file
// and the log4j.jar file (located in the lib folder) inside a "code"
// folder under this Processing sketch's folder (save this sketch, then
// click the Sketch menu and choose Show Sketch Folder).
import com.rapplogic.xbee.api.ApiId;
import com.rapplogic.xbee.api.PacketListener;
import com.rapplogic.xbee.api.XBee;
import com.rapplogic.xbee.api.XBeeResponse;
import com.rapplogic.xbee.api.zigbee.ZNetRxIoSampleResponse;

String version = "1.1";

// *** REPLACE WITH THE SERIAL PORT (COM PORT) FOR YOUR LOCAL XBEE ***
String mySerialPort = "/dev/tty.usbserial-A40084zG";

// create and initialize a new xbee object
XBee xbee = new XBee();

int error=0;

// make an array list of thermometer objects for display
ArrayList colorBoxes = new ArrayList();
// create a font for display
PFont font;
int R, G, B;
float value1, value2, value3;  //values from the 3 sensors
int tolerance = 20;  //how close do we need to get
```

```
void setup() {
  size(850, 850); // screen size
  smooth(); // anti-aliasing for graphic display

  //set up target colors
  R = round(random(0, 255));
  G = round(random(0, 255));
  B = round(random(0, 255));

  // You'll need to generate a font before you can run this sketch.
  // Click the Tools menu and choose Create Font. Click Sans Serif,
  // choose a size of 10, and click OK.
  font =  loadFont("SansSerif-10.vlw");
  textFont(font); // use the font for text

    // The log4j.properties file is required by the xbee api library, and
  // needs to be in your data folder. You can find this file in the xbee
  // api library you downloaded earlier
  PropertyConfigurator.configure(dataPath("")+"log4j.properties");
  // Print a list in case the selected one doesn't work out
  println("Available serial ports:");
  println(Serial.list());
  try {
    // opens your serial port defined above, at 9600 baud
    xbee.open(mySerialPort, 9600);
  }
  catch (XBeeException e) {
    println("** Error opening XBee port: " + e + " **");
    println("Is your XBee plugged in to your computer?");
    println("Did you set your COM port in the code near line 20?");
    error=1;
  }
}

// draw loop executes continuously
void draw() {
  background(224); // draw a light gray background

  fill(R, G, B);  //set the fill to our randomly generated color
  rect(width/2-125, 10, 250, height-20);  //and draw a nice big rect for people to
match

  // report any serial port problems in the main window
  if (error == 1) {
    fill(0);
    text("** Error opening XBee port: **\n"+
      "Is your XBee plugged in to your computer?\n" +
      "Did you set your COM port in the code near line 20?", width/3, height/2);
  }
  SensorData data = new SensorData(); // create a data object
  data = getData(); // put data into the data object
  //data = getSimulatedData(); // uncomment this to use random data for testing
```

```
 // check that actual data came in:
  if (data.value1 >=0 && data.address != null) {

    int i;
    boolean foundIt = false;
    for (i = 0; i < colorBoxes.size(); i++) {
      if ( ((ColorBox) colorBoxes.get(i)).address.equals(data.address) ) {
        foundIt = true;
        break;
      }
    }

    value1 = data.value1;
    value2 = data.value2;
    value3 = data.value3;

    //if the box exists already, update the color values
    if (foundIt) {
      ((ColorBox) colorBoxes.get(i)).value1 =  value1;
      ((ColorBox) colorBoxes.get(i)).value2 =  value2;
      ((ColorBox) colorBoxes.get(i)).value3 =  value3;
    }

    //if there's room in the array, create a new box
    else if (colorBoxes.size() < 12) {
      // ColorBox(address, sizeX, sizeY, posX, posY)
      colorBoxes.add(new ColorBox(data.address, 260, 40, 20, (colorBoxes.size()) * 61
+ 10));
      ((ColorBox) colorBoxes.get(i)).value1 =  value1;
      ((ColorBox) colorBoxes.get(i)).value2 =  value2;
      ((ColorBox) colorBoxes.get(i)).value3 =  value3;
    }

    //draw the color boxes
    for (int j = 0; j<colorBoxes.size(); j++) {
      ((ColorBox) colorBoxes.get(j)).render();
    }

    //if the R, G and B of the box are close enough, declare a match!
    if ((value1 < R + tolerance && value1 > R - tolerance) &&
      (value2 < G + tolerance && value2 > G - tolerance) &&
      (value3 < B + tolerance && value3 > B - tolerance)) {
      println("Close?  " + R + " " + value1 + "    " + G + " " + value2 + "  " + B + "
" + value3);
      //text(((ColorBox) colorBoxes.get(i)).address + " WINS!!!", width/2-125,
height/2-125);
      fill(200, 20, 20);
      textSize(20);
      text(" WINS!!!", ((ColorBox) colorBoxes.get(i)).posX,
      ((ColorBox) colorBoxes.get(i)).posY+20);
    }
    println("Close?  " + R + " " + value1 + "    " + G + " " + value2 + "  " + B + " "
+ value3);
  }
} //end of draw loop


// defines the data object
class SensorData {
  float value1;
  float value2;
  float value3;
  String address;
}

// used only if getSimulatedData is uncommented in draw loop
//
SensorData getSimulatedData() {
  SensorData data = new SensorData();
  int value = int(random(750, 890));
  String address = "00:13:A2:00:12:34:AB:C" + str( round(random(0, 9)) );
  //data.value = value;
  data.address = address;
  delay(200);
  return data;
}

// queries the XBee for incoming I/O data frames
// and parses them into a data object
SensorData getData() {

  SensorData data = new SensorData();
  //int value = -1;      // returns an impossible value if there's an error
  float value1 = -1; //sensor values
  float value2 = -1;
  float value3 = -1;
  String address = ""; // returns a null value if there's an error

  try {
    // we wait here until a packet is received.
    XBeeResponse response = xbee.getResponse();
    // uncomment next line for additional debugging information
    //println("Received response " + response.toString());

    // check that this frame is a valid I/O sample, then parse it as such
    if (response.getApiId() == ApiId.ZNET_IO_SAMPLE_RESPONSE
      && !response.isError()) {
      ZNetRxIoSampleResponse ioSample =
        (ZNetRxIoSampleResponse)(XBeeResponse) response;

      // get the sender's 64-bit address
      int[] addressArray = ioSample.getRemoteAddress64().getAddress();
      // parse the address int array into a formatted string
      String[] hexAddress = new String[addressArray.length];
      for (int i=0; i<addressArray.length;i++) {
        // format each address byte with leading zeros:
```

```
    hexAddress[i] = String.format("%02x", addressArray[i]);
   }

   // join the array together with colons for readability:
   String senderAddress = join(hexAddress, ":");
   //print("Sender address: " + senderAddress);
   data.address = senderAddress;

   // get the value of the first input pin
   value1 = ioSample.getAnalog0();
   //print(" analog value1: " + value1 );

   value2 = ioSample.getAnalog1();
   //print(" analog value2: " + value2 );

   value3 = ioSample.getAnalog2();
   //print(" analog value3: " + value3 );

   data.value1 = value1;
   data.value2 = value2;
   data.value3 = value3;
  }
  else if (!response.isError()) {
   println("Got error in data frame");
  }
  else {
   println("Got non-i/o data frame");
  }
 }
 catch (XBeeException e) {
  println("Error receiving response: " + e);
 }
 return data; // sends the data back to the calling function
}


//Class for color boxes (takes R,G,B values from 3 different sensors)

class ColorBox {

 int sizeX, sizeY, posX, posY;
 float value1, value2, value3;
 String address;

 ColorBox(String _address, int _sizeX, int _sizeY, int _posX, int _posY) {
//initialize ColorBox object

  address = _address;
  sizeX = _sizeX;
  sizeY = _sizeY;
  posX = _posX;
  posY = _posY;
 }
```

```
  void render() {

    noStroke();
    //fill(value1, value2);
    fill(value1, value2, value3);
    rect(posX, posY, sizeX, sizeY);

    textAlign(LEFT);
    fill(0);
    textSize(10);

    text(address, posX, posY + sizeY + 10);
  }
}
```

XBee AT Commands

With the exception of the first command (+++) all the following commands should be used while in AT Command mode and the user should press enter after typing the command. It is important to remember that the user should never press the enter key after typing the Enter AT Command (+++). If you wait three seconds while inside the AT Command mode the terminal will automatically exit AT Command mode and enter back into Chat mode. All other typed AT Commands must be followed by hitting the enter key. All Commands, unless otherwise noted, are for Series 2 XBee units.

AT Command	**What Command stands for**	**What Command does**	**How to use the Command**
+++	Enter AT Command mode		Type Command and wait 3 seconds, do not hit the enter key. If you wait 10 seconds without typing anything the terminal will drop out of Command mode.
AT	Attention	XBee should reply with OK	Type Command, if there is no OK response try reentering AT Command Mode.
ATID	Personal Area Network ID	Returns PAN ID # or sets PAN ID #	To check XBee's PAN ID # type the Command. To set XBee's PAN ID # type the command. PAN IDs are represented in hexadecimal.
ATSH/ATSL	64-Bit Address (or Serial #) distinct to each XBee unit	Type ATSH to return the upper half of your XBee's Address. Type ATDL to return the lower half of your XBee's Address. The Address is split into two sections because it will not fit in one. Addresses are represented in hexadecimal.	

XBee AT Commands

SparkFun Electronics Summer Semester

AT Command	What Command stands for	What Command does	How to use the Command
ATDH/ATDL	XBee Destination Address (or Serial #)	Similar to command above the Address is split into two sections and represented in hexadecimal. Type just the command to read the Address of the other XBee the unit is trying to communicate with. To set the destination Address of an XBee type the command followed by the corresponding section of the destination Address you are setting for the XBee.	
ATCN	Command Null	Use this Command to drop out of Command mode.	
ATWR	Write	Use this Command to write the configuration you have created in the AT Command mode to the firmware. This effectively saves the configuration you have created to the XBee. If you do not use this AT Command the XBee will revert to previous settings when it is disconnected,	
ATMY	My ID	Use this Command to display the XBee's 16 bit Address in non-hexadecimal form.	
ATD0…ATD7	I/O Pin configuration	Sets the configuration of I/O pins 0 through 7.	These two Commands have a variety of settings; follow the Command with the numbers below for the purposes listed beside them. 0 : Disables I/O on that pin 1: Built in function, if available on pin 2: Analog input, only pins D0 – D3 3: Digital Input 4: Digital Output, LOW (0 volts) 5: Digital Output, HIGH (3.3 volts)
ATP0…ATP1	I/O Pin configuration	Sets the configuration of I/O pins 10 and 11.	

XBee AT Commands

SparkFun Electronics Summer Semester

AT Command	What Command stands for	What Command does	How to use the Command
ATIR	I/O Rate	Sets the sample rate of the I/O pins, designating how often the pins are read and values transmitted to other XBees	This rate is set in milliseconds using hexadecimal notation. To set the rate type the Command followed by the sample value in milli-hex. To turn off periodic sampling type this Command and enter 0 for the value.
ATIT (Series 1)	Iteration Tailor	Use this Command to set the number of samples taken from D I/O pins before the XBee transmits them	Type the Command followed by the number of samples you wish the XBee to send per transmission. Samples are stored in a buffer, they are 2 bytes in size and the buffer can store up to 90 bytes, or 22 samples, so the highest value you can pass this command is 44.
ATIA (Series 1)	Input Address	Enables pin output modes to be updated from another XBee	Type the Command followed by the address of the XBee that will be sending the output mode changing commands.
AT%V	Percent Voltage	Use this Command to display the current supply voltage for the XBee. Useful for keeping track of battery status.	
ATPR	Pull-Up Resistor	Configures internal 30K Ohm pull-up resistor on pins that have been configured as input pins	Type this Command followed by a 1 to turn the internal pull-up resistor on, replace the 1 with a 0 to turn the pull-up resistor off. Internal pull-ups are available on all input pins and are come preset on.
ATRE	Reset	Reset configuration to factory presets	Simply type this command to reset all configurations.

NewSoftSerial

As you've seen, serial is an incredibly useful way to transfer information between computers. In order to transmit and receive serial data, you need, logically enough, a serial port. This is provided by a bit of hardware called a UART (Universal Asynchronous Receiver / Transmitter) that lives on the processor chip. The UART does the hard work of generating and decoding the signals used to send data across the RX and TX lines, so all you (as a programmer) need to do, is feed it characters that you want sent, and ask it for any characters it's received while you were off doing something else. Easy!

The smaller Arduinos based on the ATmega328 (the Uno, Duemilanove, Pro and Pro Mini) have one UART, giving you one serial port. The Arduino Mega, based on the ATmega2560, has four (!) UARTs, giving you four serial ports.

Typically you'll connect one device to each serial port. On all Arduinos, the lowest numbered (or sole) serial port is connected to the main computer through the USB cable, but you could also attach it to anything else you want, such as a GPS receiver, XBee, etc.

This starts becoming a problem if you want to connect your Arduino to more than one serial device. What if you want to connect your (sole) serial port to a GPS unit, but also look at debugging information back on your PC? Or forward GPS data through a radio link, or display it on a serial LCD? This isn't a big problem on the Arduino Mega, as it has four serial ports. But on the smaller Arduinos, which have only one serial port, it can be a significant restriction.

But it turns out that what the UART does with hardware, can also be done in software. Several libraries are available that emulate hardware serial ports in software, allowing you to create additional "soft" serial ports if needed.

Arduino 0022 comes with a library called SoftwareSerial (http://arduino.cc/en/Reference/SoftwareSerial) that provides this functionality. However this library has significant limitations (9600-baud only, won't receive when it's not active, etc.). Mikal Hart has written a greatly-improved library called NewSoftSerial, which solves most of these problems. For now you'll need to download and install the library yourself, but it is expected that NewSoftSerial will replace SoftwareSerial in the official Arduino IDE at some point in the future.

XBee Wireless, New SoftSerial Library

SparkFun Electronics Summer Semester

To install the NewSoftSerial library:

1. If it doesn't already exist, create a "libraries" folder in your Arduino sketches folder.
2. Download the latest library from http://arduiniana.org/libraries/newsoftserial/
3. Open the zip file
4. Drag the NewSoftSerial folder into the "libraries" folder you just made.
5. Start or restart the Arduino IDE. NewSoftSerial should now be listed in the "Sketch / Import library" menu.

You use NewSoftSerial very much like the hardware serial port. The only extra steps are to #include the library and set up the new port at the beginning of your sketch:

```
#include <NewSoftSerial.h>

const int RXpin = 2;
const int TXpin = 3;
NewSoftSerial myPort(RXpin, TXpin);
```

You can of course name the port anything you want (it's often useful to name the port after its function, such as "GPS" or "display"). For the rest of your program, you use NewSoftSerial exactly like a normal serial port:

```
void setup()
{
 myPort.begin(9600);
 myPort.println("hello, world!");
}

void loop()
{
 char c;

 // check if there are received characters
 if (myPort.available() > 0)
 {
      c = myPort.read();
      // do something with c here
 }
}
```

There's some fine print you should be aware of. Generating (and receiving) serial data in software is extremely processor-intensive, since it needs to send (and receive) every bit in every character at *exactly* the right time, thousands of times per second. The problem becomes worse at high serial speeds, and the more soft serial ports you create. Once the processor can't keep up with all those bits, you'll start losing data. Here are some tips to help you get the most out of soft serial ports:

- When choosing which ports to use, save the hardware port for the fastest / heaviest connection you have. Use soft serial ports for low speed / lightly used connections if possible. (However, if you're using the USB serial link back to your PC, you'll have to use the hardware port for that).

- Keep your soft serial rates as low as possible, and between 9600 and 57600 baud. Other baud rates (lower or higher) may not work reliably.

- Soft serial ports may interfere with other interrupt-driven libraries, such as Servo. Google for advice in these cases.

- If you create more than one soft serial port, only one of them (the one you last used) will be able to receive data at a time. This may not be a huge problem, as you can often structure your program to access them sequentially as required.

- The NewSoftSerial library is most useful on the smaller Arduinos that only have one serial port. However, it can be of use on the Mega as well. Check the NewSoftSerial webpage (http://arduiniana.org/libraries/newsoftserial/) for the latest news as to compatibility with the Mega. (As of this writing, version 10C does not fully support the Mega, however beta version 11 does on certain pins).

Common XBee Mistakes

Your XBee project isn’t working? Here are some common mistakes that both beginners and experts make:

- Not using the latest firmware (especially if ATD0 or ATIR is giving an error)
- No reference voltage to VREF pin on the 802.15.4 radios (analog and digital reads give wrong values)
- Forgetting that AT commands use hexadecimals
- Hitting return after +++ (or otherwise not respecting 1 second default guard time)
- Conversely, _not_ hitting return after an AT command
- Letting the XBee time out of command mode before issuing an AT command (you’ll know because you get no response)
- Forgetting to write the configuration to firmware with ATWR (unless your application configures the radio interactively)
- Not using ATRE (restore factory defaults) before re-configuring a previously used radio (previous settings lurk unless you manually reset them all)
- Looking for analog output on the analog input pins instead of pins 6 and 7 (P0, P1)
- Using a voltage regulator without decoupling capacitors (10uF on input, 1uF on output is good)
- Mixing up TX and RX pins (fastest way to check this is switch the wires and see if it starts working)
- Using ZigBee version (ZB Pro or ZNet 2.5) when 802.15.4 version would do just fine (if you don’t need to make a mesh network)
- Trying to read more than 1.2 Volts on the ZB Pro and ZNet 2.5 analog inputs (that’s the limit)
- Buying Pro radios when you don’t need them. (Cost more, bigger, use a lot more battery)
- Deciding the XBees are flaky. (You may not be using them correctly, but they are very reliable)

Common Mistakes with XBee Radios by Rob Faludi

SparkFun Electronics Summer Semester

- Deciding an XBee is burned out when it's set to a different baud rate (check ON and ASSC lights)
- Deciding an XBee is burned out when it is just sleeping (Check ON light to see if it blinks occasionally)
- Forgetting to supply power or ground (ON light may go on and ASSC light may blink but both will be significantly dimmer)
- Not contacting Digi sooner for support, especially if your radio seems dead or you keep getting an error you don't understand.

- XBee Arduino Mistakes
- Sending continuously without any delay (try 10ms delay)
- Not removing RX and TX connections before uploading code (Arduino will give an error)
- Not removing RX connection when reseting, if you are continuously receiving data. (Arduino will never reset)

- XBee LilyPad Mistakes
- Hooking up more than 4 Volts to the 3.3V pin
- Using switches without pull-down resistors (but not if you use the internal pull-ups)
- Not using a pull-up or pull-down resistor on pins 5 and 7 (these don't have internal pull-ups at all)
- Using sensors without voltage divider resistors (if your sensor needs that circuit)
- Using too-resistive conductive thread for power and ground (try fabric or wires)

- Windows and X-CTU are required for firmware upgrades

sparkfun ELECTRONICS

WEBSITE: sparkfun.com
6175 LONGBOW DRIVE, SUITE 200
BOULDER. COLORADO USA
ZIP CODE: 80301
[303] 284.0979 [GENERAL] P
443.0048 F

GPS Overview

SparkFun Electronics Summer Semester

Introduction to GPS

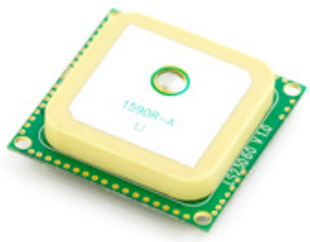
The goal of this handout is to show you how to use GPS NMEA data and demonstrate some of the features and hardware one finds in embedded GPS modules/receivers. In addition, I will show how the features and hardware can affect tracking accuracy. This is not a buying guide per se, but it will help you get an idea of the differences, shown in Google Earth, between chipsets, antennas, and update rates for the six SparkFun GPS modules seen below.

SUP500F

EM406

Copernicus

LS23060



uMini

D2523

I choose modules with different chipsets, antenna/hardware configurations, and update rates in order to get a general idea of the wide variety of features. All modules were left in their default states (except for the SUP500F which I had to configure for 10Hz update rate). The Copernicus and uMini use SMDGPS modules and are assembled by SparkFun. The other four are from outside suppliers.

GPS Overview

SparkFun Electronics Summer Semester

GPS Module	Chipset	Antenna	Update Rate
Copernicus	Trimble Trim Core	ceramic patch	1Hz
SUP500F	Venus634	small ceramic patch	10Hz
MN5010 (uMini)	SiRF Star III	chip	1Hz
LS23060	MediaTek MT3318	ceramic patch	5Hz
DS2523T	U-Blox 5	helical	4Hz
EM-406A	SiRF Star III	ceramic patch	1Hz

Refer to individual product pages and datasheets for the detailed technical information for each module.

Building the tester

Here is the bill of materials (BOM) for testing each GPS module:

- Breadboard Power Supply Stick 3.3/5V
- LiPo Battery Pack 1000mAh/7.2V
- openlog
- uSD 1GB
- jumper wires and headers

The following are some issues that I had to consider when designing the test rig.

GPS modules are radio frequency receivers.

For reliable performance, your GPS module needs to receive good unobstructed signals from the satellite constellation. GPS modules have relatively tiny antennas to receive data from satellites moving at relativistic orbital speeds, over 10,000 miles away, and so they need somewhat of a clear view of the sky. I know this might sound really obvious, but I cannot stress this enough, because...

GPS sensitivity and accuracy is highly dependent on environmental conditions.

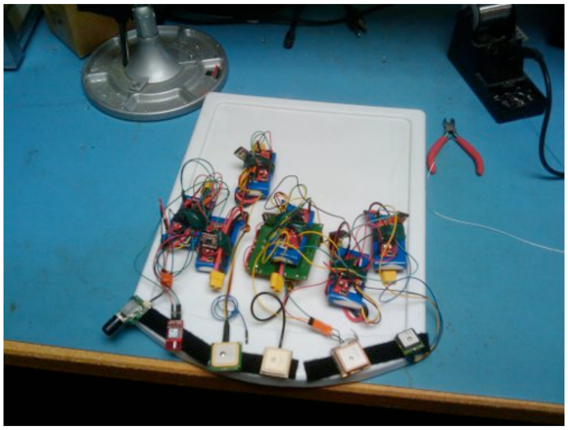
Not only does this include physical objects, like trees, earth, and structures, but also includes atmospheric effects, space weather, the overall health of the GPS constellation, and multipath interactions, all of which contribute to constant variability in receiving the GPS signal. To minimize errors due to these environmental effects, I wanted to test and track all six modules at the same time with varying views of the sky.

What I needed was a mobile logger for each GPS module. SparkFun stocks more than enough supplies to accomplish this.

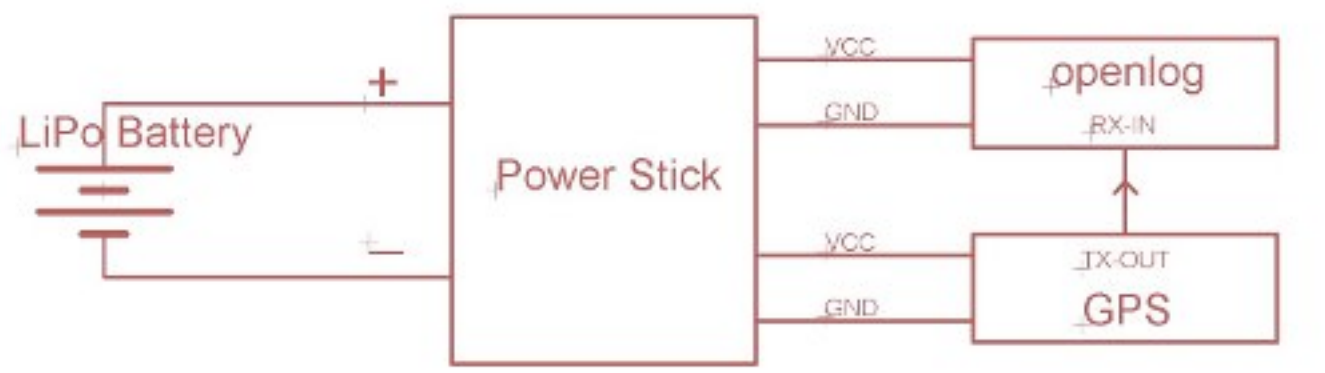
SparkFun Electronics now carries a GPS shield unit that will make hooking up your GPS module to your Arduino as easy as soldering some headers and placing the shield onto your Arduino.

GPS Overview

SparkFun Electronics Summer Semester



Tracking Setup. Note: The uMini was oriented vertically for the test and is not shown that way in the picture



Connection Block Diagram

Each GPS module should have its own clean (regulated and decoupled) DC power supply.

The GPS power supply feeds into the antenna power supply on some of the modules, so any excessive noise or unexpected loading on the circuit can affect performance. Attaching a high capacity LiPo battery to a regulated Breadboard Power Supply Stick for the power supply for each module is sufficient.

Next, I capture the data coming out of the GPS module with the openlog.

All that needs to be done is to connect the transmit pin (TX) from the GPS module to the receive (RX) pin on the openlog. The openlog will log anything it sees on that pin, just be sure to configure the baud rate on the openlog to match the baud rate coming out of the GPS module.

A brief word or two on the VBATT pin and the TTFF (time to first fix)*.*

What is the VBATT pin used for? Some embedded GPS modules contain volatile RAM that is used to store the almanac and ephemeris data of the satellites, which is globally updated on a regular basis. Think of the almanac data as a "bus schedule" of when and where each satellite will be in the sky. The ephemeris data contains the errors in the schedule. If your GPS receiver does not have the schedule or the errors of the satellites, it will have to download that information when there are updates. Since the RAM stores this data, the VBATT pin should be powered or connected to some kind of energy supply (i.e. supply voltage, battery, or supercapcitor), so that your GPS module can quickly start-up. If you don't power the RAM, all of the almanac and ephemeris data is lost with each power cycle.

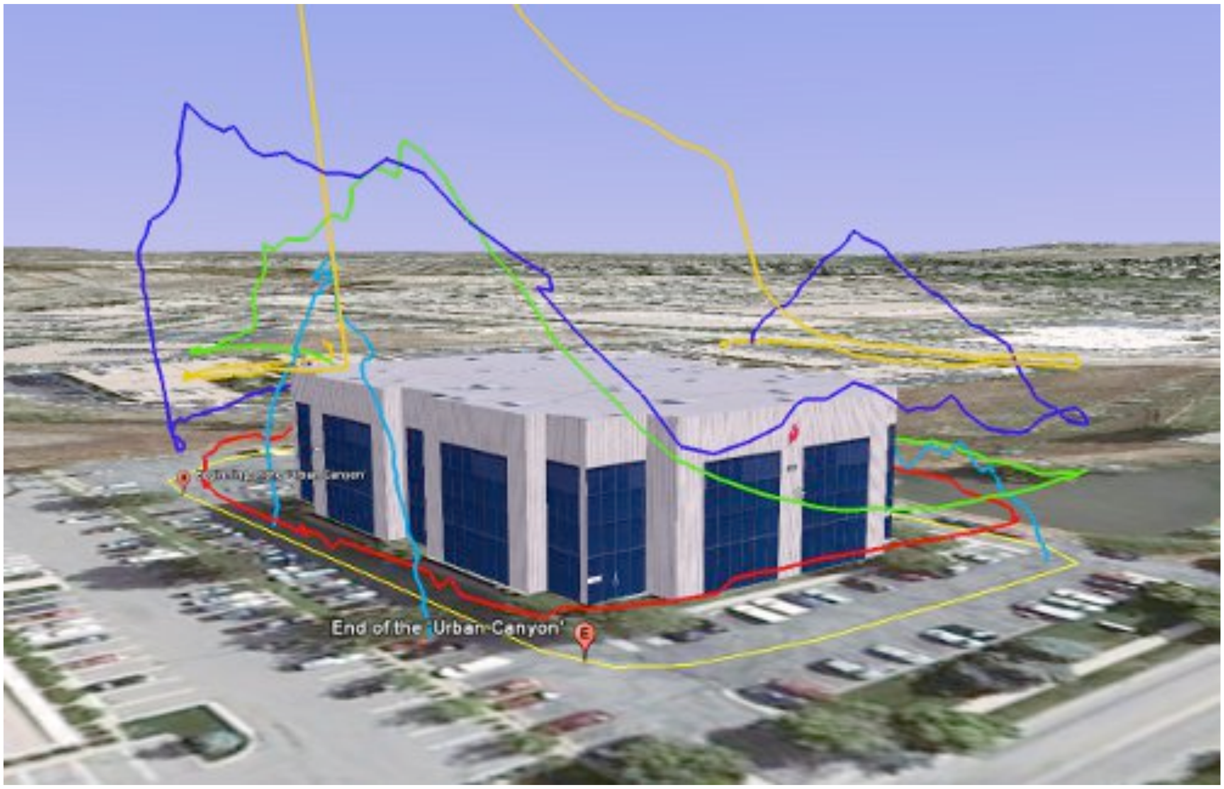
The TTFF (time to first fix) is determined by how quickly the GPS module can access the almanac and ephemeris data. If the module doesn't have any of this data (some modules don't have this data out-of-the-box) the module might take a bit of time to power up. Same goes for if you don't connect power to the VBATT (RAM power) pin, you will lose all of the data upon a power cycle. If the almanac and ephemeris data can be permanently stored in flash, like the SUP500F can, then you should get a really fast TTFF. Some modules, like the EM406, have a supercapacitor attached to the RAM. This means you will get really quick TTFF for only a period of time after you power the module down. Wait too long, the capacitor will discharge and you will lose your information. The reason for the choice in using a supercap over a battery or flash chip, is that it is in expensive, small, and the almanac and ephemeris data needs to be updated regularly anyway, so if your position is changing a great deal over the globe, you don't need to keep all of the same almanac data.

In terms of the TTFF performance for each of these modules, all have similar stated fix times that are under 1 minute, but will differ with environmental conditions and antenna hardware.

Tracking

Download Google Earth to view the KMZ files and zoom around my track in 3D (be sure to turn on 3D buildings). Just click the link and Google Earth should open and zoom to the SparkFun building.

- raw KMZ
- edited for locks KMZ

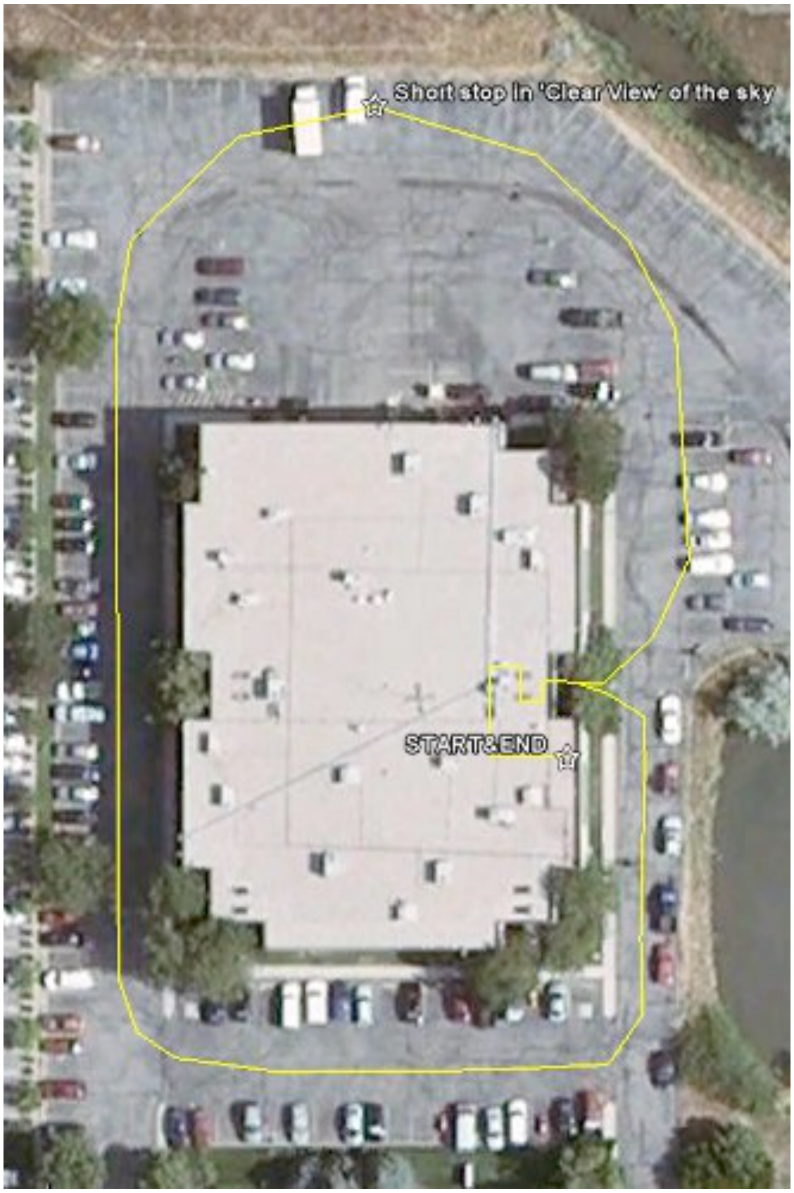


Overview

For my track, I decided to walk the GPS modules around the SparkFun building, instead of testing them in a car. **GPS modules, to an extent, seem to work better the faster they move**, so walking the units around the building should create a test condition where the GPS modules will have a greater chance to show errors.

GPS Overview

SparkFun Electronics Summer Semester



Actual track walked (top is Northish).

Here is a brief summary of my track: I turned all of the module's power ON next to a window on the second floor of the SparkFun building. I waited a few minutes, then made my way outside to the north most point of the parking lot, where I waited to make sure all of the modules had a lock before I proceeded south around the building and back to where I started.

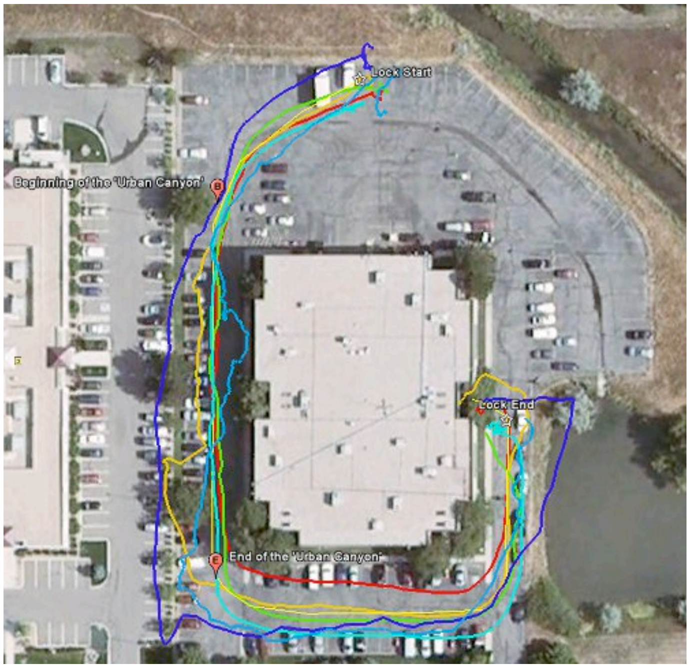
What does it mean when the receiver has a lock or a fix?

When the receiver has a lock, you get accurate position (3D) and time data. This is accomplished when the receiver sees at least four satellites in good view. Why four? You would think you only need three for triangulation, right? You forgot about time! Remember, with four unknown variables you need four equations. The fourth signal gives enough information for your GPS unit to compute all three positions (latitude, longitude, elevation) and time.

Getting accurate time is not an easy task for satellites and receivers alike. The GPS satellites need really, I mean really, accurate timing. So accurate, that there are actually atomic clocks on board the satellites in order to keep them in sync. Since the speeds and distances are so great between satellites and receivers, relativistic offsets are programmed into the satellites position and time data. This information, along with additional correction data the satellites receive from permanent ground based stations (this is how WAAS works), turns your GPS receiver into an amazingly accurate and powerful little device with, on average, five meter accuracy.

RED = Copernicus, BLUE = uMini, GREEN = EM406, ORANGE = D2523, LIGHT BLUE = SUP500, GREEN-BLUE = LS23060, YELLOW = actual path

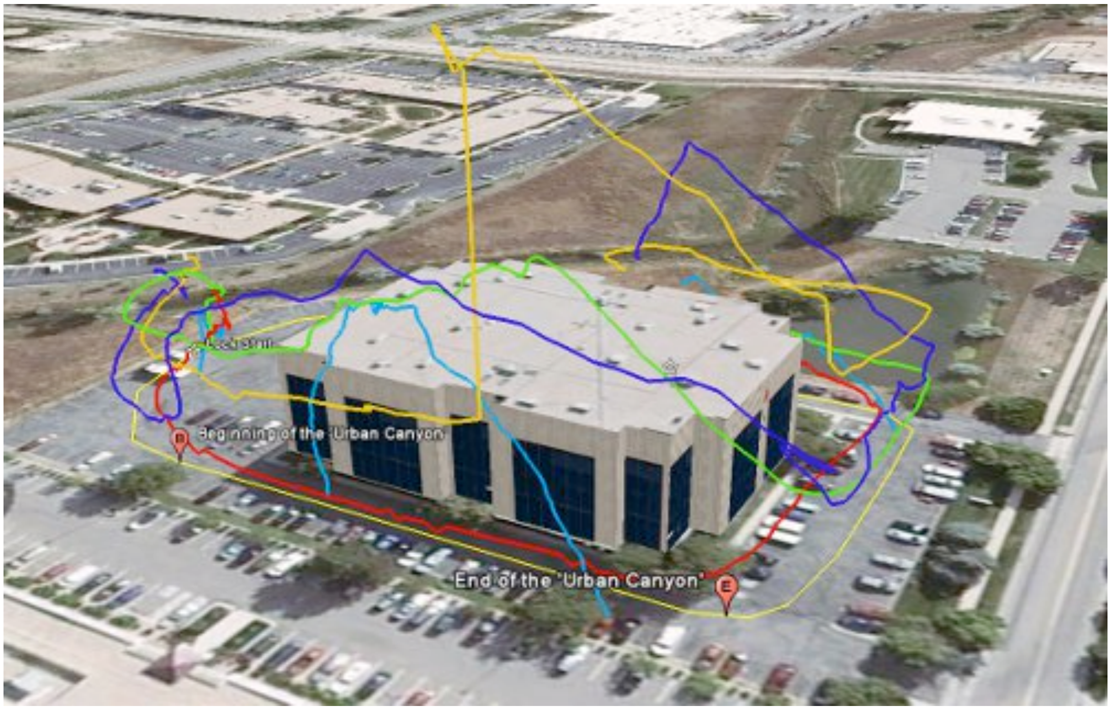
North is at the top



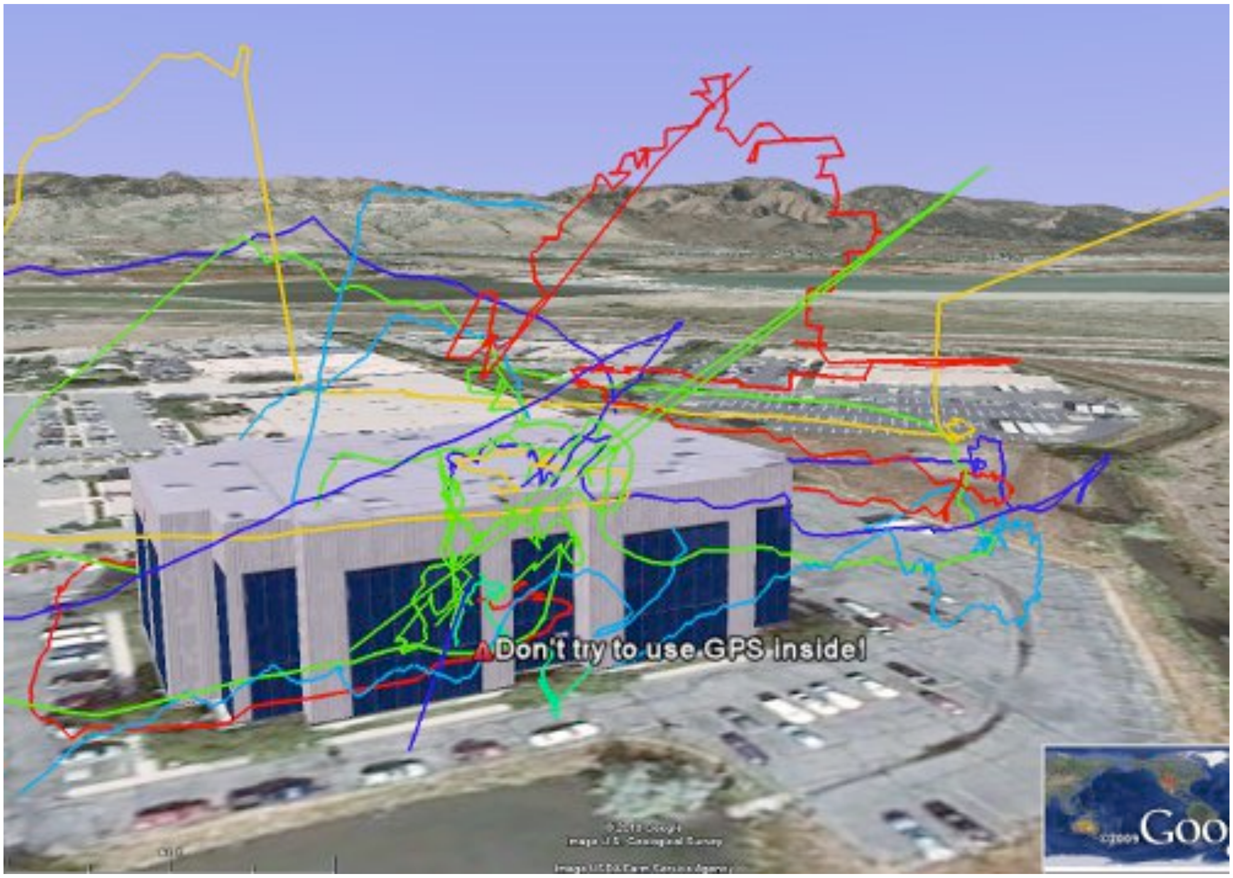
This view only shows latitude and longitude data (no elevation) and I only am showing the tracks *after* I waited in clear view of the sky to make sure all of the modules had a lock. Notice that all of the tracks in clear view of the sky (to the north and south) are decently accurate; within the claimed 1-5 meter accuracy for C/A code receivers.

There is about a 150 foot span in between the ~25 foot tall SparkFun building and our neighbors ~25 foot tall building to the west that creates a small 'urban canyon' where most of the modules had some expected trouble. In addition, there is tree cover in part of the SparkFun urban canyon, which also hindered the accuracy of the GPS modules.

Notice how the SUP500F's (light blue) track is not a straight as the other tracks, this is because the position is being updated at 10 times a second compared to one time a second for most of the other receivers. Modules with faster update rates usually work better within faster moving objects.



This is a 3D view looking at the west side of the building. Looks like the D2523 (orange) had a hiccup in the urban canyon. Also notice the uMini (blue) is not performing as well as the others. This is expected since the uMini is using a passive chip antenna. And wait...I only see five tracks, where is the sixth? The LS23060 module's elevation data is below what Google Earth calls ground height. It is there, just not visible in this view.


Don't try to use GPS inside!
© 2011 Google
Image U.S. Geological Survey
Image USDA Farm Service Agency
©2009
Goo

Don't try to use GPS inside!

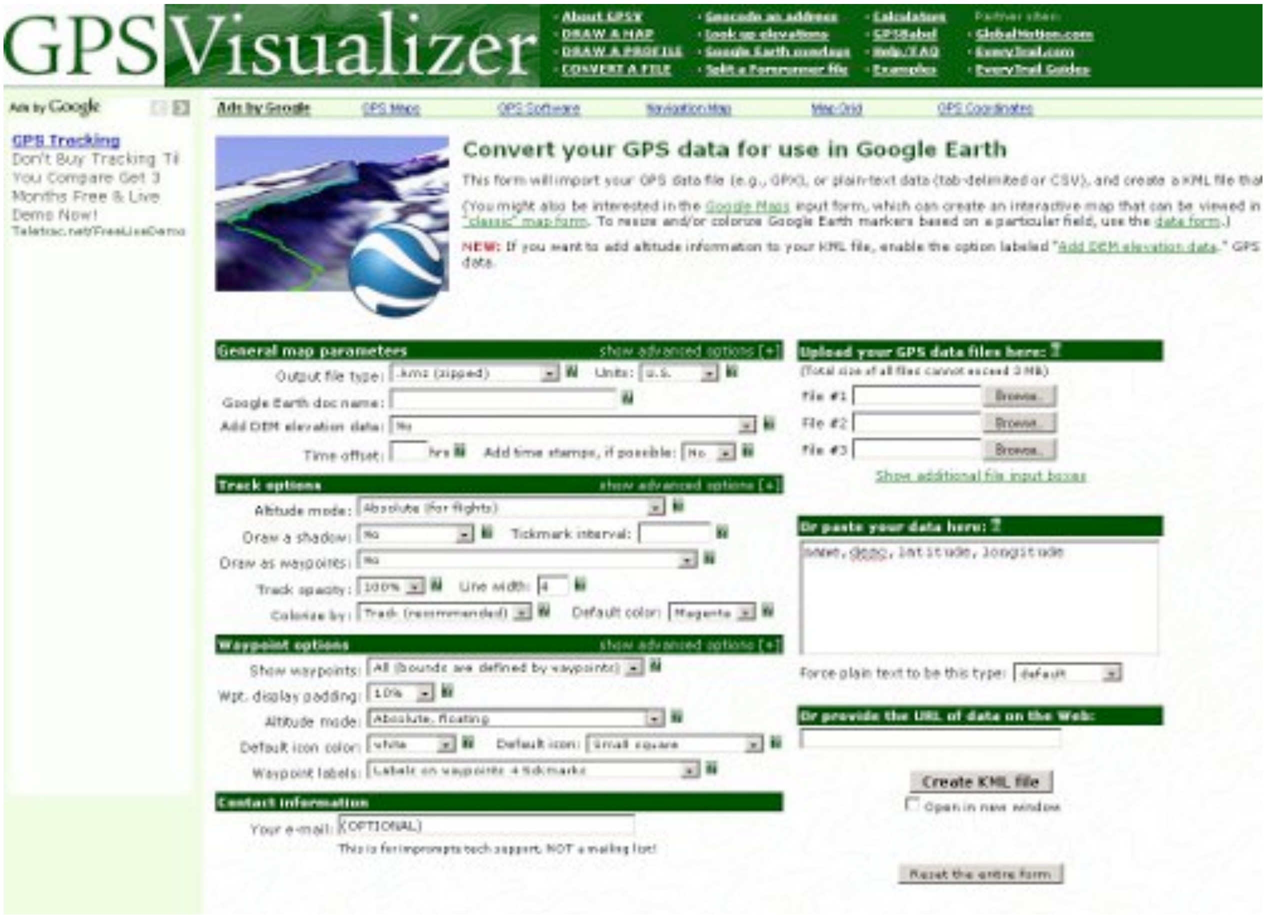
Wow, what is that? Those are the tracks of the GPS modules trying to find their location inside of the building when I first turned the units on. **This is why you don't rely on GPS inside of a structure.**

How to create files in Google Earth

Overlaying your tracks in Google Earth is really easy. After you have logged your positions to the openlog, remove the SD card and plug it into a SD card reader on your computer. Look at the contents on the card. You should see a config file and at least one log file (if you cycled power during logging, new log files would have been created). Open the log file with a text editor, like Notepad in Windows. You can see a bunch of text in the form of NMEA sentences or something in a similar format. The NMEA sentences contain, among other data, your position and time.

Here are my raw text files: http://www.sparkfun.com/tutorial/GPSTracking/raw_5-20-10.zip

All you need to do now is convert the file to a Google Earth file with a kml or kmz file extension. The website I use to convert the text file to a Google Earth file is GPSVisualizer. If you click on the Google Earth KML link in the middle of the homepage you can configure settings.



Convert your GPS data for use in Google Earth

This form will import your GPS data file (e.g., GPX), or plain-text data (tab-delimited or CSV), and create a KML file that

(You might also be interested in the Google Maps input form, which can create an interactive map that can be viewed in "classic" map form. To resize and/or colorize Google Earth markers based on a particular field, use the data form.)

NEW: If you want to add altitude information to your KML file, enable the option labeled "Add DEM elevation data." GPS data.

General map parameters — show advanced options [+]

Output file type: .kmz (zipped) — Units: U.S.
Google Earth doc name:
Add DEM elevation data: No
Time offset: hrs — Add time stamps, if possible: No

Track options — show advanced options [+]

Altitude mode: Absolute (for flights)
Draw a shadow: No — Tickmark interval:
Draw as waypoints: No
Track opacity: 100% — Line width: 4
Colorize by: Track (recommended) — Default color: Magenta

Waypoint options — show advanced options [+]

Show waypoints: All (bounds are defined by waypoints)
Wpt. display padding: 10%
Altitude mode: Absolute, floating
Default icon color: white — Default icon: Small square
Waypoint labels: Labels on waypoints + tickmarks

Contact information

Your e-mail: (OPTIONAL)
This is for impromptu tech support, NOT a mailing list!

Upload your GPS data files here:
(Total size of all files cannot exceed 3 MB)
File #1 Browse...
File #2 Browse...
File #3 Browse...
Show additional file input boxes

Or paste your data here:
name,desc,latitude,longitude

Force plain text to be this type: default

Or provide the URL of data on the Web:

Create KML file
□ Open in new window

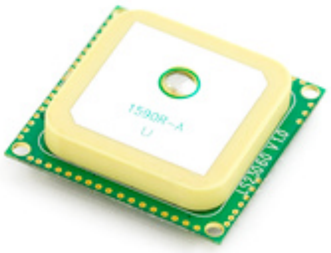
Reset the entire form

The settings I used are shown on the previous page. Once you have loaded your data into this page, hit the Create KML file. You will then be presented with a link. You can either click it and, if you have Google Earth installed, it will zoom to your tracks or you can right click and save as to save for future use or to show your friends.

If your txt files are over 3MB, you will not be able to use GPSVisualizer. GPSBabel is another site you can try if you have really big log files.

Tracking Summary

This is by no means an "end-all-be-all" summarization; it is merely a comparison on how well 6 very different GPS modules performed at the SparkFun location (not in any kind of order).

Copernicus Pros • Consistently the most accurate and reliable • Excellent elevation data in all conditions • Very good latitude and longitude data in all conditions • Very good multipath jamming, not as susceptible to interference • Least troublesome • Optional antenna • Excellent support and configuration software Copernicus Cons • SMD, can be hard to solder • Large, needs an external antenna • More than the basic (VCC, GND, TX) connections are needed to get up and running	
LS23060 Pros • One of the best along with the Copernicus • Excellent latitude and longitude in all conditions • Good elevation in all conditions • Compact and self contained • 5Hz update rate • Fix indicator LED LS23060 Cons • Solder pads for connection, have to solder on header pins or make your own cable	

GPS Overview

SparkFun Electronics Summer Semester

D2523 Pros • Helical Antenna, good for conditions where the orientation will be unknown or otherwise surrounded by high dielectric materials (i.e. human flesh) • Excellent latitude and longitude data with unobstructed view of the sky • Good elevation data with unobstructed view of the sky • 4Hz update rate • Small package for what you get D2523 Cons • Below average for all data when in the urban canyon at SparkFun	
SUP500F Pros • Very small • Very low power • Relatively inexpensive • Configurable 10Hz update rate • Excellent configuration software SUP500F Cons • Average accuracy and reliability, does much better at default update rates and higher ground speeds • Not a very common chipset, Venus 6 • Might take some time for the initial lock (fresh out of the box)	

MN5010 (uMini) Pros • Very small • Very good accuracy for a chip antenna! • Inexpensive BOM • Simple configuration, minimal pin connections MN5010 (uMini) Cons • Chip antennas are not as accurate as the ceramic patch antennas • Greater susceptibility to multipath interference with chip antenna	
EM-406A Pros • Tried-Tested-and-True • Fix indicator LED • Excellent tracking data in all conditions • PCB is completely shielded • Plenty of online support EM-406A Cons • 5V operation	

Conclusion

We now have a pretty good idea of the general performance characteristics for some of SparkFun's GPS modules, however this is obviously not all-inclusive, in terms of the performance capabilities for other GPS modules with similar features out on the market today. GPS modules with combinations of the same chipset, hardware, antennas, and configurations all will have different performance characteristics in different environmental situations. All have their benefits and costs, but it will be up to you to take the data points I have found to help choose the correct GPS module for your project.

Glossary:

ADC: Analog to Digital Converter. Any method of converting an analog signal (a voltage) to a digital signal (a number). Here is the equation necessary to do this:

$$ADC\ Value = \frac{(Voltage\ on\ Pin(mV)) * (Max.\ ADC\ Value)}{(System\ Voltage(mV))}$$

Analog: A measurement or signal that has values between On and Off. Examples include voltage, pressure, any type of wave, and volume. One useful metaphor for teaching is the zipper (if you have a zipper on your jacket or hoodie). In comparison to the button (Digital) the zipper has many states between completely open and closed. Analog's counter part is Digital.

API mode: One of the firmware modes for XBee radios, particularly important for series 2 Xbee radios, as the coordinator radio needs to be in API mode to receive i/o data. You can change the firmware to API mode using the free X-CTU software from Digi.

array: Arrays are a way to store several variables in a list, or array. In Arduino arrays are declared in a couple different ways. Example: *int arrayName [6];* this will create an array called arrayName with six spaces for variables inside it. A value of 100 would be assigned to the first space in arrayName like this: *arrayName [0] = 100;* (The first value is assigned to the 0 slot.) Here is an example of declaring and assigning values in an array at the same time: *arrayName [6] = {100, 150, 300, 50, 100, 120};* Here is an example of referencing the sixth value in the array arrayName: *arrayName [5]*.

Bias: A state describing voltage and current in a component.

Boolean: A variable type or form of logic based on the assumption that any given variable or state can only have one of two values; true or false, HIGH or LOW, 1 or 0.

bounce: Bounce occurs when a switch (or other type of input) attempts to change it's position to open or closed but does not stay in that position. Due to this you will see the electrical signal rising and falling when it should be at a constant value. If you're experiencing bounce issues, try using the `delay( )` function.

button: A digital input with only two states; pressed or not pressed. Depending on the layout of the circuit these two states can correspond to either HIGH or LOW.

char: Variable type character, 8-bit size, any single symbol. Examples: A, a, 1, or !

Glossary

SparkFun Electronics Summer Semester

command mode: The mode in which you can set certain behaviors of an XBee radio using AT commands. You can enter command mode by sending '+++' without a carriage return to the radio over serial. The radio requires a second of silence before and after in order to enter command mode. The radio will exit command mode after 10 seconds if it does not receive a command. (See the AT commands breakdown in this binder for most of the AT command set).

comments: Used to write notes in code that are not part of the execution. For a single line use //, for a block of lines start with /* and end with */

constrain: A function used to constrain a value between a given range. Example: *sensVal = constrain (sensVal, 10, 150);* This constrains the sensVal value between 10 and 150. If it is under 10 sensVal is assigned 10, if it is over 150 sensVal is assigned 150.

coordinator: The coordinator radio is the one and only node in a wireless sensor network that is responsible for receiving and coordinating the flow of data between other radios across the network. Router radios can also pass on messages, but every network must have one and only one coordinator radio.

current: This can refer to either the the flow of electrical charge or the rate of flow of electrical charge, measured in mA.

Digital: A measurement or signal that has only two values, On and Off. On and Off can also be expressed as HIGH and LOW, as well as 1 and 0. Examples include Boolean logic, open or closed and button state. One useful metaphor for teaching is the button (if you have a button on your jacket or hoodie). In comparison to the zipper (Analog) the button has only two states; connected and unconnected. Digital's counter part is Analog.

diode: A two terminal electrical component that only conducts in one direction.

Ground: In electrical engineering, ground or earth may be the reference point in an electrical circuit from which other voltages are measured, or a common return path for electric current, or a direct physical connection to the Earth.

firmware: Generally, firmware refers to any program that is loaded onto a chip to run logic in a circuit. In XBee radios, you will have to change and occasionally update the firmware on each radio depending on the functions you want them to perform within your network.

float (signal): A signal due to a pin that is not attached to anything. A floating pin can read anything between HIGH and LOW.

float (data type): Variable type float, used for floating point operations (i.e. numbers with decimal points).

for: for(variable declaration and assignment; condition; variable increment){ }
A form of iteration, code inside the curly brackets will repeat until the condition is false. Example of a for loop that will loop four times:

```
for (i = 0, i < 4, i++) {//do this code each time};
```

forward bias: The state of a component describing voltage saturation necessary to activate a component.

footprint: A footprint is the pattern on a circuit board to which your parts are attached. This includes the electrical connections and silkscreen.

flyback diode: Used to reduce voltage spikes seen across inductive loads due to a sudden loss in voltage from the power source.

hexadecimal: In mathematics and computer science, **hexadecimal** (also **base 16**, or **hex**) is a positional numeral system with a base of 16. It uses sixteen distinct symbols, most often the symbols **0**–**9** to represent values zero to nine, and **A**, **B**, **C**, **D**, **E**, **F** (or alternatively **a**–**f**) to represent values ten to fifteen. For example, the hexadecimal number 2AF3 is equal, in decimal, to $(2 \times 16^3) + (10 \times 16^2) + (15 \times 16^1) + (3 \times 16^0)$, or 10,995. Often you will need to add hexadecimal notation to the beginning of a number represented in hexadecimal. To do this type “0x” before the number.

if statements: if(condition){ }
else if (condition) { }
else { }
This will execute the code inside the curly brackets if the condition is true, and if the condition is false it will test the “else if” condition. If the “else if” condition is true it will execute the code in the second set of curly brackets. Otherwise it will execute the code inside the third set of curly brackets.

Input: A signal or data received by a processor.

int: Variable type integer, 16-bit size, any number between -32,768 and 32,767.

iteration: When something happens over and over and over, but changes a little each time.

lead: Electrical contacts for parts, these usually look like wires or pins extending off the part.

Glossary

SparkFun Electronics Summer Semester

LED: A light emitting diode.

Library: A collection of code that has been packaged so it can be included and then used in code. Servo example: `#include <Servo.h>` (This includes the library so it can be used) `Servo myservo;` (This creates a servo object so the functions inside the library can be used) `myservo.write(90);` (This uses the write function in the servo library, setting the servo's angle to 90 degrees.

loop ( ): Looks like- `void loop ( ){ }` The main loop in an Arduino sketch, this where the action happens. The `loop ( )` function is present in every single Arduino sketch. The Arduino will execute the code inside the curly brackets, once it has finished this code it will start over from the beginning of the loop function.

map: A function used to re-map a value between a range to a value between another given range. Example: `sensVal = map (sensVal, 10, 150, 100, 1500);` This maps the sensVal value from somewhere between 10 and 150 to somewhere between 100 and 1500 proportionally.

microcontroller: A tiny computer on a single integrated circuit with a core processor, memory and programmable input/output.

motor: An electrical component that converts electrical energy to mechanical energy.

Ohm's Law: V = I * R where V is Voltage, I is Current and R is Resistance. A handy way to figure out any one of these values given the other two. The complicated version: Ohm's law states that the current through a conductor between two points is directly proportional to the potential difference or voltage across the two points, and inversely proportional to the resistance between them.

operators: Similar to mathematical symbols, pay special attention to the difference between = and ==.

output: A signal or data transmitted by a processor.

PAN ID: Personal Area Network ID. A number used to define which Xbee Radios can talk to each other.

PAN Address: A hexadecimal number in the range 0x0 – 0xFFFF that defines which Network the Xbee will attach to.

piezo element: A digital component which moves a disc to one of two possible positions to create an analog output via vibration, often used to create annoying melodies with little aesthetic value.

Pin: A place to connect electrical circuits to one another.

pinMode: Arduino command used to set pins to either INPUT or OUTPUT.
Looks like- `pinMode ( pinNumber, value );` where pinNumber is the pin to be set and value is either INPUT or OUTPUT. pinMode can also be used to turn Analog In pins into Digital pins.

potentiometer: A voltage divider with a dial to change the values of the two resistors inside.

polarity: Electical polarity is present is every circuit we make, and refers to the flow of electrons through a circuit (typically from negative to positive). In DC circuits, one pole is always positive and one pole is always negative. Often we call a specific component ‘polarized’ if it requires a specific orientation within the circuit (e.g. since diodes block current in one direction, we want to make sure they are placed so that they allow current in the direction that we need).

pseudo-code: A human-readable way of describing a computer program so that it is easier to understand. See the description of ‘if statements’ in the glossary for an example.

Pulse Width Modulation: A commonly used technique for controlling digital systems to create a simulated analog output. Often abbreviated as PWM.9

relay: An electrically operated switch.

Resistance: A measure of the opposition to electrical current. Measured in Ω (ohms).

resistor: Component used to restrict the amount of current that can flow through a circuit. Rated in Ω (ohms).

router: A router can refer to any device that can receive and re-route data packets towards a specific destination. In XBee radios, a radio in Router mode performs a similar function: to send, receive, and route data from other radios to their intended receivers. There can be zero, one, or many routers within any wireless sensor network design.

Serial: universal asynchronous receiver/transmitter.

Serial Monitor: Window in Arduino programming environment that allows the user to monitor serial communication.

Glossary

SparkFun Electronics Summer Semester

setup (): Looks like- `void setup ( ){ }` All the code between the curly brackets will run once when the Arduino program first starts. (As well as each time it is restarted.)

shift register: In this case a chip which allows you to change the value of it's output pins, starting with either the first or last pin, by “shifting” a new value in and “shifting” all the old values towards the opposite end of the chip. The shift register uses three pins (latch, clock, and data) to control eight output pins.

sketch: The code created in your Arduino environment which is then saved onto your Arduino board to make something happen.

sketchbook: Folder where your sketches are stored.

temperature sensor: A sensor used to convert temperature to an analog reading, in this case a voltage, which can be read by your Arduino.

trace: A copper path on a PCB necessary for electrical conductivity between parts.

transistor: A semiconductor device used to amplify or control an electronic signal.

value: Worth or symbol stored in a variable.

variable: A symbolic name associated with a value and whose associated value may be changed.

voltage: The difference in electrical potential between any two given points of a circuit.

voltage saturation: When a component has the necessary voltage present to allow it to operate.

Xbee: Low power radios that use the IEEE 802 standard for Personal Area Networking. These units operate at 2.4 Ghz.