# INTERACTIVE PROTOTYPING

AN INTRODUCTION TO PHYSICAL COMPUTING USING ARDUINO, GRASSHOPPER, AND FIREFLY

# PREFACE

In 1991, Mark Weiser published a paper in Scientific American titled, *The Computer for the 21st Century*, where he predicted that as technology advanced, becoming cheaper, smaller, and more powerful, it would begin to "recede into the background of our lives" - taking a more camouflaged, lifestyle-integrated form. He called this *Ubiquitous Computing* (Ubicomp for short), or the age of calm technology. There have been numerous examples to support Weiser's claim, including Natalie Jeremijinko's "Live Wire" project (1995), the Ambient Orb (2002), or the Microsoft Surface Table (2007) to name just a few.

In 1997 Hiroshi Ishii expanded Weiser's idea in a seminal paper titled, *Tangible Bits* where he examined how architectural spaces could be transformed through the coupling of digital information (bits) with tangible objects. Where Wieser's research aimed to make the computer 'invisible' by embedding smaller and smaller computer terminals into everyday objects, Ishii looked to change the way people created and interacted with digitally augmented spaces.

Both Weiser and Ishii have had a significant impact on the development of *physical computing*, a term used to describe a field of research interested in the construction of physical systems that can sense and respond to their surroundings through the use of software and hardware systems. It overlaps with other forms of tangible computing (ie. ubiquitous, wearable, invisible) and incorporates both material and computational media, employing mechanical and electronic systems.

Interest in physical computing has risen dramatically over the last fifteen years in the fields of architecture, engineering, industrial design, and art. Designers in the future will be called upon to create spaces that are computationally enhanced. Rather than simply design traditional buildings and then add a computational layer, it is better to conceive and design this integration from the outset. A review of the literature reveals that there are no established methodologies for designing architectural spaces as smart or intelligent spatial systems. As such, it is clear that a new multidisciplinary approach is needed to bring together research in the fields of interaction design (IxD), architectural design, product design, human computer interaction (HCI), embedded systems, and engineering to create a holistic design strategy for more livable and productive spaces. Preparing designers for these challenges demands a range of knowledge, skills, and experience well beyond the traditional domain of architectural education.

This book is intended to teach the basics of electronics (using an Arduino) as well as various digital/physical prototyping techniques to people new to the field. It is not a comprehensive book on electronics (as there are already a number of great resources already dedicated to this topic). Instead, this book focuses on expediting the prototyping process. As architects and designers, we make things and build objects that interact with other objects, people, and systems. We're constantly seeking faster and more inexpensive methods to build prototypes, yet at times we are hindered by practical and time consuming factors that arise in the process of bringing our ideas to life. It is the hope that after reading this book, you will have a better understanding how to create digital or physical prototypes that can interact with the world around them. It's also supposed to be fun! I know electronics can at times seem a little intimidating, so I try to keep the explanations as simple as possible and provide questions on how to expand on the work in your own unique way. If you have further questions, feel free to join the Firefly community and post questions, provide feedback, and share your own knowledge at http://www.fireflyexperiments.com.

- Andrew Payne

# TABLE OF CONTENTS

# TABLE OF CONTENTS

## ACKNOWLEDGEMENTS

Firefly was founded by Andy Payne and Jason Kelly Johnson in 2010. Andy has been primarily responsible for writing the code underlying Firefly's various core components. He has also been involved (along with Jason) with the development of the Firefly Firmata, a critical piece of open source software which enables a direct connection between the Arduino hardware and the Grasshopper environment. As of 2011, Andy is the lead researcher and developer of Firefly's new X control tools. He is currently pursuing this research independently as a part of his Doctoral degree at Harvard GSD in Boston, MA. Jason continues to develop and support the core Firefly tools and the Firefly primer. Jason has been active in helping to establish the connection between Firefly and the Arduino software and hardware environment. He is currently an Assistant Professor of Architecture and Interaction at CCA in San Francisco, California.

Firefly is built upon the **Grasshopper** plug-in for **Rhino**, both developed by Robert McNeel and Associates. We would like to thank **Prof. Panagiotis Michalatos** for his generous support and guidance in the development of the Firefly_X toolset. Robert Lee also made critical contributions to early versions of the original Firefly build file. The Arduino language syntax is based on Wiring by Hernando Barragan. The **Arduino** environment is based on **Processing** by Ben Fry and Casey Reas, and is now supported by an amazing team of software and hardware developers that continue to refine and expand its capabilities. **Pachube** was created by Usman Haque and his amazing team at pachube.com. The **reacTIVision** framework is being developed by the Music Technology Group at the Universitat Pompeu Fabra in Barcelona, Spain. Firefly uses WinAVR, a suite of open source software development tools that includes avr-gcc (compiler) and avrdude (programmer) among other things. Ronnie Parsons and Gil Akos of StudioMode in New York City have also contributed to the development of some amazing example files that were released with version 1.004. We would also like to acknowledge the input of various colleagues, students and workshop participants for their support and questions which help the evolution of Firefly.

# 1 ARDUINO

## 1.1 What Is Arduino?

Arduino is a tool for making computers that can sense and control more of the physical world than your desktop computer. It's an open-source physical computing platform based on a simple microcontroller board, and a development environment for writing software for the board.

Arduino can be used to develop interactive objects, taking inputs from a variety of switches or sensors, and controlling a variety of lights, motors, and other physical outputs. Arduino projects can be stand-alone, or they can be communicate with software running on your computer (e.g. Flash, Processing, MaxMSP.) The boards can be assembled by hand or purchased preassembled; the open-source IDE can be downloaded for free.
- *source http://www.arduino.cc*

## 1.2 Installing The Arduino IDE

The Arduino IDE (Integrated Development Environment) contains a text editor for writing code, a message area, a text console, a toolbar with buttons for common functions, and a series of menus. It connects to the Arduino hardware to upload programs and communicate with them.

Before we can begin working with the Arduino hardware we need to download the IDE and install the appropriate drivers.

- Goto the Arduino Download area - http://arduino.cc/en/Main/Software
- Download Arduino IDE 0022 choosing your specific operating system (Windows, Mac, Linux)
- Extract files to a location in your directory

Now, we have to install the drivers.

- Goto to Arduino Getting Started Page - http://arduino.cc/en/Guide/HomePage
- If you have Windows, follow these step-by-step instructions for installing drivers.
- If you have a Mac, follow these step-by-step instructions for installing the drivers.

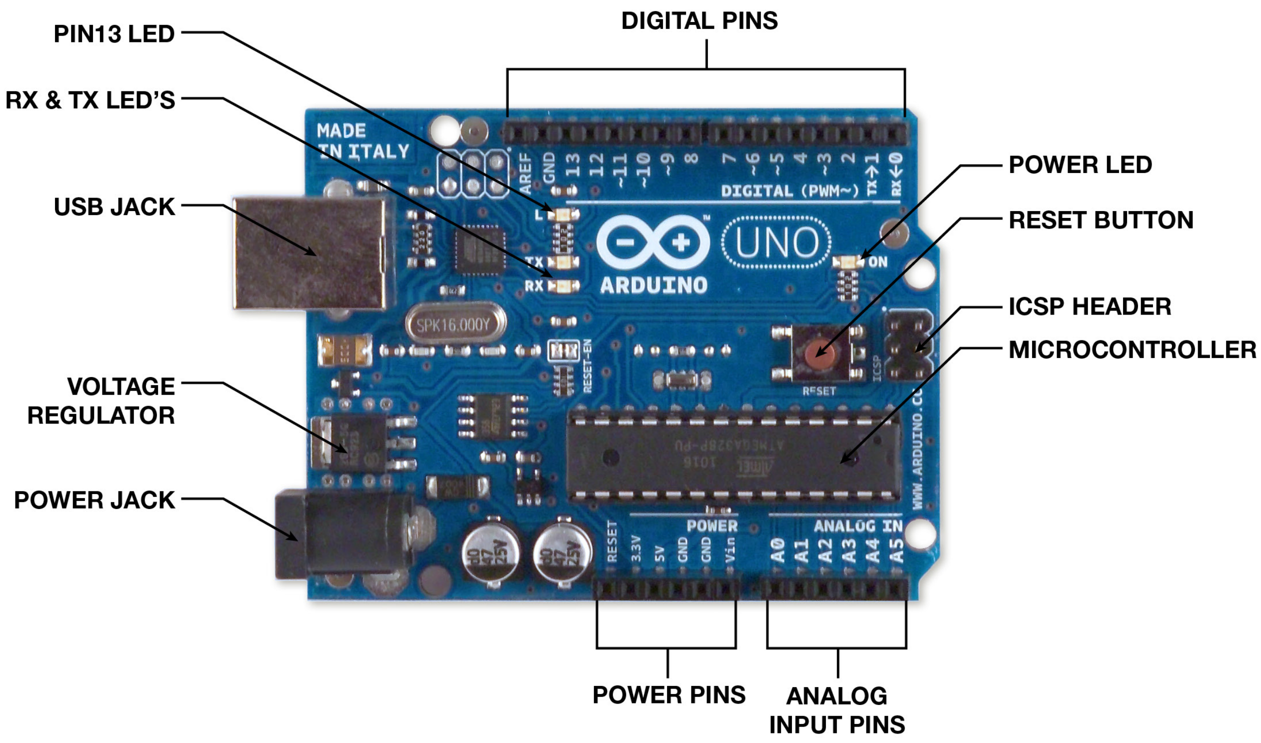
*Note: this step only has to be done one time. Once you have installed the drivers, every Arduino that is connected to your computer will be recognized.*

## 1.3 Check If The Drivers Are Installed Correctly (Blink)

- Open the Arduino folder that you extracted from the zip file you downloaded and click on the Arduino.exe application
- Connect the Arduino microcontroller to your computer using the USB cable
- Plug an LED in the Arduino (the long, positive leg goes into Pin 13 and the shorter, negative leg goes into GND)
- In the IDE, click **File>Examples>Basics>Blink** to open the Blink example.
- Goto **Tools>Board** and choose the Arduino board you are using
- Goto **Tools>Serial Port** and verify the COM port number (we'll use this later)
- Click on the "**Upload to I/O Board**" on the toolbar at the top of the IDE

**Note:** You should see the RX & TX LED's blink as the sketch is uploaded to your board. These LED's indicate each byte of information received by the board. If all goes well, you should see the LED that you plugged into your board blinking at one second intervals.

## 1.4 The Arduino Uno Hardware



The Arduino Uno is a microcontroller board based on the ATmega328 (datasheet). It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button. It contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable or power it with a AC-to-DC adapter or battery to get started.

The Uno differs from all preceding boards in that it does not use the FTDI USB-to-serial driver chip. Instead, it features the Atmega8U2 programmed as a USB-to-serial converter.

"Uno" means one in Italian and is named to mark the upcoming release of Arduino 1.0. The Uno and version 1.0 will be the reference versions of Arduino, moving forward. The Uno is the latest in a series of USB Arduino boards, and the reference model for the Arduino platform; for a comparison with previous versions, see the index of Arduino boards. - *source http://www.arduino.cc*

| | |
|---|---|
| Microcontroller | ATmega328 |
| Operating Voltage | 5V |
| Input Voltage (recommended) | 7-12V |
| Input Voltage (limits) | 6-20V |
| Digital I/O Pins | 14 (of which 6 provide PWM output) |
| Analog Input Pins | 6 |
| DC Current per I/O Pin | 40 mA |
| DC Current for 3.3V Pin | 50 mA |
| Flash Memory | 32 KB (ATmega328) of which 0.5 KB used by bootloader |
| SRAM | 2 KB (ATmega328) |
| EEPROM | 1 KB (ATmega328) |

# 1 ARDUINO

## 1.5 Four Modes of Communication

The communication process for any interactive device follows a simple pattern - information about the environment is gathered from one or more sensors (an electronic component that converts real-world measurements into electrical signals) and then uses that information to interact with the world using actuators (electronic components that convert an electical signal into physical action). Obviously, there is one glaring omission in that pattern... and that is the software needed to synthesize the sensor data and convert it into a signal the actuator can understand. But, we'll get to that in later sections.

Given the flow of information (input from sensors, output to actuators) we can break the communication process down even further using the Arduino microcontroller. Each input or output signal can be broken down into two subcategories: *Analog* and *Digital*.

### Input

- **DigitalRead** - Each sensor value will be represented by a HIGH (1) or a LOW (0). These types of sensors are good for binary conditions (eg. a push buttons, motion detection, etc.).

- **AnalogRead** - Each sensor value will be represented by a 10-bit numeric value (0-1023). The analog to digital converter on the Arduino board will will map input voltages between 0 and 5 volts into integer values between 0 and 1023. Analog sensors are good for gradient changes (eg. light, temperature, humidity, etc.)

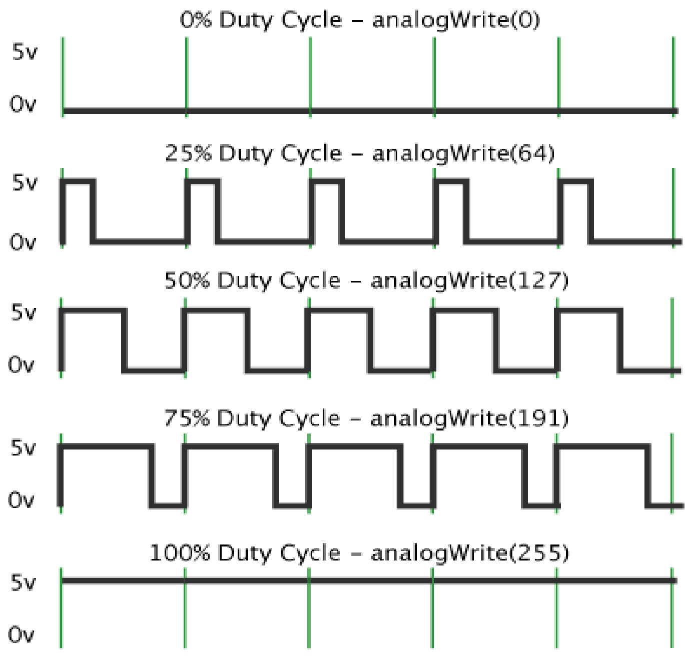
### Output

- **Digital Write** - A digital value (HIGH/LOW) can be sent to any digital pin to turn actuators on or off. If a HIGH value is assigned, then 5V is sent to that pin. If a LOW value is assigned then 0V is sent to that pin. This method is often used to create blink patterns (see first example).

- **Analog Write** - We can send a value to simulate gradient changes in actuation. Because the microcontroller can only send values of HIGH or LOW to each pin, we have to use a subtle trick to get gradient changes in values - **Pulse Width Modulation** (PWM).

  Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the portion of the time the signal spends on versus the time that the signal spends off. The duration of "on time" is called the pulse width. To get varying analog values, you change, or modulate, that pulse width. If you repeat this on-off pattern fast enough with an LED for example, the result is as if the signal is a steady voltage between 0 and 5v controlling the brightness of the LED.

**Pulse Width Modulation**


0% Duty Cycle - analogWrite(0)
25% Duty Cycle - analogWrite(64)
50% Duty Cycle - analogWrite(127)
75% Duty Cycle - analogWrite(191)
100% Duty Cycle - analogWrite(255)
5v
0v

## 1.6 Example: Digital Output - Blink an LED

Now that we know a little more about the Arduino hardware, let's start by taking a look at a simple example of an Arduino Sketch. A 'Sketch' is the term used for the code that is compiled and uploaded to your Arduino board. Once you have successfully upload a sketch, it remains stored in the microcontroller's memory. Whenever you re-connect your board to a power source (USB cable or external power supply) the microcontroller will begin running the sketch that was last loaded onto it. There is no On/Off switch on your Arduino. To stop it, simply unplug it from your power supply.

Every Arduino Sketch requires two functions to exist - one called *setup()* and one called *loop()*.

- ***setup()*** - where you put all of the code that you want to execute once at the beginning of your program.
- ***loop()*** - contains the core of your program which is executed over and over again (hence the name loop).

Let's first take a look at the Blink source code in it's entirety and then we can walk through it step-by-step. Here is the Blink Sketch.

***Source Code: Examples/Basics/Blink.pde***

```
/*
  Blink an LED using Digital Write.
  Turn on an LED on for one second, then off for one second, repeatedly.

  The circuit:
  LED connected from digital pin 13 to ground.
*/

int LEDpin =  13;    // LED connected to digital pin 13

// The setup() function runs once, when the sketch starts

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

// the loop() function runs over and over as long as the Arduino has power

void loop(){
   digitalWrite(LEDpin, HIGH);    // turn the LED on
   delay(1000);                  // wait for one second
   digitalWrite(LEDpin, LOW);     // turn the LED off
   delay(1000);                  // wait for one second
}
```

# 1 ARDUINO

## Breaking it Down

The first thing we come across when looking at the blink sketch is a block of text preceded by a /* and followed by a */. These symbols (/* and */) tell the Arduino that everyting that falls between these characters should be ignored. These lines are comments or notes that you leave in a program so that you can remember what you did when you wrote it. Commenting your code is invaluable as it makes your code more readable to yourself and others.

So, the following code indicates that this sketch will blink an LED on Digital Pin 13 repeatedly at once second intervals.

```
/*
   Blink an LED using Digital Write.
   Turn on an LED on for one second, then off for one second, repeatedly.

   The circuit:
   LED connected from digital pin 13 to ground.
*/
```

The next line of code declares an integer variable called ledPin and assigns it a value of 13.

```
int LEDpin =  13;    // LED connected to digital pin 13
```

A variable is a place in Arduino memory where we can store some data. To declare a variable, you have to tell the program what type of data will be stored in the variable and then give it a name. So, the letters *int* tells the Arduino that the data being stored will be an integer and the name of the variable will be called LEDpin. It's not necessary to always assign the variable a value in the declaration line (eg. `int ledPin;` would also be an appropriate variable declaration), however we have chosen to do this to save space. In this case, we assign the integer 13 to LEDpin. So, anytime we refer to LEDpin in the rest of the sketch, the number 13 will be used in its place (unless we modify it later).

**Note:** Each line of Arduino code (other than comments or include statements) must have a semi-colon (;) to indicate the end of the line.

The next line of code is another way to comment our code - this time using two forward slashes (//). The Arduino recognizes a double forward slash as the beginning marker of a comment line. The difference in this type of comment is that it will only work on a single line comment. If you want to comment out multiple lines of code, it's easier to use the (/* and */) characters.

```
// The setup() function runs once, when the sketch starts
```

The next thing we come across in the code is one of the two required functions in every Arduino sketch - the setup() function. As we mentioned earlier, the setup() function is run only once at the beginning of the sketch. A function is a list of instructions that is run whenever the name of the function is called in the sketch. Thus, we need two things to define a function: a name and a list of instructions.

The first thing we see in the function declaration is the following: `void setup()`. The word 'void' indicates that we're starting the function declaration, but since it's void we wont be returning a value at the end of the function. Next, we name the function 'setup' (this is a predefined word in the Arduino language). An open and closed parentheses follow the function name. If we were creating our own custom function where we wanted to pass a value into the function, we could define those arguements inside these parenthese. In this case, we wont be needing to pass the setup function any variables so we'll leave it empty.

Following the function declaration, we next come across an open curly bracket ({). This tells the Arduino where the list of instructions that will be run when the function is called will begin. Inside the setup curly brackets we see one line that declares the LEDpin as an OUTPUT. This is necessary because digital pins can be used as both INPUT and OUTPUT depending on what you have connected to them. The default value for a pin is as an INPUT, so if we had a digital sensor connected to pin 13, then we wouldn't need to set the pinMode. However, since we'll be using digital pin 13 as an output (to turn the LED on and off) we need to tell the Arduino how to set this pin. We end the setup function by closing it with a curly bracket (}).

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

We next see another comment that provides helpful notes as to what will be happening next.

```
// the loop() function runs over and over as long as the Arduino has power
```

Next, we see the loop declaration. Inside the loop curly brackets we see a list of instructions that tell the Arduino to set the LEDpin to HIGH (5V) and then wait one second (1000ms). Then it sets the LEDpin to LOW (0V) and waits another second (1000ms). Notice, we've run across two preset terms 'digitalWrite' and 'delay'. These are terms that are predefined in the Arduino IDE. The digitalWrite command requires two arguments inside the parentheses: first which pin you want to effect and second whether you want to turn it to HIGH or LOW. In our example, a HIGH will send 5V out to the pin and turn the LED on and a LOW value will turn it off. The command delay requires only one argument: the number of milliseconds you want the computer to wait until proceeding to the next line of code.

```
void loop(){
   digitalWrite(LEDpin, HIGH);    // turn the LED on
   delay(1000);                   // wait for one second
   digitalWrite(LEDpin, LOW);     // turn the LED off
   delay(1000);                   // wait for one second
}
```

The loop() function runs continuously as long as the Arduino is connected to a power supply. To make sure you have entered everything correctly, click on the **verify button** on the toolbar at the top of the IDE. If your code has compiled correctly, we next have to upload the sketch to the board.

## 1.7 Uploading The Sketch

If you haven't already done so, connect your Arduino board to your computer using a USB cable.

- In the IDE, click on Tools>Board and make sure board configuration is set to the same one as the one you are using.

- Click on Tools>Serial Port and verify that you have the correct COM port selected. If you are on a Windows machine and you only have one Arduino connected to your computer, you will likely only see one COM#. If you are on a Mac, then the appropriate board will likely be the first one in the list.

- Click on the 'Upload to I/O Board' button on the toolbar at the top of the IDE.

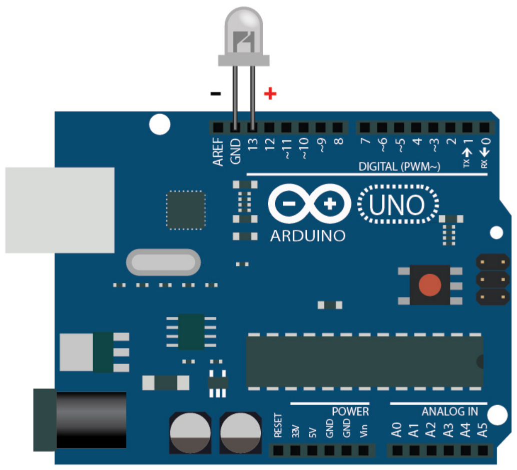
**Note:** You should see the RX & TX LED's blink as the sketch is uploaded to your board. These LED's indicate each byte of information received by the board. If all goes well, you should see the LED that you plugged into your bread-board blinking at one second intervals.

## 1.8 The Blink Circuit

To build the circuit, get an LED and attach its long, positive leg (called the anode) to pin 13. Attach the short negative leg (called the cathode) to ground. It is generally a good idea to use a resistor when connecting an LED in circuit, but we'll cover that in the next tutorial.

### Components

LED (1) any color



### Things to Consider

Try changing the delay values to vary the blink pattern of your LED. Each time you make a change in your code, you will need to re-upload the sketch to your board. Instead of blinking an LED, how might you make an LED fade?

## 1.9 Example: Analog Output - Fade an LED

Hopefully that wasn't too painful. You'll learn more about how to program as we go through other tutorials, but hopefully the main structure of the code is clear. In the last example, we made an LED blink at a set time interval. But how would we fade an LED? Well, in concept it isn't that much different than the blink example. We will still blink turn the LED on and off, only this time we'll oscillate the on/off pattern so fast that our eye will not be able to recognize the pattern. Thus, the amount of brightness (or fade) is determined by the ratio of on to off time intervals. We call this **Pulse Width Modulation** (PWM).

Not all digital pins have PWM capabilities. The Arduino Uno has 6 pins that can use PWM output (pins 3, 5, 6, 9, 10, and 11). The Arduino Mega 2560 has 14 pins that have PWM output. The on-off pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the integer value (0-255) passed to the analogWrite() function. An integer value of zero will mean the entire pulse width will be held down LOW and thus will be completely off. An integer value of 127 (half way between 0 and 255) will represent a 50% modulation wave - meaning half of the pulse width will be held LOW and the other half will be held HIGH. To the human eye, we will recognize the LED as half as bright as if it were fully on. Likewise, if an integer value of 255 is passed to the analogWrite() function, then the entire pulse width will be held HIGH and the LED will be as bright as it can be.

So, let's take a look at how we can use the loop() function to fade and LED back and forth. Here's a look at the complete Fade Sketch (following page).

***Source Code: Examples/Basics/Fade.pde***

```
/*
  Fade an LED using Analog Write

  The circuit:
  LED connected through 220ohm resistor to digital pin 3 and to ground.
*/

int LEDpin = 3;            //connect LED to pin 3
int brightness = 0;       // how bright the LED is
int fadeAmount = 5;    // step value for LED

void setup(){
   pinMode(LEDpin, OUTPUT);   // declare LEDpin to be an output
}

void loop(){
   analogWrite(LEDpin, brightness);   // write the PWM value to LEDpin

   // change the brightness for next time through the loop
   brightness = brightness + fadeAmount;

   // reverse the direction of the fading at the ends of the fade
   if (brightness <= 0 || brightness >= 255) {
      fadeAmount = -fadeAmount;
   }
   delay(30);  // wait for 30 milliseconds to slow down the dimming effect
}
```

## Breaking it Down

The first thing we see (after the block comment) are three integer variable declarations - LEDpin, brightness and fadeamount.  The brightness variable will store the current PWM value that will be assigned to the LED and the fadeamount is the value that the brightness will be increased or decreased each time through the loop() function.

Following that, we notice that we once again have to declare pin 3 as an output using the pinMode command.  This is done once in the setup() function.  So far, our code looks fairly similar to the blink sketch.

The loop() function is where all the magic happens.  The first instruction inside the loop() function is the line:

```
analogWrite(LEDpin, brightness);   // write the PWM value to LEDpin
```

In the blink sketch, we used digitalWrite() to set the pin to either HIGH or LOW.  In this sketch, we use analogWrite to set a specific PWM value to a specific pin.  Here, we tell the Arduino to assign the brightness variable to the LEDpin. During the first pass through the loop, the brightness value will be zero, but we will increase this each time we step through the loop() function.  We do this on the following line by writing:

```
// change the brightness for next time through the loop
brightness = brightness + fadeAmount;
```

If we were to end the loop() function at this point, the brightness value would simply increase by 5 until the memory on the chip eventually ran out and crashed. Remember, a PWM value can only be between 0 and 255. So, we need to set a conditional statement where if we find the brightness value is greater than or equal to 255 then invert the fadeAmount variable so that we will begin decreasing the brightness value each pass through the loop. Of course, this also brings up another condition where we need to test whether or not the brightness value is less than or equal to zero. If this condition is met, then we once again need to invert the fadeAmount variable and begin increasing the brightness again. The conditional statement looks like this:

```
// reverse the direction of the fading at the ends of the fade
if (brightness <= 0 || brightness >= 255) {
  fadeAmount = -fadeAmount;
}
```

The last line of code sets a delay value of 30ms. We need this line to slow down the loop() function. You can control how fast the LED fades in and out by changing the delay time interval (the lower the number, the faster your LED will fade).

```
delay(30);  // wait for 30 milliseconds to slow down the dimming effect
```

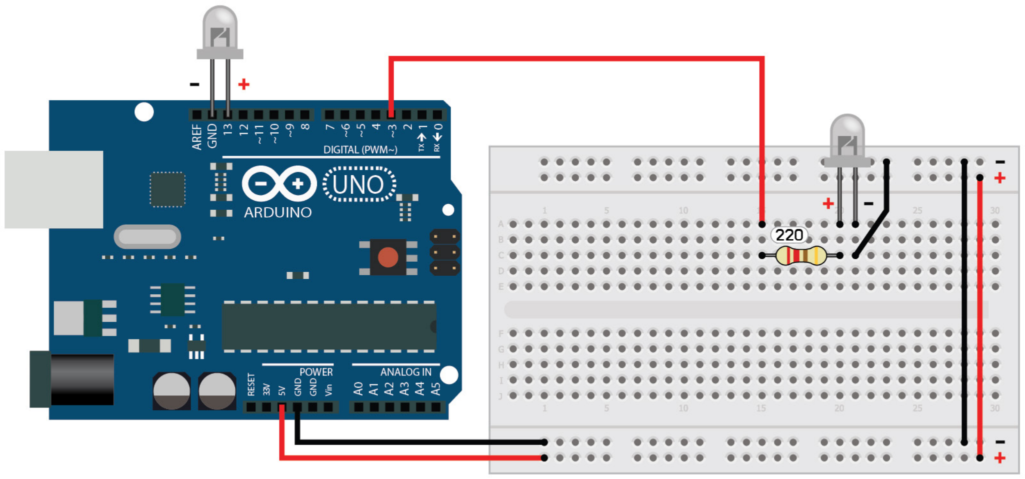
## 1.10 The Fade Circuit

To build the Fade circuit, get another LED and attach its long, positive leg (called the anode) to digital pin 3 through a 220 ohm resistor (see image below). Attach the short negative leg (called the cathode) to ground.

We'll use a breadboard to extend our circuit as we will be adding more components in the coming tutorials. Breadboards are great for making quick prototype circuits and they're quite easy to use. Notice the two rails on the outer edges of the breadboard. When you connect a wire from the Arduino to any hole on one of these rails, the electronic signal is transferred horizontally to every other hole on that rail. Normally, these two rails are reserved for Power (5V) and Ground (GND). This way you can place components anywhere on the inner rails and simply connect small jumpers to the outer rails to get Power and Ground. The inner rails work perpendicular to the outer rails - meaning when you connect a wire to a hole on the inner section of the board, the signal is transferred vertically along that rail. Any component connected anywhere within that vertical rail will also receive that signal. We'll be using breadboards extensively, so we'll get plenty of practice using them. See if you can make your circuit look like the one below.

## Components

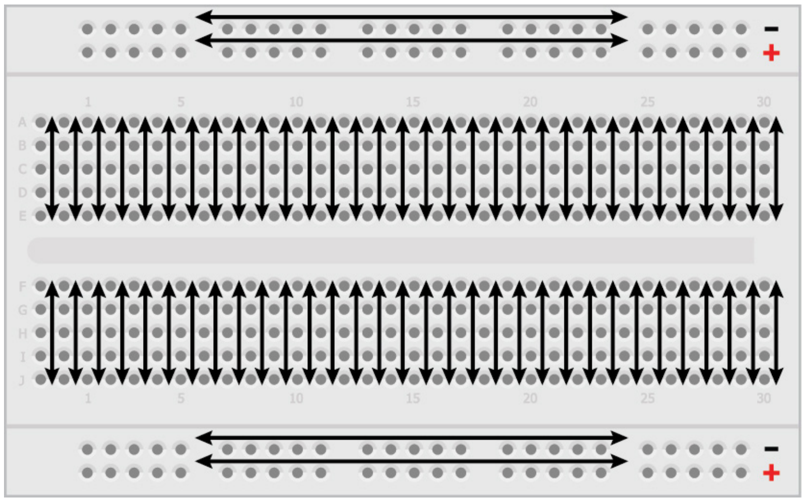
- LED (1)
- LED (1)
- Breadboard (1)
- 220 Ω resistor
- Jumper wires



## 1.11 A Note About Solderless Breadboards

A modern solderless breadboard consists of a perforated block of plastic with numerous tin plated phosphor bronze or nickel silver alloy spring clips under the perforations. The clips are often called tie points or contact points.

A half-sized breadboard usually has two power buses (rails that have two rows of contact points on either side of the breadboard) and 30 columns and 10 rows of tie points in the middle - providing a total of **400 contact points**. All pins are spaced by a standard 0.1" and can accept wire sizes in the range of 29-20AWG. They usually have adhesive on the back as well so you can fasten the breadboard to your project.



The two outer rails (called bus strips) are used to provide power to the electronic components. A bus strip usually contains two rows: one for ground and one for supply voltage (usually marked with a "+" or a red line). Each of these rows are connected electrically, so if you supply a power voltage on one row, then every other contact point on that row is also connected to the supply voltage. The same is true for the ground rail.

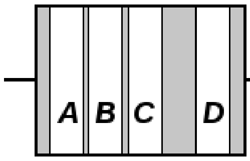
The inner rows and columns are called the terminal strips and are electrically connected in sets of five (vertically in the image above). If you connect a component or wire to a contact point in the terminal strip, then all other points in that column are also connected to one another.

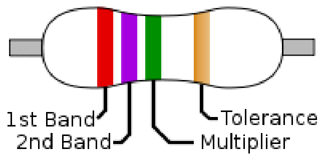
*Note: that the notch running down the middle of the board provides an electrical separation from the two halves of the breadboard. Thus, connecting a wire on one side of the terminal strips does not mean that it will be connected to the other side of the board, unless you provide a direct connection (like a jumper cable).*

## 1.12 How To Read a Resistor

Resistor values are always coded in ohms (symbol Ω), and use a series of color bands to tell you how much resistance the resistor will provide. Most resistors have four color bands (although some have 5 or even 6 bands).

- Band A is first significant figure of component value (left side)
- Band B is the second significant figure
- band C is the decimal multiplier
- band D if present, indicates tolerance of value in percent (no color means 20%)





For example, a resistor with bands of ***yellow***, ***violet***, ***red***, and ***gold*** will indicate that the first digit is a 4 (yellow in table on next page), the second digit is a 7 (violet), followed by 2 (red) zeros: 4,700 ohms. Gold signifies that the tolerance is ±5%, so the real resistance could lie anywhere between 4,465 and 4,935 ohms.

# 1 ARDUINO

## 1.13 Resistor Color Chart

| Color | Digit 1 | Digit 2 | Multiplier | Tolerance |
|---|---|---|---|---|
| Black | 0 | 0 | x$10^0$ | |
| Brown | 1 | 1 | x$10^1$ | ±1% (F) |
| Red | 2 | 2 | x$10^2$ | ±2% (G) |
| Orange | 3 | 3 | x$10^3$ | |
| Yellow | 4 | 4 | x$10^4$ | |
| Green | 5 | 5 | x$10^5$ | ±0.5% (D) |
| Blue | 6 | 6 | x$10^6$ | ±0.25% (C) |
| Violet | 7 | 7 | x$10^7$ | ±0.1% (B) |
| Gray | 8 | 8 | x$10^8$ | ±0.05% (A) |
| White | 9 | 9 | x$10^9$ | |
| Gold | | | x0.1 | ±5% (J) |
| Silver | | | x0.01 | ±10% (K) |

**Note**: A resistor color code reads left to right. Let's take a look at an example. If a resistor displays the colors *yellow*, *violet*, *yellow*, *brown* then the overall value would be 470 kΩ.

The first two bands represent the digits 4, 7. The third band, another yellow, gives the multiplier x$10^4$. The value is then 47 x $10^4$ Ω, or 470 kΩ. The brown band shows a tolerance of ±1%.

## 1.14 Feedback Systems

By now you should be getting the hang of programming in the Arduino IDE and building prototype circuits on a breadboard. We've looked at both analog and digital outputs using LED's (we'll also cover servo control in the next example). But, up til now our electronic devices haven't really been 'interactive'. To be considered an *interactive device,* it needs to be able to gather information about the environment, synthesize that information and create a behavioral pattern which can be used to control a specific actuation (see Figure 1)

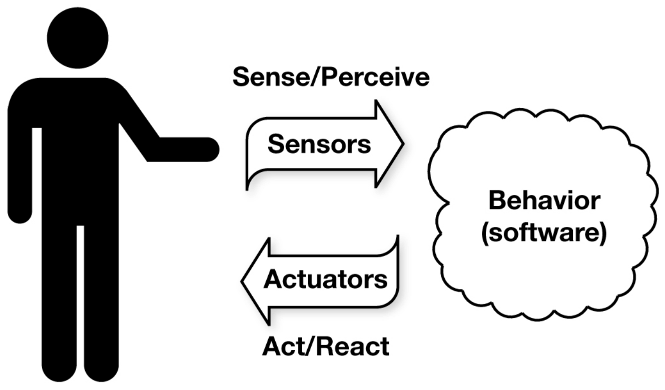


**Figure 1** - The Interactive Device (image by Massimo Banzi)

Sensors help translate physical real-world attributes into values that the computer or microcontroller can use. In general, most sensors fall into one of two categories: 1) **Analog Sensors** and 2) **Digital Sensors**. Analog sensors return a range of voltage in formation that is converted by the microcontroller into a value between 0 and 1023. Digital sensors typically return values in discreet steps - usually a binary condition (ie. HIGH or LOW).

## 1.15 Example: Analog Input - Using the Serial Monitor

Most analog sensors (such as photocells, bend sensors, pressure sensors, or potentiometers to name a few) will be wired into a circuit in such a way that it will have an output voltage that ranges anywhere between 0 volts to 5 volts. The analogRead() command converts the input voltage range, 0 to 5 volts, to a digital value between 0 and 1023. This is done by a circuit inside the Arduino called an **analog-to-digital converter** or **ADC**. Typically, an ADC is an electronic device that converts an input analog voltage (or current) to a digital number proportional to the magnitude of the voltage or current.

In this next example, we'll print the value of our analog sensor (either a potentiometer or photocell) to the Serial Monitor. So let's first take a look at how the analogRead() function works.

***Source Code: Examples/Basics/AnalogReadSerial.pde***

```
/*
  AnalogReadSerial
  Reads an analog input on pin 0, prints the result to the serial monitor
*/

void setup() {
   Serial.begin(115200);   //start the serial connection
}

void loop() {
   int sensorValue = analogRead(0);
   Serial.println(sensorValue);
}
```

## Breaking it Down

In the setup() function, we find a new command. The Serial.begin() function sets the data rate in bits per second (called the baud rate) for serial data transmission and starts the serial communication process. All Arduino boards have at least one serial port (also known as a UART or USART). Serial communication is perhaps one of the best features of the Arduino platform because it's widely supported and  is used for communication between the Arduino board and a computer or other devices. It's incredibly helpful for debugging our programs. For communicating with the computer, it's often preferrable to use one of these rates: 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or **115200**.

**Note:** The Firefly Firmata (which we will discuss in later sections) uses a baud rate of 115200. For this reason, all of the sketches in this document will also use this transmission rate for consistency.

Now that we've started the communication process, let's take a look at the loop() function.

```
int sensorValue = analogRead(0);
Serial.println(sensorValue);
```

In the first line, we create a variable called 'sensorValue' and we assign it the current analog value of analog input pin 0 using the analogRead function. We know that the ADC on the microcontroller will return an integer value between 0 and 1023, so the variable sensorValue will fall within this range.

# 1 ARDUINO

We can use the Serial.println() function to print some message to the serial port. There are two different printing methods - one called Serial.print (print) and another called Serial.println (print line). The print command prints data to the serial port as human-readable ASCII text. The only difference in the print line command is that the line is followed by a carriage return character (ASCII 13, or '\r') and a newline character (ASCII 10, or '\n'). In this instance, we'll use the print line function.

So far, this sketch is pretty simple. But go ahead and upload it to the board so we can start looking at the sensor values using the Serial Montor. Of course, before we can look at the incoming sensor values, we'll need to add another elemet to our circuit.

## 1.16 Analog Input Circuit (potentiometer)

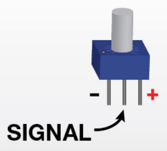
A potentiometer (also known as a 'pot') is a simple analog sensor in the form of a knob that provides a variable resistance, which you can read into the Arduino board as an analog value. By turning the shaft of the potentiometer, you change the amount of resistance on either side of the center pin (or wiper) of the potentiometer. This changes the relative resistances between the center pin and the two outside pins, giving you a different voltage at the analog input. When the shaft is turned all the way in one direction, there is no resistance between the center pin and the pin connected to ground. The voltage at the center pin then is 0 volts, and analogRead() returns 0. When the shaft is turned all the way in the other direction, there is no resistance between the center pin and the pin connected to +5 volts. The voltage at the center pin then is 5 volts, and analogRead() returns 1023. In between, analogRead() returns a number between 0 and 1023 that is proportional to the amount of voltage being applied to the pin.

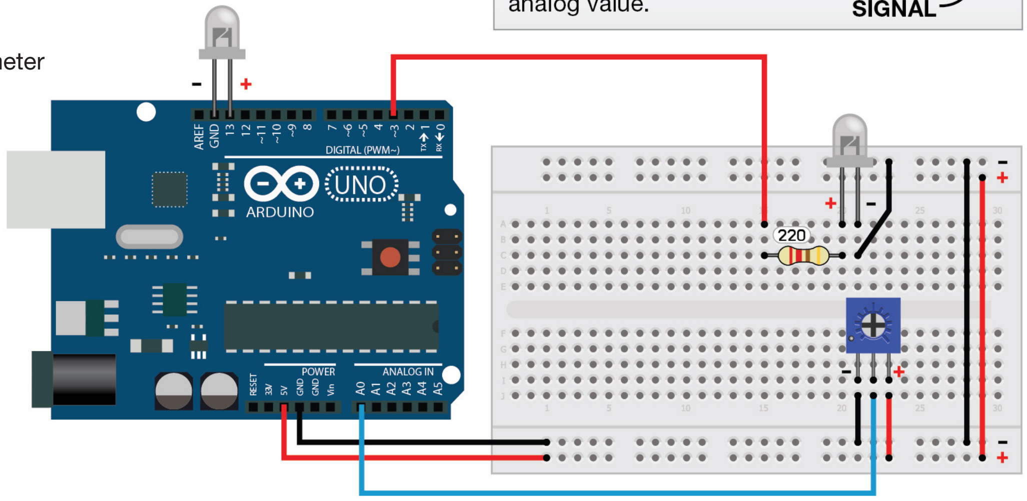
The outer two pins of the potentiometer need to be connected to power (+5V) and ground, although because potentiometers act like other resistors (which are non-polarized), it doesn't matter how you connect these two outer pins. The middle pin on a potentiometer should be connected to one of the analog input pins, in this case pin zero.

### Components

- LED (2)
- Breadboard (1)
- 220 Ω resistor
- Jumper wires
- 10 kΩ Potentiometer

A potentiometer is a simple knob that provides a variable resistance which we can read into the Arduino board as an analog value.





## 1.17 Alternative Analog Input Circuit (photocell)

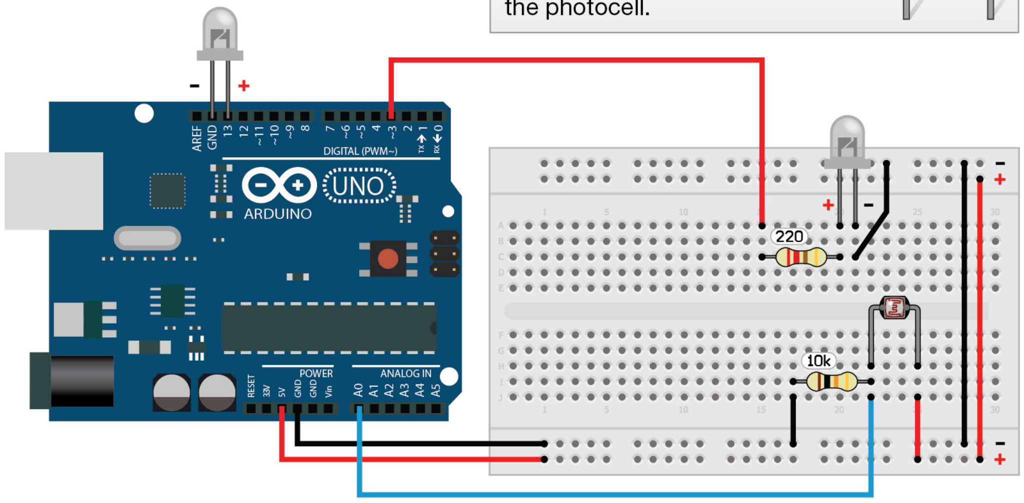
There are many different types of analog sensors (including bend, pressure, and stretch sensors to name a few), many of which have two legs and behave similarly to other two-legged resistors. Photocells are sensors that allow you to detect light. Theys are also sometimes referred to as CdS cells (they are made of Cadmium-Sulfide), light-dependent resistors (LDR), or photoresistors. Each photocell sensor will behave a little differently than any other photocell, even if they are from the same batch. The variations can be as much as 50% or more. For this reason, they shouldn't be used to try to determine precise light levels in lux or millicandela. Instead, you can use them to determine basic light changes.

The easiest way to measure a resistive sensor is to connect one end to power (+5V) and the other to a pull-down resistor to ground. Then the point between the fixed pulldown resistor and the variable photocell resistor is connected to the analog input. The way this circuit works is that as the resistance of the photocell decreases, the total resistance of the photocell and the pulldown resistor decreases from over 600KΩ to around 10KΩ. This means that the current flowing through both resistors *increases* which in turn causes the voltage across the fixed 10KΩ resistor also to *increase*.

A photocell is a sensor that allows you to detect changes in light levels. They act as a resistor that changes its resistance depending on how much light hits the face of the photocell.

## Components

- LED (2)
- Breadboard (1)
- 220 Ω resistor
- Jumper wires
- Photocell
- 10 kΩ resistor

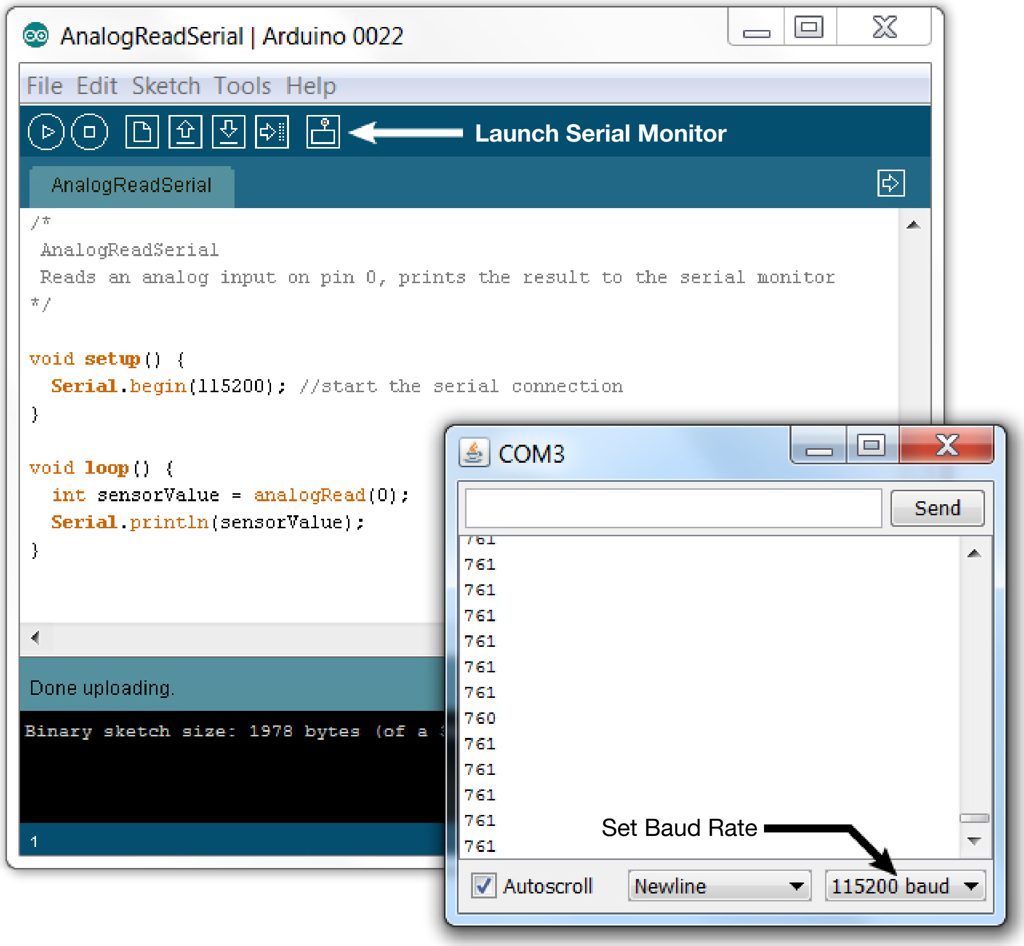


## 1.18 Using the Serial Monitor

Now that we have our circuit setup and the code has been uploaded to the board, the next thing we'll want to do is to launch the Serial Monitor so that we can see the real-time analog sensor values. The Serial Monitor displays serial data being sent from the Arduino board (via USB or serial board). You can also send data to the board by entering text and clicking on the "send" button or press enter. In this sketch we are only printing data to the serial port, so sending data back to the board wont have any effect.

To launch the Serial Monitor, simply click on the icon in the top menu bar (located on far right). The Serial Monitor will communicate with your board at a given baud rate, so this must match the same baud rate set in your sketch. In the bottom-right hand corner, choose the baud rate (115200) from the drop-down menu. (see Figure 2). You should now see the sensor values (somewhere between 0-1023) streaming down the Serial Monitor interface.

**Note:** On Mac or Linux, the Arduino board will reset (rerun your sketch from the beginning) when you connect with the serial monitor.



```
/*
 AnalogReadSerial
 Reads an analog input on pin 0, prints the result to the serial monitor
*/

void setup() {
  Serial.begin(115200); //start the serial connection
}

void loop() {
  int sensorValue = analogRead(0);
  Serial.println(sensorValue);
}
```

```
Binary sketch size: 1978 bytes (of a
```

```
761
761
761
761
761
761
761
760
761
761
761
761
761
```

**Figure 2** - The Serial Monitor can be used to send or receive messages from the Arduino board.

## 1.19 Making Things Move

Hopefully, you were able to see a list of numbers streaming by on the Serial Monitor. These were the analog sensor values coming from your circuit, and now that we have that data we can use it to do anything want... Turn on or fade LED's or other lights, create sounds, or even make things move using a motor. Or more precisely, a Servo motor. Servo motors are great because they are easy to setup, and they have integrated gears and a shaft that can be precisely controlled. Standard servos allow the shaft to be positioned at various angles, usually between 0 and 180 degrees. Continuous rotation servos allow the rotation of the shaft to be set to various speeds (usually through Pulse Width Modulation).

In order to control a standard servo, we're going to need to do a couple of things. First, we'll need to include the Servo library. The Servo library includes a number of useful functions that we can call to control a servo. We'll get into some of these function shortly. We'll also need to calibrate our sensor data so it can be used as position informatino for the servo. But, before we get into any of these things, perhaps it would be easiest to just look at the entire code, and then we'll break it down.

For this next example, we'll be continuing the Analog Input sketch we started in section 1.15.

```
/*
  AnalogReadSerial
  Reads an analog input on pin 0, prints the result to the serial monitor

  Use the Analog sensor value to precisely control a standard servo
*/

#include <Servo.h>  //import the servo library to use functions

Servo MyServo;     //create a variable of type Servo
int ServoPin = 9;    //connect the signal pin of the servo to Digital Pin 9

void setup() {
   Serial.begin(115200);     //start the serial connection
   MyServo.attach(ServoPin);
}

void loop() {
   int sensorValue = analogRead(0);
   Serial.println(sensorValue);

   //remap the raw sensor values into a range for Servo position (0-180)
   sensorValue = map(sensorValue,0,1023,0,180);
   MyServo.write(sensorValue);  //set the servo position to remapped value
}
```

## Breaking it Down

Looking at the line of code (after the block comment), we instantly see a new command:

```
#include <Servo.h>  //import the servo library to use functions
```

The keyword '#include' (note that the # symbol is required) is used to include outside libraries in your sketch. This gives the programmer access to a large group of standard C libraries (groups of pre-made functions), and also libraries written especially for Arduino. The Servo library is bundled in the Arduino package that you downloaded.

*Note: the keyword #include has no semicolon terminator, and the compiler will yield cryptic error messages if you add one.*

In our sketch, the first line tells the program to include the Servo Library when the sketch gets compiled. The Servo library has many functions (including: attach(), write(), read(), etc.) that will help us to precisely control our servo. But before we can use these functions, we have to tell the program to include the library.

```
Servo MyServo;     //create a variable of type Servo
```

The next line down creates a variable of type Servo and it's name is called MyServo. It's a good idea to create a variable for each physical servo you are using. In this example, we're only going to be using one servo (MyServo).

Next, we see another integer variable declaration called ServoPin and it is assigned the value of nine. This will be the digital pin that we'll connect the signal pin of our servo (see the next section).

```
int ServoPin = 9;   //connect the signal pin of the servo to Digital Pin 9
```

In the setup() function, we find another new command. After we instantiate the serial connection, we see the line:

```
void setup() {
  Serial.begin(115200); //start the serial connection
  MyServo.attach(ServoPin);
}
```

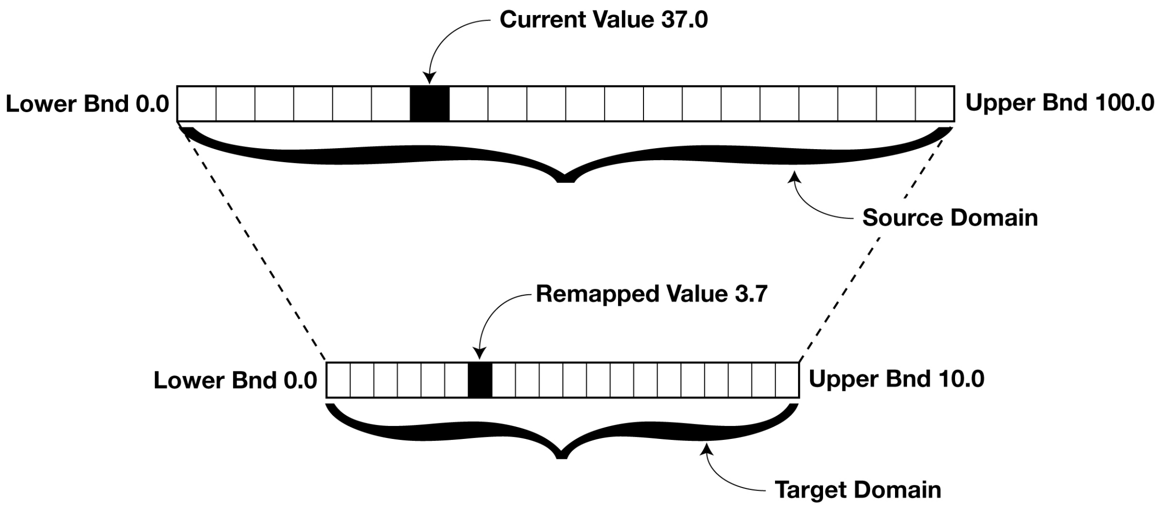
The Servo.attach() function tells the microcontroller that we're planning on connecting a servo to a specific digital pin number (in this case pin 9). We include this line in the setup() function because we need to attach the servo before we can tell the servo to move to a specific position.

Let's now jump into the loop function. In the previous sketch, we had used the analogRead() function to return the current value of our sensor (either potentiometer or photocell). We used the Serial Monitor to look at the current values which will always range from 0-1023. However, the position of our servo can only be set to a value between 0-180, so we'll need to recalibrate our sensor values.

Inside the loop() function we see the line that uses the map function:

```
//remap the raw sensor values into a range for Servo position (0-180)
sensorValue = map(sensorValue,0,1023,0,180);
```

The map function remaps a number from one range to another. That is, a value of from one known value range (source domain) would get mapped to another numeric range (target domain). There are five parameters required for the map function to work: 1) value to be mapped 2) low bound of source range 3) high bound of source range 4) low bound of target range 5) high bound of target range.



Once we've read in the analog sensor value and remapped it to the servo range, we must then tell the servo to move to that position. For this, we'll use the Servo.write() function.

```
MyServo.write(sensorValue);  //set the servo position to remapped value
```

At this point, try compiling the sketch (use the verify button) to see if you have any syntax errors.  If the code compiles correctly, go ahead and upload the sketch to your board.  Next, we'll look at adding a servo motor to our circuit.

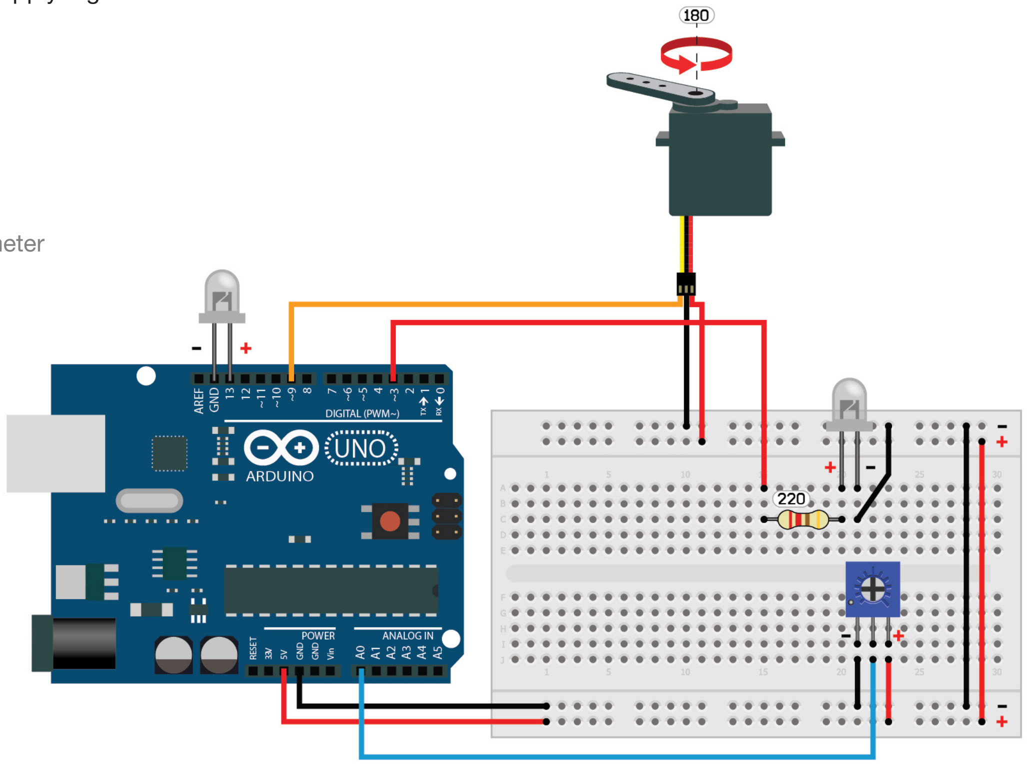
## 1.20 The Servo Circuit

Servo motors have three wires: power, ground, and signal. The power wire is typically red, and should be connected to the 5V pin on the Arduino board. The ground wire is typically black or brown and should be connected to a ground pin on the Arduino board. The signal pin is typically yellow, orange or white and should be connected to pin 9 on the Arduino board.  Let's expand our current circuit setup by adding a servo so that our circuit looks like the diagram below.

**Note:** Servos draw considerable power, so if you need to drive more than one or two, you'll probably need to power them from a separate supply (i.e. not the +5V pin on your Arduino). Be sure to connect the grounds of the Arduino and external power supply together.

## Components

- LED (2)
- Breadboard (1)
- 220 Ω resistor
- Jumper wires
- 10 kΩ Potentiometer
- Standard Servo


180
ARDUINO
UNO
DIGITAL (PWM~)
POWER
ANALOG IN
220
AREF GND 13 12 ~11 ~10 ~9 8 7 ~6 ~5 4 ~3 2 TX 1 RX 0
RESET 3V3 5V GND GND Vin A0 A1 A2 A3 A4 A5

## Things to Consider

Now, that you know how to remap (or calibrate) analog sensor values, try to modify the code so the sensor (either potentiometer or photocell) controls the fade amount of the LED.  Remember, to fade and LED we have to use Pulse Width Modulation (PWM) which uses a specific numeric range (0-255) to control how fast to pulse the HIGH/LOW values.

# 1 ARDUINO

## 1.21 Example: Digital Input - Push Button

In contrast to analog sensors, digital sensors generate what is called a 'Discrete Signal'. This means that there is a range of values that the sensor can output, but the value must increase in steps. There is a known relationship between any value and the values preceding and following it.

Consider, for example, a push button switch. This is one of the simplest forms of a digital sensors. It has two discrete values. It is on, or it is off. A motion detection sensor (also known as a Passive Infrared Sensor or PIR) works on this same principle - is it moving or is it not. The most common digital sensors used with an Arduino provide you with a binary output which has two discrete states (HIGH and LOW).

In the following example, we will turn an LED on or off based on the current state of a push button. Let's take a look at the completed Arduino Sketch.

***Source Code: Examples/Digital/Button.pde***

```
/*
   Turns an LED on and off with a pushbutton

   The circuit:
   LED attached from pin 13 to ground
   pushbutton attached to pin 7 from +5V
   10K resistor attached to pin 7 from ground
*/

int buttonPin = 7;      // the number of the pushbutton pin
int LEDPin =  13;        // the number of the LED pin
int buttonState = 0;   // variable for reading the pushbutton status

void setup(){
  pinMode(LEDPin, OUTPUT);      // initialize the LED pin as output
  pinMode(buttonPin, INPUT);   // initialize the pushbutton pin as input
}

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

  // sets the LED to the button's value
  digitalWrite(LEDPin, buttonState);
}
```

### Breaking it Down

By now, you should be getting the hang of programming in the IDE and there aren't that many new features in this code that we haven't seen before. We start by declaring our variables (buttonPin, ledPin, and buttonState). In the setup() function we have to set the pinMode for both the digital pins connected to the LED and the Button. Notice that the button pin is set to INPUT because we've connected a digital sensor (or input device) to that pin.

Inside the loop() function we encounter this line:

```
buttonState = digitalRead(buttonPin);
```

The digitalRead() command returns the value from a specified digital pin, either HIGH or LOW.

*Note: If the pin isn't connected to anything, digitalRead() can return either HIGH or LOW (and this can change randomly).*

When the pushbutton is open (unpressed) there is no connection between the two legs of the pushbutton, so the pin is connected to ground (through the pull-down resistor) and we read a LOW on digital pin 7. However, when the button is closed (pressed), it makes a connection between its two legs, connecting the pin to 5 volts, so that we read a HIGH value on digital pin 7.

Now that we have read the current digital state of pin 7, all we need to do is to assign that value to our LED pin. We'll do this with the digitalWrite() function.

```
digitalWrite(LEDPin, buttonState);
```

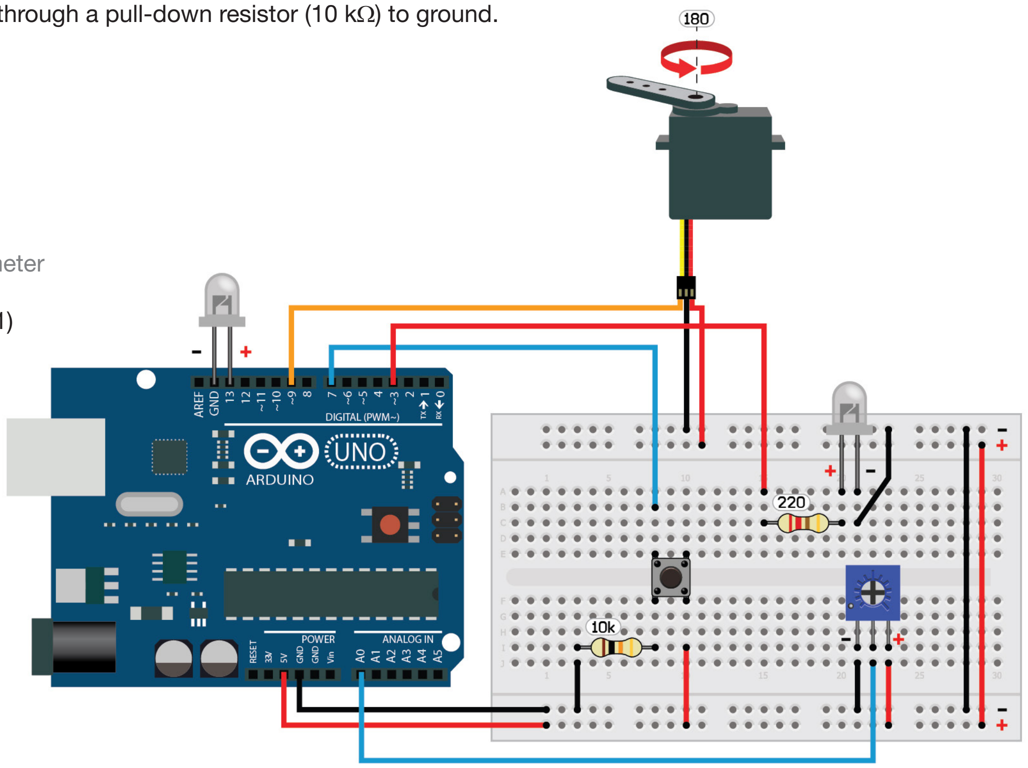
Every cycle through the loop, the microcontroller will go and read the current state of the pin connected to the push button. It will then use that sensor value to turn the LED on or off. So simple! Go ahead and upload this sketch.

## 1.22 The Push Button Circuit

Connect three wires to the Arduino board. The first wire, red, should connect to the power bus (outer rails) and to one side of the push button. Another wire goes from digital pin 7 to other side of the pushbutton. That same side of the button connects through a pull-down resistor (10 kΩ) to ground.

### Components

- LED (2)
- Breadboard (1)
- 220 Ω resistor
- Jumper wires
- 10 kΩ Potentiometer
- Standard Servo
- 10 kΩ Resistor (1)
- Pushbutton (1)



## 1.23 Example: Digital Input - State Detection

Using a pushbutton switch to turn an LED on and off is great, but if you were designing a light fixture then you may have already found a flaw in the sketch we just uploaded. Basically, the light only turns on when we have the button pressed - meaning we would constantly have to hold the switch down to make our LED light up.

Often, you don't need to know the state of a digital input all the time, but you just need to know when the input switched from one state to another. For example, you want to know when a button goes from OFF to ON. This is called state change detection, or edge detection.

This example demonstrates the use of a pushbutton as a switch: each time you press the button, the LED is turned on (if it's off) or off (if on).

The circuit will remain the same as in the previous sketch. Let's take a look at the finished sketch.

***Source Code: Examples/Digital/StateChangeDetection.pde***

```
/*
 State change detection (edge detection)
 The circuit:
    pushbutton attached to pin 7 from +5V
    10K resistor attached to pin 7 from ground
    LED attached from pin 13 to ground
 */

int  buttonPin = 7;              // the pin that the pushbutton is attached to
int LEDPin = 13;                 // the pin that the LED is attached to
int buttonPushCounter = 0;      // counter for the number of button presses
int buttonState = 0;             // current state of the button
int lastButtonState = 0;        // previous state of the button

void setup() {
   pinMode(buttonPin, INPUT);
   pinMode(LEDPin, OUTPUT);
   Serial.begin(115200);
}

void loop() {
   buttonState = digitalRead(buttonPin);

   // compare the buttonState to its previous state
   if (buttonState != lastButtonState) {

      // if the state has changed, increment the counter
      if (buttonState == HIGH) {
         buttonPushCounter++;
         Serial.println("current button state: pressed");
         Serial.print("number of button pushes:  ");
         Serial.println(buttonPushCounter, DEC);
      // Code continues onto next page...
 // Code continued from previous page...
    }else{
      Serial.println("current button state: unpressed");
    }
  }
  // save the current state as the last state for next time through the loop
  lastButtonState = buttonState;
  // turns on the LED every other press by checking the modulo of button counter.
  if (buttonPushCounter % 2 == 0){
    digitalWrite(LEDPin, HIGH);
  }else{
    digitalWrite(LEDPin, LOW);
  }
}
```

## Breaking it Down

The first five variable declarations are exactly the same as we have seen before - all are integer data types and have been given a unique name and a value. The setup() function should also look familiar to you now. We have declared the button pin to be used as an input (we're going to read the state of the button ie. INPUT) and the LED pin will be used as an output. We'll also be printing some information to the serial port for debugging purposes, so we'll need to start the serial connection using a baud rate of 115200.

The first instruction inside the loop() function is to read the current state of the button pin (HIGH/LOW) and assign that value to variable called buttonState. So far so good.

In the next line we see a conditional statement that asks us to compare two values.

```
if (buttonState != lastButtonState) {
```

In this case, we're checking to see whether or not the current button state is ***not equal*** (!=) to the last button state. The (!=) sign is one of the predefined Comparison Operators. There are a number of other operators we could have used (equal to, greather than, less than, etc.) but in this case we want to determine when the current button state has changed from its previous state. If the statement is true, then proceed to the next instruction inside the curly brackets and check if the button has just been pressed (or if the button sttate is HIGH). If so, then we'll increment the button counter by 1. Let's look at the next conditional statement.

```
if (buttonState == HIGH) {
```

To increment the counter by one, we simply need to use the following line:

```
buttonPushCounter++;
```

Adding a double plus sign (or minus sign) is the short-hand notation for incrementing or decrementing a value by 1. It is the equivalent of saying:

```
buttonPushCounter = buttonPushCounter + 1;
```

Next, we find a few print statements. This will be helpful for debugging purposes once we have the sketch running. You may have noticed one additional value inserted into the last print line:

```
Serial.println(“current button state: pressed”);
Serial.print(“number of button pushes:  “);
Serial.println(buttonPushCounter, DEC);
```

The second value inside the parentheses of the last print statement is called an overload value and it tells the serial port how to format the buttonPushCounter string. The DEC value will format the string in Decimal (or base 10).

**Note:** Notice that we have used both Print and PrintLine functions. Remember, the only difference between the two functions is that the PrintLine function adds a carriage return and line feed after the string. We can use the regular Print function before the PrintLine function to concatenate (or add two strings together) before adding the carriage return and line feed.

Following the conditional statement, we save the current state of the button as the last state of the button so we can use it for comparison the next time through the loop.

Finally, we need to whether or not we need to turn the LED on or off based on the last state change. We do this by taking the modulo of the button counter which is incremented by one each time the button is pushed. The modulo() function (%) returns the remainder of the division between two numbers. That is it calculates what is left of the dividend after you subtract from it as many multiples of the divisor as possible. For example 7%2 is one because 2 fits exactly three times in 7 and what remains is 1. Or to put it another way 7/2 = 3.5 and 7-3*2=1.

So the remainder is a very convenient way to find out whether an integer can divide exactly into another integer. Whenever the result of applying the % operator is 0, it means that the dividend is an exact multiple of the divisor. So, let’s take a look at the next line:

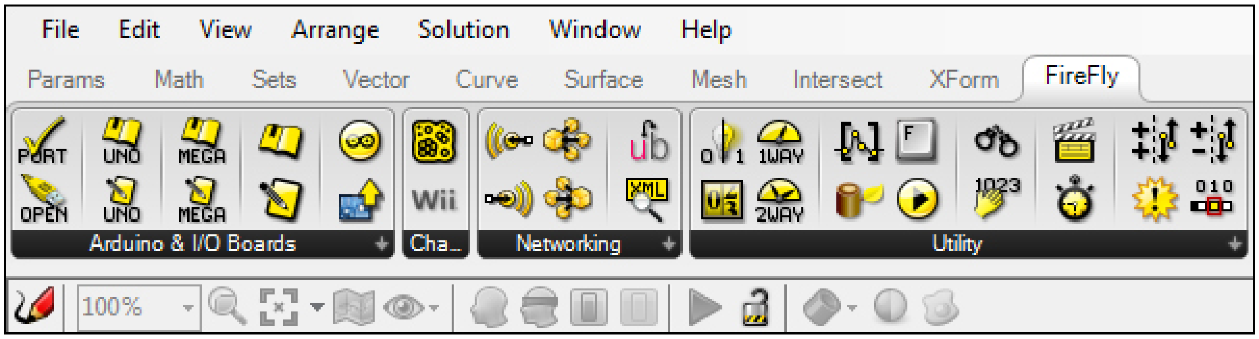
```
if (buttonPushCounter % 2 == 0){
```

This takes the current button counter (which we increment each time we detect a state change to HIGH) and divides it by 2 and returns the remainder. If the button counter is an odd number then the remainder would be one. If the button counter is an even number then the remainder would be 0. In this way we can switch the LED (using digitalWrite) every other time the counter is incremented. Pretty neat. So our final conditional statement looks like this:

```
if (buttonPushCounter % 2 == 0){
       digitalWrite(LEDPin, HIGH);
   }else{
       digitalWrite(LEDPin, LOW);
   }
```

Now, go ahead and upload this sketch to your board. You now should be able to push your button down once and watch your LED turn on (and stay on). Push it again and it turns off. If you wanted it to turn on only every 4th button press, then all you would need to change is the modulo value (ie. buttonPushCounter%4). Now you can start to program different actions based on different usage patterns!

# 2 FIREFLY BASICS



## 2.1 What Is Firefly?

Firefly offers a set of comprehensive software tools dedicated to bridging the gap between Grasshopper (a free plug-in for Rhino) and the Arduino micro-controller.  It allows near real-time data flow between the digital and physical worlds – enabling the possibility to explore virtual and physical prototypes with unprecedented fluidity.

As a generative modeling tool, Grasshopper offers a fluid visual interface for creating sophisticated parametric models, but by default, it lacks the ability to communicate with hardware devices such as programmable microcontrollers or haptic interfaces.  Firefly fills this void. It is an extension to the Grasshopper's parametric interface; combining a specialized set of components with a novel communication protocol (called the Firefly Firmata or Firmware) which together enable real-time  communication between hardware devices and the parametric plug-in for Rhino.

A notable distinction for Firefly is that it is the first visual microcontroller programming environment designed specifically for a 3-D parametric CAD package (such as Rhino). This feature means that real-world data, acquired from various types of sensors or other input devices (video cameras, internet feeds, or mobile phone devices, etc.) can be used to explicitly define parametric relationships within a Grasshopper model. Firefly completes the communication feedback loop by allowing users the ability to send information from Grasshopper back to the microcontroller in order to incite specific actuations (ie. lights, motors, valves, etc). Ultimately, this workflow creates a new way to create interactive prototypes.

By now, you should have a fair understanding of the Arduino IDE and how to build prototype circuits with the Arduino microcontroller and a solerless breadboard.  Next, we'll expand our knowledge by recreating many of the same Arduino processes; only this time we'll use the Firefly toolset.  I think you'll find that it will enable you to build interactive devices and models in a faster and more intuitive way.  I call this new methodology *interactive prototyping* becuase it enables you to quickly test virtual and physical constructs that can communicate with humans, parametric data, and the world around them.  So, let's begin!

## 2.2 Installing The Firefly Toolbar

To install the Firefly toolbar for Grasshopper, we must copy/paste a couple of files into one of the Rhino directories. This installation requires the use of Rhino 4.0 SR8 and Grasshopper 0.8.050 or higher.  If you do not already have these installed on your computer, please follow these links:

- Rhino 4.0 SR8 (evaluation version)
- Grasshopper v. 8.004 (or higher)

Now the we have both of those software packages installed we need to download the Firefly build file which contains all of the installation and example files.

- Go to http://www.fireflyexperiments/download and download the latest **Firefly_Build 1.00xx.zip** file

# 2 FIREFLY BASICS

Launch Rhino and type the word 'Grasshopper' in the command line. This will launch the Grasshopper Editor.

- Click on File>Special Folders> Components Folder
- Delete any prior versions of the Firefly.gha or Firefly_X.gha build files or the libTUIO.dll file that may be in this folder.
- Open the Firefy Installation folder that was just downloaded (in the zip folder) and copy all the contents of the Firefly Build folder (Firefly.gha, Firefly_X.gha and libTUIO.dll) into the Components Folder that was opened from within Rhino.

**Note:** There are three files that need to be copy/pasted into the Components Folder. The first two are the **Firefly.gha** and the **Firefly_X.gha** (Grasshopper Assembly) files. These files contain all of the libraries for all of the Firefly components in the Grasshopper toolbar. The second is the **libTUIO.dll** file which is the reactTIVsion library which is referenced by the other two assembly files. All three of these files have to be copy/pasted into that directory for Firefly to run properly. **This process only has to be done once!**

- Now, restart Rhino and Grasshopper

This time when you launch Grasshopper, you should see a new Firefly tab at the top of the Grasshopper Editor. Congratulations! You've now installed the Firefly toolbar.

**Note:** If you are having problems getting Firefly to load properly in Rhino 5.0 beta, try the following steps:

- Open the Special Components folder (where we just copied the 3 files) and right-click on each file and select "Properties". Under the General tab, check the box to "Unblock" the file. Do this for each of the three files.
- Now, restart Rhino and Grasshopper

## 2.3 Installing Additional Software To Interface With Firefly

There are a couple of other software packages that are required in order to run some of the advanced features of Firefly. Here is a list of the most currently supported applications with some installation instructions.

### WinAVR

WinAVR is a suite of open source software development tools that includes avr-gcc (compiler) and avrdude (programmer) among other things. To install the latest version of WinAVR:

- Go to: http://sourceforge.net/projects/winavr/files/WinAVR/
- Click on the folder called "20100110" and download the file WinAVR-20100110-install.exe
- Run the executable and follow the instructions
- Now, restart your computer

### ReacTIVision

ReacTIVision is an open-source computer vision framework for fiducial marker and finger tracking. It was mainly designed as a toolkit for the rapid development of table-based tangible user interfaces (TUI) and multi-touch interactive surfaces. To install the latest version of reacTIVision:

- Go to: http://reactivision.sourceforge.net/#files
- Download the reacTIVision vision engine for your appropriate platform (reacTIVision-1.4-w32.zip (Win32))
- Unzip the file to a directory on your computer
- Run the reacTIVision.exe file to launch the vision engine

- To download the fiducial markers go to: http://reactivision.sourceforge.net/data/fiducials.pdf

## 2.4 Uploading the Firefly Firmata Sketch

We're going to assume that you've already gone through Chaper 1 and installed the Arduino IDE (the software needed to upload programs to the Arduino board) and have downloaded the appropriate drivers.  But, just in case this is your first time, you should probably take a look at these tutorials (Win, Mac).

In order to begin working with Firefly, we'll first need to program the Arduino to set up all of the pins so that they can communicate back and forth with Grasshopper.  Fortunately, we've already taken care of all the dirty work, so you don't have to worry about a thing.  First, copy/paste the three sub-folders from within the Arduino Code (from the zip file you downloaded) into your Arduino Sketchbook folder which is usually found in one of two places on your hardrive:

- My Documents\Arduino
- Username\Documents\Arduino

**Note:** Your sketchbook folder should now contain three new folders called:

- Firefly_MEGA_Firmata
- Firefly_UNO_Firmata
- Firefly_Wii_Nunchuck

If you're using an Arduino Uno, launch the Arduino 0022 application and open the Firefly Firmata sketch: File > Sketchbook > Firefly_Uno_Firmata.pde

- Click on Tools> Boards and make sure you have assigned the appropriate board.
- Now, click on Tools> Serial Port and make sure you have designated the appropriate COM# (these are automatically assigned when you connect your board to your computer).

**Note:** You'll want to make a note of the Serial Port number.  We will use this again once we get into Grasshopper.

The only thing that is left is to upload the sketch to the board.  Select the "Upload to I/O Board" button (at top).  Wait a few seconds - you should see the RX and TX leds on the board flashing. If the upload is successful, the message "Done uploading." will appear in the status bar.  You will keep the USB connected to the Arduino - this is how Firefly will communicate with the sensors, motors and LED's connected to the board.

Congratulations! Your Arduino board is now ready to "talk" to Grasshopper.

## 2.5 Circuit Prerequisites For The Following Demos

The next set of tutorials will assume that you have completed the circuit shown on page 20 of this document and that you have already uploaded the Firefly_Uno_Firmata.pde sketch (see above).  If you haven't already taken these steps, please do so before continuing this tutorial.

## 2.6 Opening The Serial Port Connection

Once we have uploaded the Firmata onto our boards, open Rhino and launch the Grasshopper editor by typing 'Grasshopper' in the command prompt.  The first thing we need to do to begin communicating with our Arduino board is to open the serial connection.  Click over to the Firefly tab in the Grasshopper editor to see all of the components.

- Drag and drop a ***Ports Available*** component (Firefly>Arduino Boards>Ports Available) onto the Grasshopper canvas.

# 2 FIREFLY BASICS

If you have one or more Arduino boards connected to your computer, the Ports Available component will poll your device manager and return the number to each device you have connected. If you remember when we installed the Firefly Firmata sketch, I asked you to take note of the COM port number because we would be using it once we started Firefly. The integer value returned from the Ports Available component should be the same as the serial port you designated in the Arduino IDE. You can verify the COM port number is correct by clicking on the Tools>Serial Port button on the Arduino IDE.

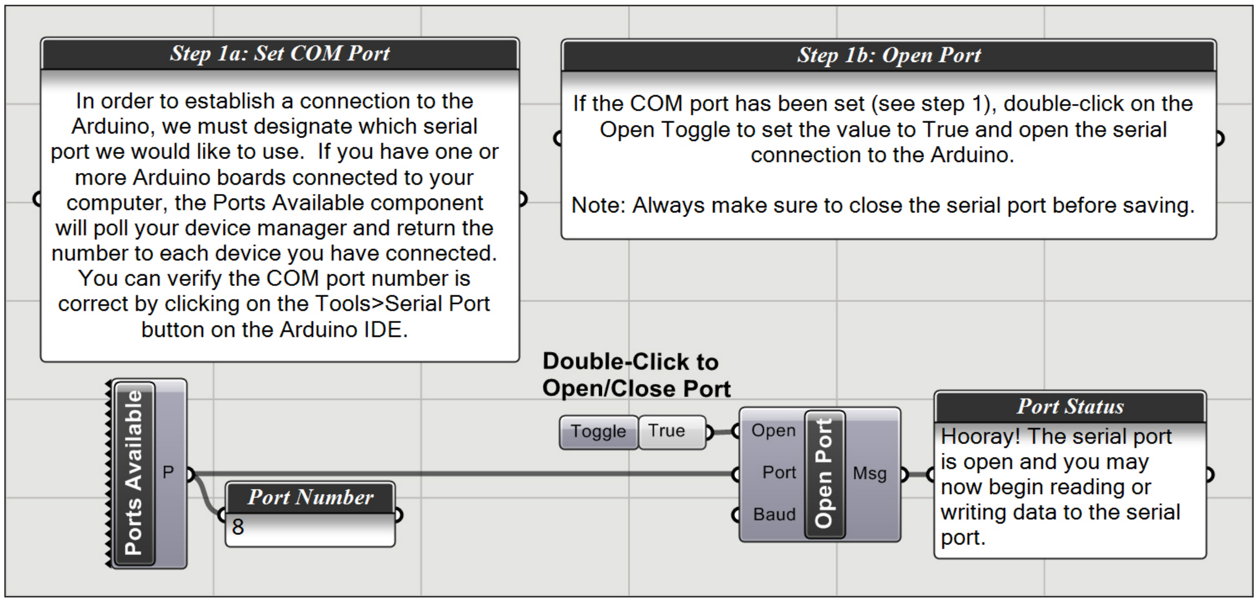
**Note:** The Ports Available component has no inputs. It simply returns a value when you connect your Arduino board to your computer. If your component returns no value, but you have followed all of the steps above, try hitting F5 to recompute the solution.

- Next, drag and drop an ***Open/Close Port*** (Firefly>Arduino Boards>Open/Close Port) component onto the Grasshopper canvas.
- Connect the output of the Ports Available component to the Port input of the Open/Close Port component.
- Drag and drop a ***Boolean Toggle*** (Params>Special>Boolean Toggle) onto the Grasshopper canvas.
- Connect the output of the Boolean Toggle to the Open input of the Open/Close Port component.
- Double click on the Boolean Toggle to set the value to True.

If everything has worked properly, a message will be output from the Open/Close Port component stating:

***Hooray! The serial port is open and you may not begin reading or writing data to the serial port.***

Your Grasshopper definition should look like the image below.


Step 1a: Set COM Port

In order to establish a connection to the Arduino, we must designate which serial port we would like to use. If you have one or more Arduino boards connected to your computer, the Ports Available component will poll your device manager and return the number to each device you have connected. You can verify the COM port number is correct by clicking on the Tools>Serial Port button on the Arduino IDE.

Step 1b: Open Port

If the COM port has been set (see step 1), double-click on the Open Toggle to set the value to True and open the serial connection to the Arduino.

Note: Always make sure to close the serial port before saving.

Double-Click to Open/Close Port

Port Number: 8

Port Status: Hooray! The serial port is open and you may now begin reading or writing data to the serial port.

**Figure 3** - First, we need to open the serial connection.

## 2.7 Sensing

Now that we have opened the serial port, next we need to go and read all of the analog and digital sensor information from the Arduino board. In this case, we'll use the Uno Read component which will read the values from all six of the Analog Input pins and three of the Digital pins (2, 4, & 7).

- Drag and drop a ***Uno Read*** (Firefly>Arduino Boards>Uno Read) component onto the canvas.
- Connect the output of the Ports Available component to the Port input of the Uno Read component.
- Drag and drop a ***Boolean Toggle*** (Params>Special>Boolean Toggle) onto the Grasshopper canvas.
- Connect the output of the Boolean Toggle to the Start input of the Uno Read component.
- Double click on the Boolean Toggle to set the value to True.

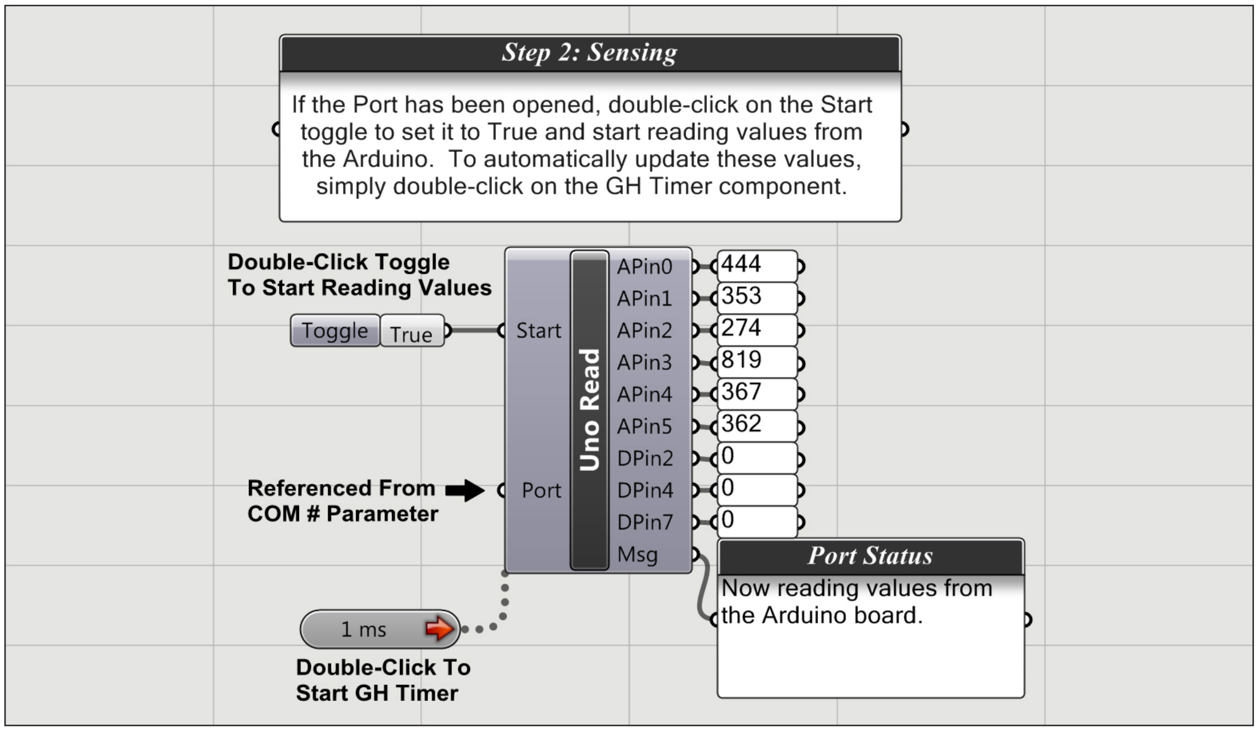
At this point, the Uno Read component has just read the most current value from all of the Analog In and the three digital pins. If you connect a ***Text Panel*** (Params>Special>Panel) you can see the output of each pin. Remember, analog sensor values will return a value between 0 and 1023 and digital sensors will either a 0 (LOW) or a 1 (HIGH).

**Note:** You probably also noticed that there are numbers coming out of all of the pins, even though we only have a potentiometer connected to Analog In pin 0 and a pushbutton connected to Digital Pin 7. This is because there is some voltage static on the board (electronic noise). It causes no harm. Just make sure to only use the output from the pins that you have actually connected your sensors to and disregard the other numbers coming from the other pins that aren't being used.

So far we have only returned a single value for each pin, but they aren't updating in real-time. We need to use a Grasshopper Timer so that we can refresh the values at a given interval.

- Drag and drop a ***GH_Timer*** (Params>Special>Timer) onto the Grasshopper canvas.
- Connect the output of the the Timer input of the Uno Read component.
- Right-click on the Timer and set the Timer interval to 1 millisecond (type 1 in the box)
- Double click on the Timer to start it up!

At this point, you should see the values updating quite quickly. The Uno Read component has been setup to refresh as quickly as possible. Of course, the refresh interval is dependent on how complex your Grasshopper definition has become. Hopefully your Grasshopper definition should look like the image below.



Step 2: Sensing

If the Port has been opened, double-click on the Start toggle to set it to True and start reading values from the Arduino. To automatically update these values, simply double-click on the GH Timer component.

Port Status

Now reading values from the Arduino board.

Double-Click Toggle To Start Reading Values
Toggle
True
Start
Referenced From COM # Parameter
Port
Uno Read
APin0
APin1
APin2
APin3
APin4
APin5
DPin2
DPin4
DPin7
Msg
444
353
274
819
367
362
0
0
0
1 ms
Double-Click To Start GH Timer

## 2.8 Creating A Responsive Cellular System

Now that we have access to our analog sensors (either the potentiometer or photocell) and your digital sensor (pushbutton), we can use that information to create some sort of responsive design. The next example shows how easy it is (using just 3 or 4 new components) to create an adaptive hexagonal aperture system. So let's get started.

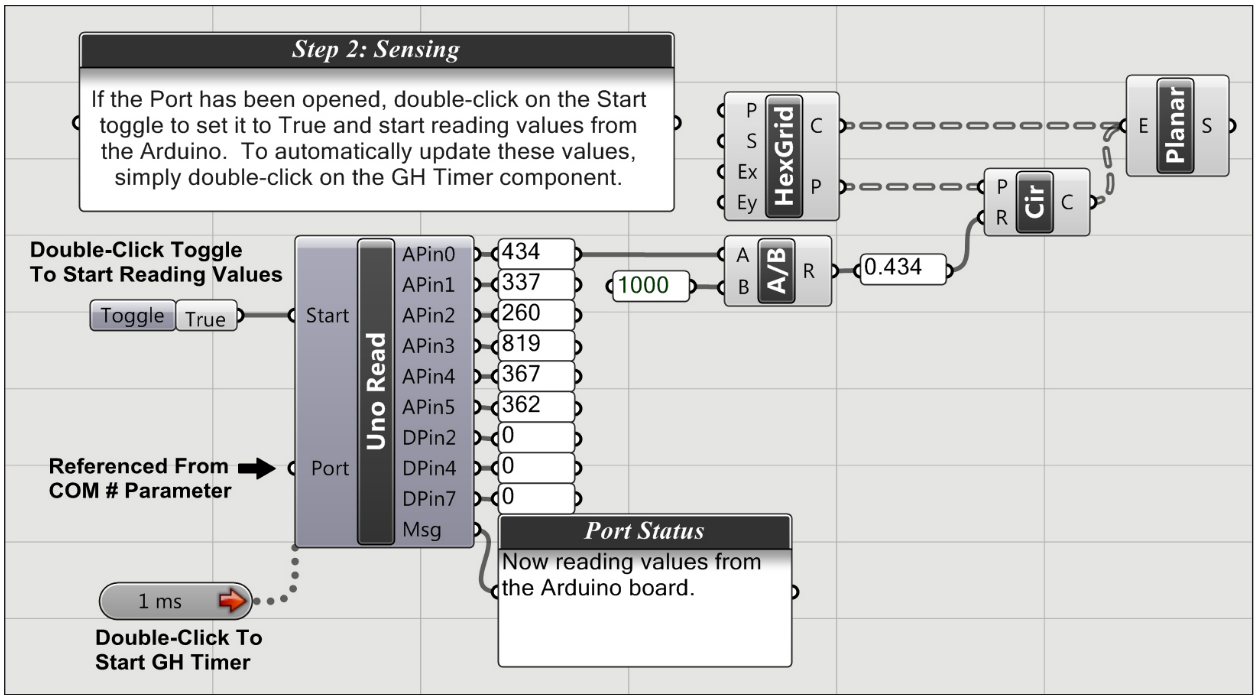
- Drag and drop a ***Hexagonal Grid*** component (Vector>Grids>Hexagonal) onto the canvas.
- Drag and drop a ***Circle*** component (Curve>Primitive>Circle) onto the canvas.
- Connect the P-output of the Hexagonal Grid component to the P-input of the Circle component

You should see a series of hexagonal grid cell outlines with another series of circles placed at the center of each of those polygons. You may have noticed that the circles are a little larger than the outlines. To fix this, we'll need to decrease the size of the circle radii (alternatively we could change the size of the hexagonal cells, although for this definition it's easier just to control the circle size). Instead of connecting a slider to the R-input of the Circle component, we'll use our sensor values so we can control them with a physical input. But, first we'll need to scale down the raw sensor values.

- Drag and drop a ***Division*** component (Math>Operators>Division) onto the canvas.
- Connect the APin0 output of the Uno Read component to the A-input of the Division component.
- Right-click on the B-input of the Division component and Set Integer to 1000.
- Connect the R-output of the Division component to the R-input of the Circle component.

If your GH_Timer is still running, you should now see all of the center circles scaling as you change the value of your analog sensor (photocell or potentiometer). The last step of this tutorial is to create surfaces out of curve geometry in our scene.

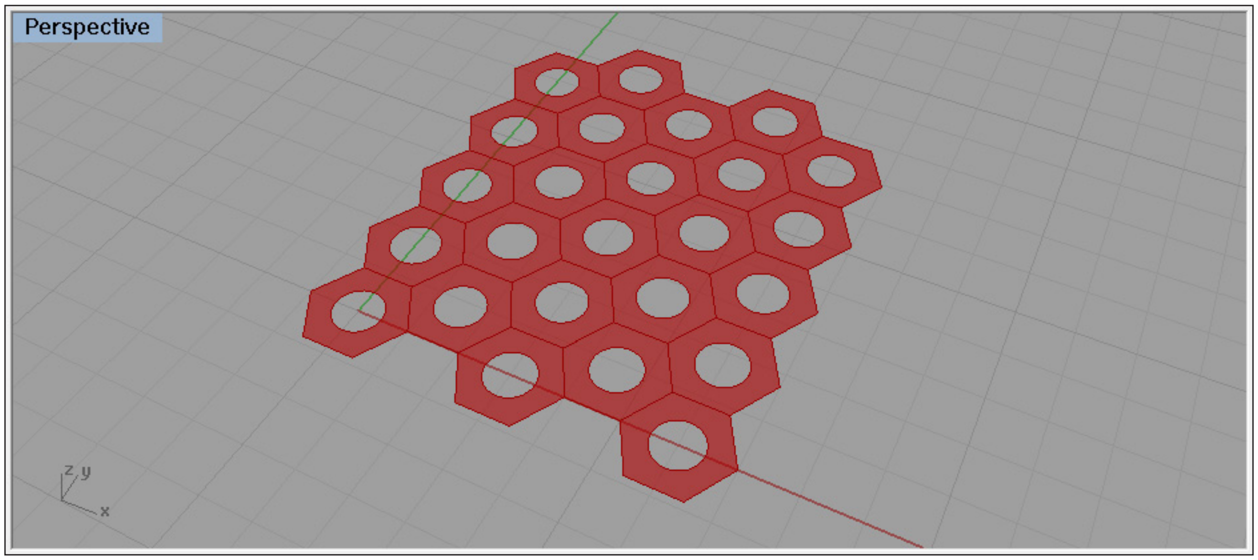
- Drag and drop a ***Planar Surface*** component (Surface>Freeform>Planar Srf) onto the canvas.
- Connect the C-output of the Circle component to the E-input of the Planar Surface component.
- Holding Shift down, connect the C-output of the Hexagonal Grid component to the E-input of the Planar Surface component.


*Step 2: Sensing*

If the Port has been opened, double-click on the Start toggle to set it to True and start reading values from the Arduino. To automatically update these values, simply double-click on the GH Timer component.

*Port Status*

Now reading values from the Arduino board.

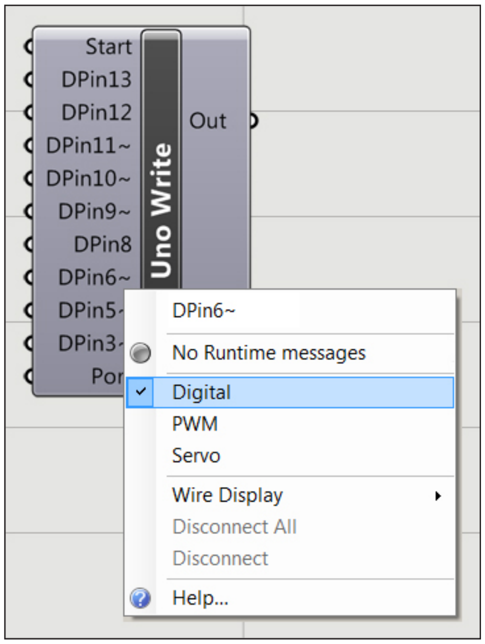


## Things to Consider

At this point, you now access to live input from your analog sensor (potentiometer) and your digital sensor (pushbutton). What kind of responsive models could you build (virtual or physical) with this sort of live input? What other kinds of sensors could you use to create various effects within your parametric model? Try using these inputs to create a dynamic surface or wall. Or how would you track this sensor information over time? You may find some of the other Firefly components (ie. Buffer or Data Log components) helpful in this endeavor. Grasshopper also has a number of data visualization tools which can also be very useful. You may try using some of the tools found under the Special subcategory of the Params tab.

## 2.9 Actuation

The Uno Write component will write values to all corresponding digital pins on an Arduino Uno board. Currently, all of the digital pin inputs for this component have the abliity to write different types of information to the board. By right-clicking on each input, you can set the following (see image to the right):



- **Digital**: With this input, connecting a zero will send a LOW value out to that pin (and thus the pin will output 0V). Alternatively, if you connect an integer value of one to this input, then a HIGH value will be sent to that same pin (and accordingly the pin will output 5V). **Note: all digital pins on the Uno Write component have the ability to send digital information out to the board.**

- **PWM**: You may remember from Chapter 1 that PWM stands for Pulse Width Modulation and as such will use an 8-bit integer value (0-255) to control the digital signal. With this input set, you can attach any numeric value between zero and 255 to control the PWM signal on the pin (the Firefly Firmata uses the analogWrite() function using this input setting). **Note: only pins that have the (~) symbol have PWM capability.** Also, because the Firefly Firmata makes use of the Servo library, PWM functionality is removed from pins 9 and 10.

- **Servo**: Servo motors are great because they are easy to setup, and they have integrated gears and a shaft whose position can be precisely controlled (usually between zero and 180 degrees). The Firefly Firmata uses the Servo library to control this type of motor. This input pin setting will allow an integer value between zero and 180 (which represents the position of the servo motor). **Note: all digital pins on the Uno Write component have the ability to send servo information out to the board.**

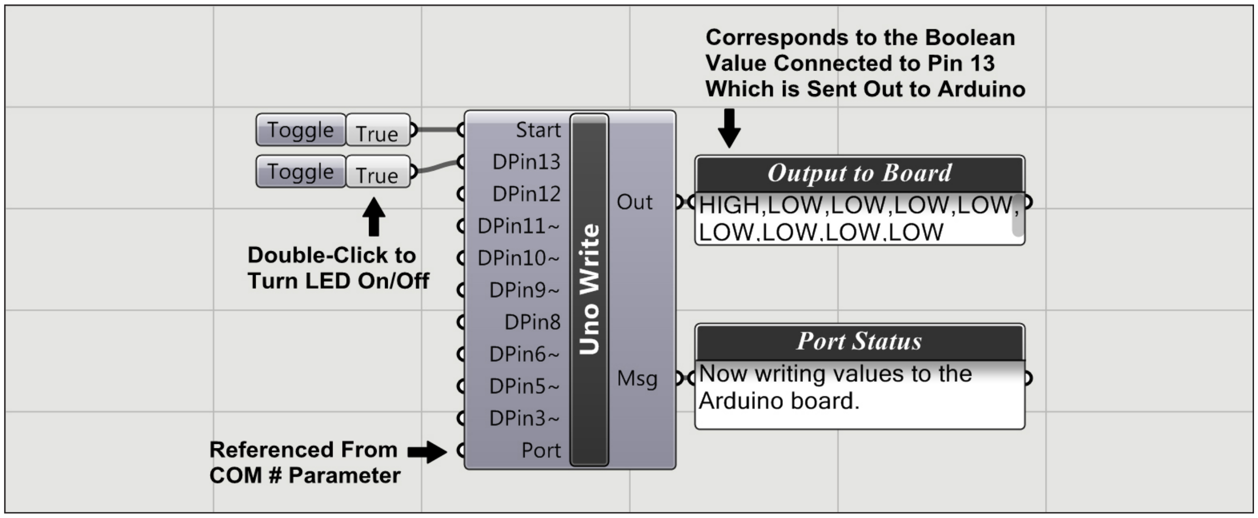
So, let's start by adding some components to our definition to control the actuators we have in our circuit.

- Drag and drop the ***Uno Write*** (Firefly>Arduino Boards>Uno Write) component onto the Grasshopper canvas.
- Connect the output of the Ports Available component to the Port input of the Uno Write component.
- Drag and drop a ***Boolean Toggle*** (Params>Special>Boolean Toggle) onto the Grasshopper canvas.
- Connect the output of the Boolean Toggle to the Start input of the Uno Write component.
- Double click on the Boolean Toggle to set the value to True.

You should now a long comma separated string (list of characters) that is being output from the Out output of the Uno Write component. This is the output string that is sent to the board and parsed up and assigned to the corresponding pin (using the Firefly Firmata). All of the initial values are set to LOW meaning all pins will be turned off to start. The first value in the string corresponds to the first pin input (Digital Pin 13). The next value is associated with Pin 12, and so on and so forth. We'll next try turning on the LED connected to pin 13 in our circuit with a boolean toggle.

- Drag and drop a ***Boolean Toggle*** (Params>Special>Boolean Toggle) onto the Grasshopper canvas.
- Connect the output of the Boolean Toggle to the DPin13 input of the Uno Write component.
- Double click on the Boolean Toggle to set the value to True.

A boolean holds one of two values, True or False. (Each boolean variable occupies one byte of memory.) The input for Digital Pin 13 is expecting an integer value of either a 0 or 1 which will be converted by the Firefly Firmata to a HIGH or LOW value before being assigned to that pin. Fortunately, Grasshopper utilizes many auto-conversions for its inputs... if we assign it a boolean value and it's expecting an integer, it will convert a False value into a 0 and a True value into a 1. Thus, if you double-click this boolean toggle, you should see the output message change from LOW to HIGH (or vice versa) and then watch the LED connected to pin 13 blink on and off.


Corresponds to the Boolean Value Connected to Pin 13 Which is Sent Out to Arduino
Toggle
True
Toggle
True
Double-Click to Turn LED On/Off
Referenced From COM # Parameter
Start
DPin13
DPin12
DPin11~
DPin10~
DPin9~
DPin8
DPin6~
DPin5~
DPin3~
Port
Uno Write
Out
Msg
Output to Board
Port Status

```
HIGH,LOW,LOW,LOW,LOW,
LOW,LOW,LOW,LOW
```

```
Now writing values to the
Arduino board.
```

**Figure 4** - Turn an LED on Digital Pin 13 with a Boolean Toggle

Next, let's try to fade the LED on Digital Pin 3 using the Fader Two Way component.

- Drag and drop a ***Fader Two Way*** (Firefly>Utility>Fader Two Way) component onto the Grasshopper canvas.
- Drag and drop a ***Numeric Slider*** (Params>Special>Numeric Slider) onto the canvas.

- Double click on the handle of the slider and set the following (Rounding: Integers; Upper Limit: 255)
- Copy/Paste this numeric slider
- Connect first slider to V1 input (set the current value to 0)
- Connect second slider to V2 input (set current value to 255)
- Drag and drop another ***Numeric Slider*** (Params>Special>Numeric Slider) onto the canvas
- Double click on the slider and set the following (Rounding: Integers; Upper Limit: 5,000)
- Copy/Paste this numeric slider three times (so that there are 4 total sliders)
- Connect the first slider to T1 input (set current value to 2,000)
- Connect the second slider to T2 input (set current value to 2,000)
- Connect the third slider to D1 input (set current value to 500)
- Connect fourth slider to D2 input (set current value to 500)
- Connect the output of the Fader Two Way component to the DPin3 input of the Uno Write component

- **IMPORTANT: Right-click on the DPin3 input and set the state to PWM**

- Drag and drop ***GH_Timer*** (Params>Special>Timer) onto the canvas.
- Right-click on the Timer and set the interval to 1 millisecond (type 1 in the box).
- Connect the output of the Timer to the Fader Two Way component
- Double-click on the Timer to start it up!

So, let's walk through what we just did. First, we added the Fader Two Way component to the document. The first two values (V1 & V2) stand for the lower limit and upper limit of the fade. Since, we now know that PWM values can only fall within a range of 0 and 255, we set these inputs accordingly. Next, we see the T1 and T2 inputs. These correspond to the time intervals (in milliseconds) for which to fade out and back. So, in this instance it will take two seconds (2000 ms) to fade from 0 to 255 and another two seconds (2000 ms) to fade back to 0. The next two inputs (D1 & D2) correspond to the delay time intervals to wait at the end of each phase of the fade. So, if it takes two seconds to fade from 0 to 255, then it will wait exactly 0.5 seconds (D1) before fading back down to zero. Once it reaches the V1 value, it will wait another 0.5 seconds (D2) before starting all over again. Finally, the Grasshopper Timer is used to automatically refresh the current fade amount.

It is important to point out the note above which is highlighted and placed in bold text. The default value of all digital pins is set to only send digital signals to those pins. However, as previously stated, some pins have the ability to use PWM (the ones with the ~ symbol). In order for us to fade the LED connected to pin 3 in our circuit, we must first right-click on the input and toggle the PWM check box.

Hopefully, if you have setup your definition to look like the image on the next page (Figure 5) then you should see your LED fade at exactly 2 second intervals.

The last thing we want to be able to do is to control a servo motor using Firefly. As we saw in the Servo tutorial, standard servos typically have 180 degrees of rotation. Since we have a servo connected to Digital Pin 9 in our circuit, we can easily control its position by assigning it a specific numeric value between 0 and 180. Of course, in order to control servo motors, we must right-click on that input pin and toggle the Servo checkbox.

- Drag and drop a ***Numeric Slider*** (Params>Special>Numeric Slider) onto the Grasshopper canvas.
- Double-click on this Numeric Slider and set the following (Rounding: Integer; Upper LImit: 180)
- Connect the output of the Numeric Slider to the DPin 9 input of the Uno Write component

- **IMPORTANT: Right-click on the DPin9 input and set the state to Servo**

Hopefully, your definition looks like the next page (Figure 5). You should now be able to blink and LED, fade and LED, and control the position of a servo using a very simple interface.

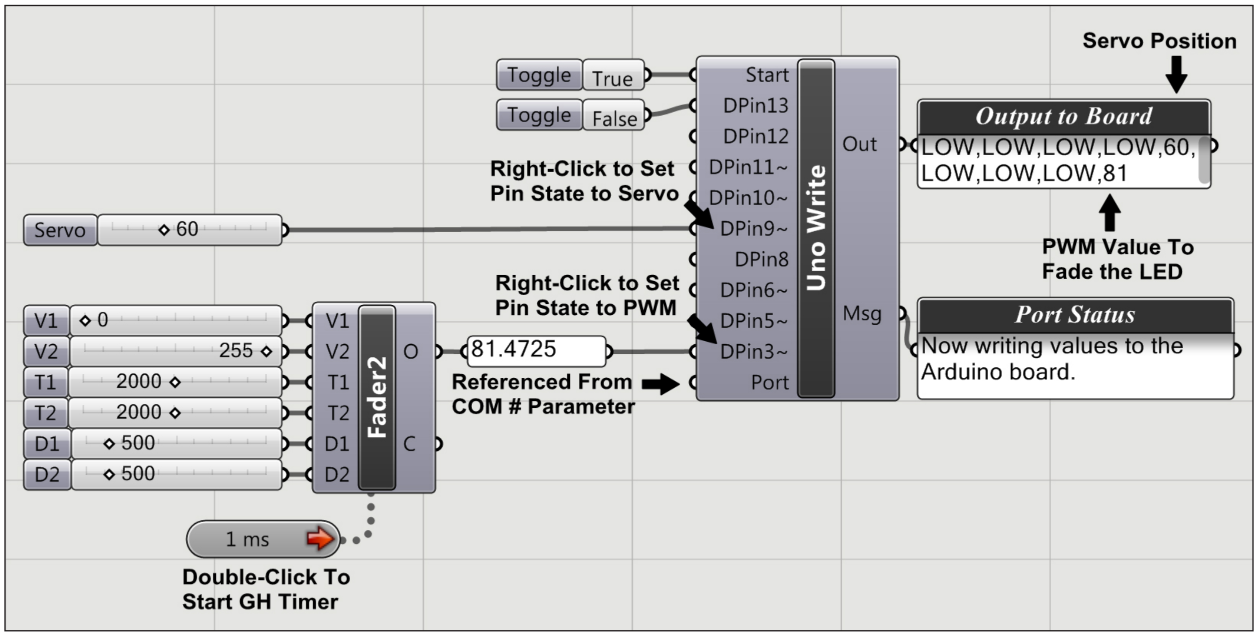


**Figure 5** - Control the position of a servo with a numeric slider and fade an LED using a Fader Two Way component.

## 2.10 Interaction

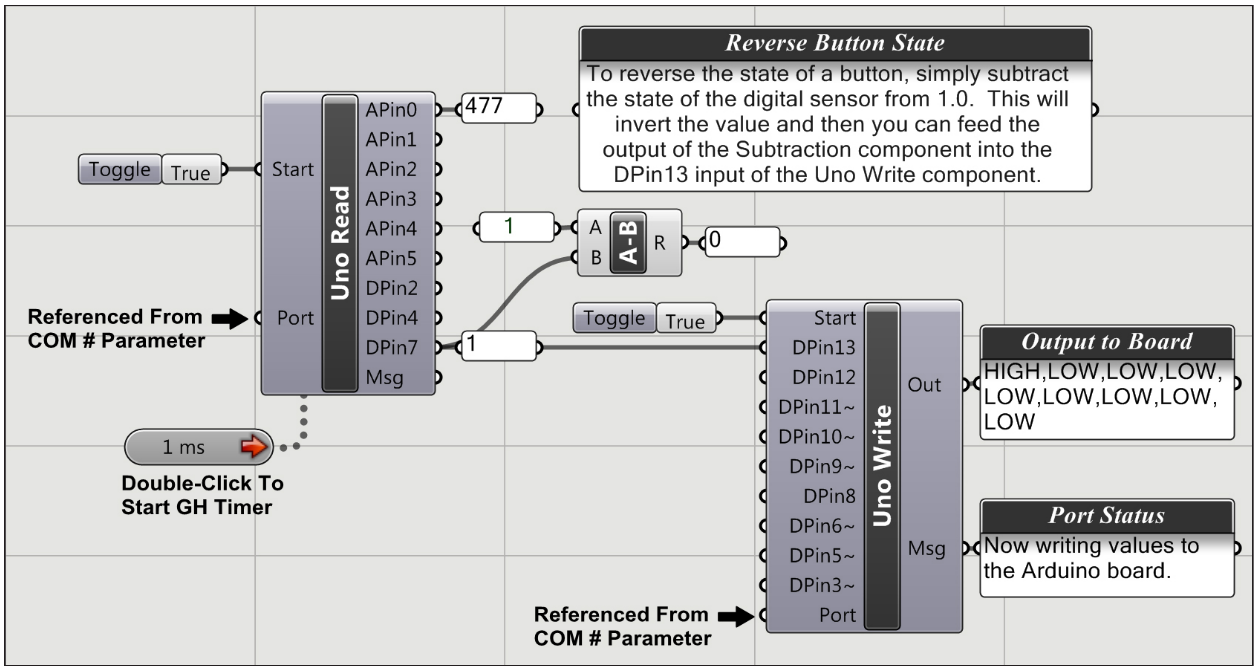
You should now be well versed in reading sensor information into Grasshopper and writing data back out to the board for actuation. You can make virtual prototypes and simulations using real-world parameters. And you can activate change through a series of physical actuations. But, we have the ability to take this approach one step further by combining these two processes (input/output) into one communication workflow (ie. gather sensor information, sythesize it, and trigger an actuation).

This next step will use the potentiometer to control the servo and the pushbutton to turn on/off the LED in our circuit. Let's start with the pushbutton.

The first thing we need to do is delete all of the components that are connected to the Uno Write component (except for the Boolean Toggle connected to the Start input). Select the Fader Two Way component, the numeric slider, and the other boolan toggle and hit delete. We will be replacing these components with sensor information in due course.

- Connect the output of DPin 7 on the Uno Read component to the DPin 13 input of the Uno Write component.
- Make sure the two boolean toggles connected to the Start inputs of Uno Read and the Uno Write components are set to True.
- Make sure the Grasshopper Timer connected to the Uno Read is running (arrow should be red).

That's it. It's as simple as that. If the Timer is running, when you push the button down then the output from pin 7 will turn from zero to one. When you let go, it will go back to zero. If we feed this value to the pin 13 input, then the button state will turn on/off the LED. If you want to reverse the button state, simply subtract the output from pin 7 from 1.0. This will invert the value and then you can connect the output of the Subtraction component into the input of pin 13 (override the existing connection). Hopefully your definition looks like the image facing page.



To reverse the state of a button, simply subtract the state of the digital sensor from 1.0. This will invert the value and then you can feed the output of the Subtraction component into the DPin13 input of the Uno Write component.

HIGH,LOW,LOW,LOW,
LOW,LOW,LOW,LOW,
LOW

Now writing values to the Arduino board.

**Figure 6** - Turn an LED connected to Digital Pin 13 on/off using a pushbutton connect to pin 7.

Next, let's take a look at how we can use the analog sensor in our circuit to control the position of our servo. If you remember from the previous Arduino tutorial, we had to use the map() function (in the Arduino IDE) to remap the incoming sensor value (0-1023) into a range that would be suitable for the servo (0-180). We'll also follow a similar proceedure here (with some added bonus features)!

- Drag and drop a ***Constrain*** (Firefly>Utility>Constrain) component onto the canvas
- Connect the output from APin 0 on the Uno Read component to the V input of the Constrain component
- Drag and drop a ***Text Panel*** (Params>Special>Panel) onto the canvas
- Double-click on the text panel and write the phrase: "0.0 To 1023.0" (no quotations needed)
- Connect the output of the Text Panel to the D input of the Constrain component

The constrain component sets numeric limits (called a domain) to an incoming value. If the value drops below the lower limit of the domain, then the output will simply return the lower limit. Similarly, the upper limit will be returned if the incoming value goes above the upper bound of the domain. In this case, we've constrained our values to fall within a range of 0-1023. You're probably saying to yourself that this is silly. We already know that the limit of any analog sensor can only fall within that range, so there's no need to constrain the value. And you would be correct. However, not all sensors in a circuit return values that fall within the entire range. For example, let's say we were using a light sensor to detect the amount of light in a space. If the room is fairly dim, we may only return a value between 300 to 750 (for example). So, we want to set our domain of the Constrain component to be the upper and lower limit of our sensor range. This way we know exactly where our limits will fall. If the sensor value does drop below (or go above these values) it wont really matter, because we've constrained the values to a set domain. This will be important once we remap the values.

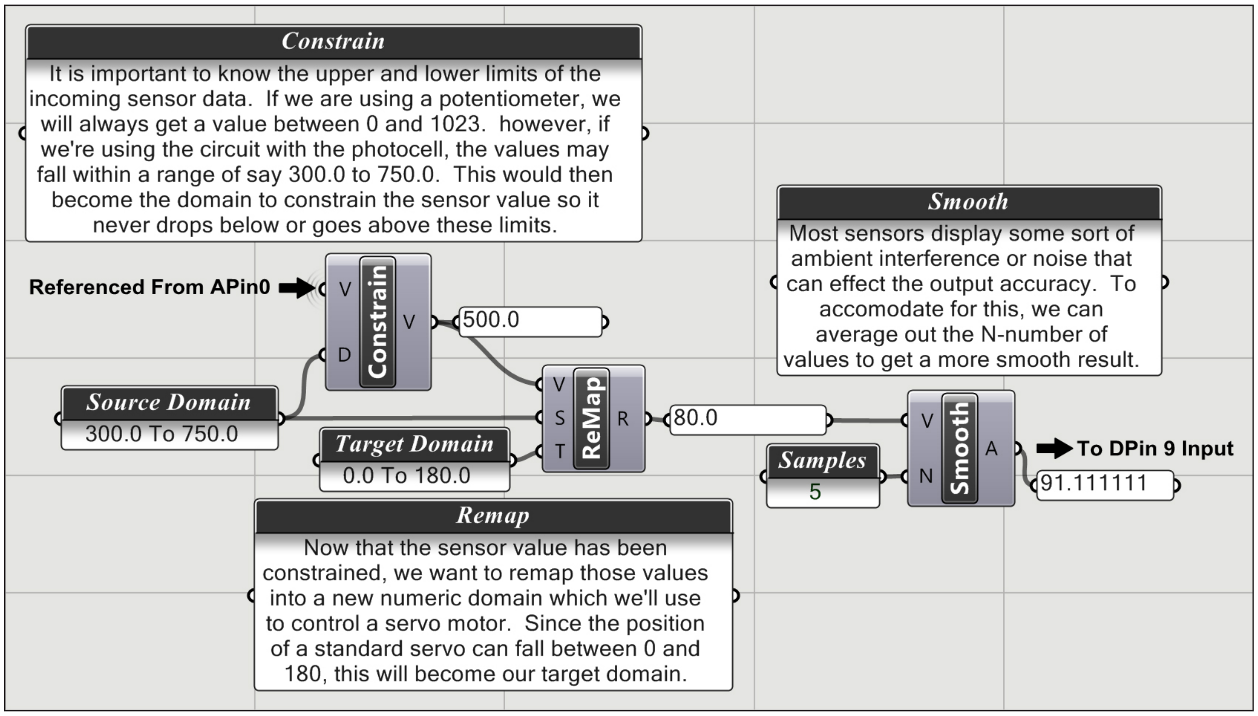
- Drag and drop a ***Remap Numbers*** (Math>Domain>Remap Numbers) component onto the canvas
- Connect the output of the Constrain component to the V input of the Remap component
- Connect the output of the Domain Text Panel to the S input of the Remap component
- Drag and drop another Text Panel (Params>Special>Panel) onto the canvas
- Double-click on this text panel and write the phrase: "0.0 To 180.0" (no quotations needed)
- Connect the output of this Text Panel to the T input of the Remap component

The Remap Numbers component works exactly the same way as the map() function in the Arduino IDE. We essentially need five numeric values: 1) the value to be remapped 2) lower limit of source domain 3) upper limit of source domain 4) lower limit of target domain and 5) upper limit of target domain. The V input corresponds to the value we want to remap (our constrained value). The S input is looking for a domain which is a numeric interval between two numeric extremes. This means it's the range of our sensor value. Since we already defined this in the Constrain component, we'll simply reuse this value in the Remap component. The last input (T or Target Domain) is the numeric range we want to remap our value into. This would be the range of our servo. Since we know our servo can only move 180 degrees, we set the target domain to 0.0-180.0.

At this point, we have remapped our sensor value into a range that is suitable for the servo, so we could go ahead and connect the output of the Remap component into the Pin 9 input. However, most sensor are effected by ambient interference or noise that can change the sensor output or accuracy of the sensor. Meaning, we may see small spikes in our data because of this noise. These small spikes could also cause our servo to move in a jerky or erratic behavior. In order to avoid this, we can smooth out the values by taking the average of all the sensor values in the last N number of samples. To do this, we'll use the Smoothing component from the Firefly toolbar.

- Drag and drop a ***Smooth*** (Firefly>Utility>Smooth) component onto the Grasshopper canvas
- Connect the output from the Remap component to the V input of the Smooth component
- Drag and drop a ***Numeric Slider*** (Params>Special>Numeric Slider) onto the canvas
- Double-click on the slider and set the following (Rounding: Integers; Upper Limit: 20.0)
- Set the current value of the slider to 5
- Connect the output of the slider to the N input of the Smooth component
- Connect the output of the Smooth component to the DPin 9 input of the Uno Write component

As was previously mentioned, the Smooth component will average the values over a certain number of samples (N input). In this case we're taking the last five remapped sensor readings and finding their average before sending that over to the Uno Write component. If you've turned the Timer connected to the Uno Read component, you should see your servo move in a smooth and continous way while you turn the knob on your potentiometer or move your hand over the photocell. Hopefully the calibration portion of your definition looks like the image below.



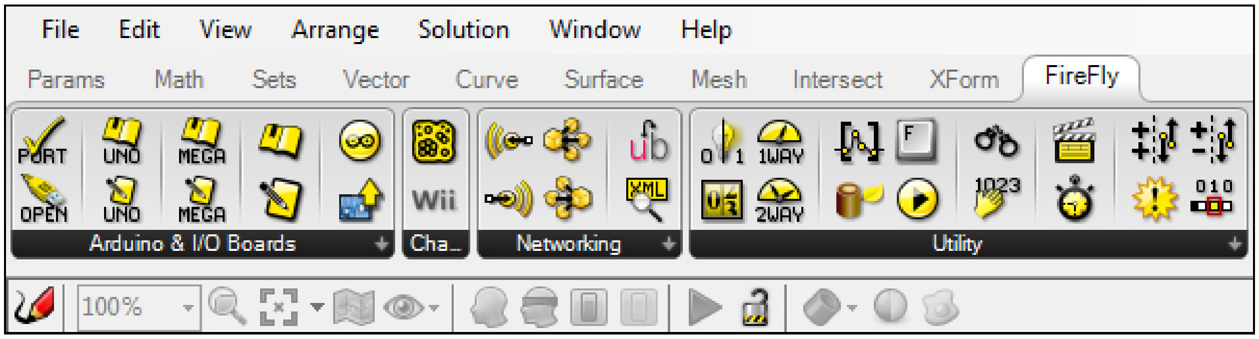
**Constrain**
It is important to know the upper and lower limits of the incoming sensor data. If we are using a potentiometer, we will always get a value between 0 and 1023. however, if we're using the circuit with the photocell, the values may fall within a range of say 300.0 to 750.0. This would then become the domain to constrain the sensor value so it never drops below or goes above these limits.

**Smooth**
Most sensors display some sort of ambient interference or noise that can effect the output accuracy. To accomodate for this, we can average out the N-number of values to get a more smooth result.

**Remap**
Now that the sensor value has been constrained, we want to remap those values into a new numeric domain which we'll use to control a servo motor. Since the position of a standard servo can fall between 0 and 180, this will become our target domain.

**Figure 7** - Calibrating analog sensor values to control a standard servo.

# 3 COMPONENT TOOLBOX



## 3.1 Arduino Related Components

### Ports Available

Check to see which COM Ports are curretly available. If you have one or more Arduino boards connected to your computer, this will return the integer associated with each port. You can verify the COM number is correct by clicking on Tools>Serial Port inside the Arduino IDE (board must be connected to the computer).

### Open/Close Port

Open or close the serial port connection. To be used in tandem with the Serial Read and Write components to establish the connection between Grasshopper and the Arduino.

### Uno Read

Using the FireFly Uno sketch, this component will read values for all corresponding digital and analog pins on the Arduino Uno, Duemilanove, Diecimilla, or Lillypad board. Note: This component is meant to be used in tandem with the FireFly Uno Firmata Sketch.Firmata Sketch.

### Uno Write

Using the FireFly Uno sketch, this component will write values for all corresponding pins on an Arduino Uno, Duemilanove, Diecimilla, or Lillypad board. Note: This component is meant to be used in tandem with the FireFly Uno Firmata Sketch.

### Mega Read

Using the FireFly Mega sketch, this component will read values for the corresponding digital and analog pins on an Arduino Mega board. Note: This component is meant to be used in tandem with the FireFly Mega Firmata Sketch.

### Mega Write

Using the FireFly Mega sketch, this component will write values to all corresponding digital pins on an Arduino Mega board. Note: This component is meant to be used in tandem with the FireFly Mega Firmata Sketch.

### Serial Read (Generic)

Returns any string value coming over the Serial Port. This component does not have any specific Arduino sketch associated with it, which means you can create your own code to send a string value from the Arduino using the Serial.println command.

### Serial Write (Generic)

Write any string value to the Serial Port. You will need to also provide some Arduino code to process the string value coming over the serial port (usually handled by the Firmata sketch).

# 3 COMPONENT TOOLBOX

### Arduino Code Generator

This component will attempt to convert any Grasshopper definition into Arduino compatible code (C++) on the fly. Simply connect the Out string from either the Uno/Mega Write component to the input of the Code Generator and watch the code build as you create your Grasshopper definition. It works by detecting any component 'upstream' from the Uno/Mega Write component. When it detects a component, it checks whether or not it has a function for that component. If it does, it adds the function to the top of the code, and properly calls the function inside the main setup() or loop() functions. If the library of functions does not contain a definition for a particular Grasshopper component, it will provide a warning. This code can be simultaneously saved as a .pde file to be opened in the Arduino IDE.

### Upload to I/O Board

This component will attempt to upload any Arduino sketch to your I/O Board. If the Arduino sketch file path is valid (can be from the Code Generator), you can click on the arrow icon to upload the sketch to the board. Alot of stuff happens behind the scenes, but it should create a dynamic MakeFile and convert the .pde file into a .cpp (C++) file and then into .hex code (machine readable code for the microcontroller). Note: WinAVR is required to be installed in order for the Upload to I/O Board component to work properly. WinAVR (pronounced "whenever") is a free suite of executable, open source software development tools which includes avr-gcc (compiler), avrdude (programmer), among other things. You can download the latest version of WinAVR at http://sourceforge.net/projects/winavr/files/WinAVR/

## 3.2 Channels

### Reactivision

Reactivision Listener draws the position and rotation of each Reactivision fiducial marker. This component is meant to be used in tandem with the Reactivision software, available for free from http://www.reactivision.com/downloads. Fiducial marker set available from http://reactivision.sourceforge.net/#files. Note: You must have the Reactivision software and working camera enabled to use this component.

### Wii Nunchuck

This component will allow you to read all of the sensor values (accelerometers, joystick, C and Z buttons) from the Wii Nunchuck. Note: This component is meant to be used in tandem with the Wii Nunchuck Arduino Sketch.

## 3.3 Networking

### OSC Listener

Open Sound Control (OSC) messages are essentially specially formatted User Datagram Protocols (UDP) transmissions. The OSC Listener component works by opening a UDP port and listening for any OSC message on the network. It receives a message it creates a data tree structure which contains the OSC name and the value (can be floats, integers, strings, etc.). If a new message is received with a different name, then a new branch is created for that message. However, if message is received and a branch already contains that name, then the value will updated. Use a List Item or Tree Item to retrieve the latest value for each message in the data tree. A reset toggle can be used to clear the data tree values.

### OSC Sender

As stated above, OSC messages are specially formatted messages which can be broadcast over a network. In order to send an OSC message, you must know the IP address of the device you're trying to send the message to. You must also specify a unique UDP port. Once you have specified the IP address and the Port, you can connect any type of data (sliders, toggles, panels, etc.) and it will automatically format the data into an OSC message and broadcast that to the specified address.

### UDP Listener

User Datagram Protocol (UDP) is a transmission model for sending/receiving messages over a network. The UDP Listener component simply opens a UDP port and listens for any incoming message.  An IP address is not required for listening for messages (only for sending messages).  The return value is a string.

### UDP Sender

The UDP Sender component sends any string message to a specified IP address on the network through a given port.  This can be useful in sending messages to other programs running on another computer (such as Processing) or other mobile devices.

### Pachube Read

Read a file from any online Pachube feed. Will require an online URL and your pachube API key. Visit http://www.pachube.com for more information.

### XML Search

This component allows you to search through an XML file (either from the web or from a local directory) for a specific element tag, returning the value of that tag in a list.

## 3.4 Utility Components

### Binary Blink

Oscillates 0’s and 1’s based on an incoming pattern of integers. Can be used to create intricate (HIGH/LOW) blinking patterns to be sent to the Serial Write components to control LED’s, etc.

### Counter

A counter which counts upwards and downwards with a specified step size.  This counter also uses a clamping domain so the values will either count up or down to specified limits.

### Fader One Way

Fade from one value to another based on a single time interval (ms).

### Fader Two Way

Fade between a minimum and maximum value based on the fade in and fade out time interval (ms).

### Buffer

Store a set of values in the buffer based on the buffer domain. For example, a buffer domain of 0 To 10 will store the last 10 values. However, if we used a buffer domain of 5 to 15 then the buffer will store the 10 values, but only those that occurred five frames before the current incoming value.

### Data Log

The Data Log component will store a set of values in a list. If the List Length is set to “0” then the log will store values indefinitely (or until you stop the recording or reset the log). Otherwise, the List Length will determine how many values to store in the log. If the Wrap input is set to True, then the values will begin re-recording over the previous log values after it has reached the end of the List Length.

### Is Key Pressed

Test whether or not a specified key has been pressed on the keyboard. Note: This component requires the use of the GH_Timer component.

# 3 COMPONENT TOOLBOX

### Playback

The Playback component will retrieve values from a text file (acceptable file types: .txt, .csv, and .dat) and will begin returning individual values at a given frame rate(s). You can input multiple framerates to simultaneously keep track of multiple outputs at different intervals.

### Constrain

Constrains or clamp a number to a specific numeric range (Domain).

### Smoothing

Smooth (or average) an incoming value based on a sampling level (number of values).

### Frame Rate

Time in milliseconds since the data was updated. Returns the elapsed time and the frames-per-second (fps).

### Stop Watch

Elapsed time in milliseconds since the stop watch was started.

### AND Flip Flop

The AND flip-flop differs from the NOR flip-flop in the sense that the output gate is determined by its present output state as well the states of both of its inputs. Thus, in a AND flip-flop, the output will not change if both S and R are false, but will toggle to its complement if both inputs are set to true.

### NOR Flip Flop

The NOR flip-flop has two inputs, namely, a Set input (S) and a Reset (R) input. A simple representation of an S-R flip-flop is a pair of cross-coupled NOR gates, i.e, the output of one gate is tied to one of the two inputs of the other gate and vice versa. Thus, setting the output gate to false requires R=true and S=false, while setting the output gate to true requires S=true and R=false.

### Bang

Detects when an input boolean has switched states. The equivalent of a 'Bang' component in other programs like MAX/MSP, Pd, and VVVV. Note: This component requires the use of the GH_Timer component.

### State Detection

The State Detection component is used when you want to do some action based on how many times a button is pushed.  It checks the number of times the current value has switched from HIGH to LOW (or vice versa) and whether or not the modulo of that counter value equals zero.  If so, it flips the state detection output. So, if the modulo value is 2, then every other time the button is pushed, the state value will switch. Or if the modulo value is 4, then every 4th push of the button will trigger the state value to change.

# 4 RESOURCES

## 4.1 LITTLE BITS OF EVERYTHING

- Adafruit Based in NYC - Arduinos, Sensors, Wireless, DIY Central, great range of things ...
- Sparkfun Located in Boulder CO - Huge selection of all kinds of electronic parts
- Jameco Located outside SF in Belmont, CA - Huge selection of electronics
- RobotShop Huge selection of robot-based electronics, SMAs, Motors, Sensors
- Al Lashers Electronics Located in Berkeley, CA -1734 University Ave (510) 843-5915
- Electronics Plus San Rafael, CA - (415) 457-0466
- Fry's 1077 East Arques Ave. Sunnyvale, CA, (408) 617-1300 - also in Palo Alto, San Jose
- Marlin P. Jones Huge supplier including power supplies, LED, connectors, etc
- DigiKey Online resource for all kinds of electronic parts (huge selection)
- Octopart Online resource for all kinds of electronic parts (huge selection)
- Mouser Electronics Large selection of electronic parts

## 4.2 ARDUINO & GENERAL ELECTRONICS SUPPLIERS

- Arduino mothership: great links to everything Arduino
- Sparkfun sensors, hardware, kits ...
- MakerShed sensors, hardware, kits ...
- LiquidWare lots of everything

## 4.3 POWER SUPPLIES

- PowerSupplyOne Huge selection of power supplies

## 4.4 SENSORS

- Acroname Sharp IR sensor source - recommended GP2Y0A02YK
- Making Things Great range of sensors - highly recommend
- Sparkfun Sensors (many flavors) Sensor Kit by Sparkfun
- LadyAda Bits and pieces; sensors and Arduinos and much more

## 4.5 SERVOS

- Servocity Great selection of servos and supplies

## 4.6 LEDS

- SuperBrightLEDS.com Single LEDs or multiple strings, all types, good quality

## 4.7 GOOD ARDUINO BASED STARTER KITS

- **Minimum** : Arduino Budget Pack or equal (includes Arduino Uno Atmega328, and a small selection of starter components)
- **Mid-Range** : Arduino Starter Pack or equal (includes Arduino Uno Atmega328, Protoboard, and a good selection of starter components)
- **High-End Recommended** : Arduino Experimentation Kit v1.0 or equal (includes Arduino Uno Atmega328, pro totyping bundles, and a great selection of starter components)
- **High-End Recommended** : Arduino Inventors Kit from Sparkfun or equal (includes Arduino Uno Atmega328, pro  totyping bundles, and a great selection of starter components

# 4 RESOURCES

## 4.8 SHAPE MEMORY ALLOYS

- Dynalloy Maker of Flexinol - Located in Tustin, CA
- Images Scientific Instruments Large selection of nitinol and flexinol based products
- Miga Motors Large selection of SMA driven servos and actuators

## 4.9 MISCELLANEOUS TOOLS

- Other misc. tools that you might consider purchasing: a soldering iron and solder, wire strippers, helping hands, digital multimeter, etc. Here is a great link for info on the best tools to purchase.

## 4.10 SOFTWARE

- Arduino The home for all things Arduino
- Grasshopper3d Plug-in for Rhino; Created by David Rutten and augmented by many others
- Food4Rhino Large selection of Rhino and Grasshopper plugins
- Firefly Experiments The online community for the Firefly toolset
- Processing Arduino is based on Processing by Casey Reas and Ben Fry
- Modkit Drag and Drop programming for Arduino
- Fritzing Draw, Visualize and Output Circuits Diagrams; Arduino to Breadboard
- NetLabToolkit Tools for tangible interaction using Arduino / Flash Widgets

## 4.11 DIY Circuits

- How To Get What You Want Great resource, collaboration between Mika Satomi and Hannah Perner-Wilson
- A Kit of No Parts Great resource for various techniques on DIY circuits
- Instructables Huge selection of tutorials on all things DIY

## NOTES

## NOTES