Arduino Workshop

Overview

Arduino, The open source Microcontroller for easy prototyping and development

What is an 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 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.

Why Arduino?

There are many other microcontrollers and microcontroller platforms available for physical computing. Parallax Basic Stamp, Netmedia's BX-24, Phidgets, MIT's Handyboard, and many others offer similar functionality. All of these tools take the messy details of microcontroller programming and wrap it up in an easy-to-use package. Arduino also simplifies the process of working with microcontrollers, but it offers some advantage for teachers, students, and interested amateurs over other systems:

Inexpensive - Arduino boards are relatively inexpensive compared to other microcontroller platforms. The least expensive version of the Arduino module can be assembled by hand, and even the pre-assembled Arduino modules cost less than $40

Cross-platform - The Arduino software runs on Windows, Macintosh OSX, and Linux operating systems. Most microcontroller systems are limited to Windows.

Simple, clear programming environment - The Arduino programming environment is easy-to-use for beginners, yet flexible enough for advanced users to take advantage of as well. For teachers, it's conveniently based on the Processing programming environment, so students learning to program in that environment will be familiar with the look and feel of Arduino

Open source and extensible software- The Arduino software is published as open source tools, available for extension by experienced programmers. The language can be expanded through C++ libraries, and people wanting to understand the technical details can make the leap from Arduino to the AVR C programming language on which it's based. Similarly, you can add AVR-C code directly into your Arduino programs if you want to.

Open source and extensible hardware - The Arduino is based on Atmel's ATMEGA8 and ATMEGA168 microcontrollers. The plans for the modules are published under a Creative Commons license, so experienced circuit designers can make their own version of the module, extending it and improving it. Even relatively inexperienced users can build the breadboard version of the module in order to understand how it works and save money.

Setting up your Arduino Environment

Get an Arduino based board and usb cable

We will be using Sparkfun's Redboard which is an Arduino Clone based on the Arduino uno, In fact if you come up and see my arduino, youll notice only subtle changes. The original Arduino uno has a removable Atmega Chip which can be swapped out. The redboard has it soldered on and is not removable, Youll also notice the original uno uses a Usb A to B cable much like a printer or copier would use. The redboard uses a Mini usb cable. Everything else is identical.

Why do Arduino clones exist?

They can exist because the Arduino Architechture is Open source, so anyone could make an arduino which is just as functional as an original board. In fact you can build your own arduino for as little as $7.oo however you would need to find a way to interface usb for programming, or already have an existing arduino which can be used as the programmer.

Getting started

You have been provided a Sparkfun Redboard Arduino Clone, This clone identifies itself as an Arduino uno in the Arduino IDE and has an identical layout to the Uno. You also should have the following components in your kit

- 2 Resistors
- 2 Led's
- A breadboard
- 6 breadboard wires
- A Momentary tactile switch

Installing the Arduino IDE

To get started you will need to Download the Arduino IDE, it is available on all major platforms such as Linux, Windows and Mac OSX. The official website for Arduino is http://Arduino.cc. Your machines in this workshop already have the IDE installed, but i would like you to browse the Arduino website.

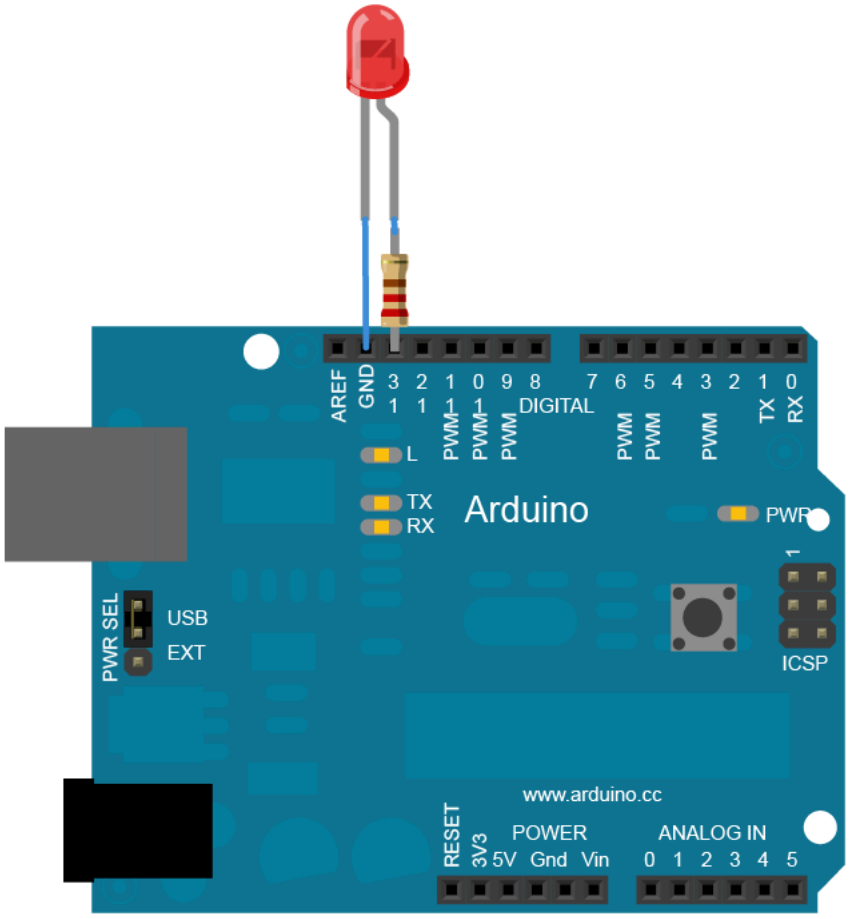
Connecting your Arduino to the computer

Its simple really, plug the usb Mini into your Arduino and the usb A into the computer. You will notice your Arduino board will light up with Led's and will begin running a test program i have loaded on all the Arduinos. This test program simply blinks an led on an off in 100ms intervals.

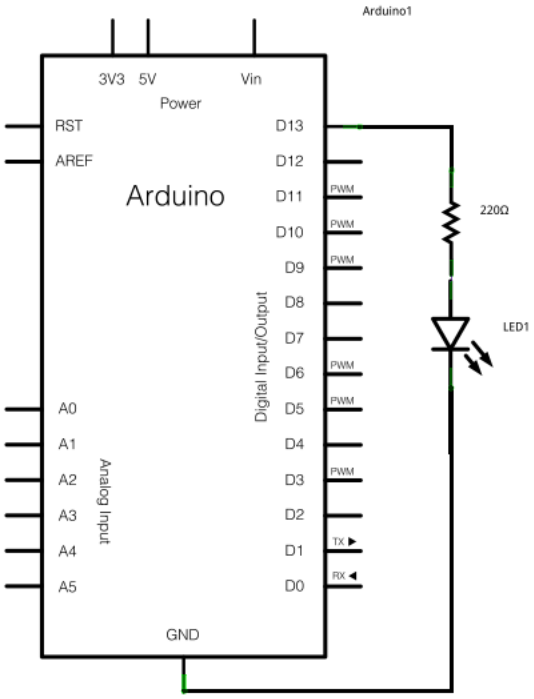
Your first Arduino program

First, I want you to recreate the test program that is running on the arduino now. But i want you to slow the blink interval to 1000ms. The following is the code that is running on your arduino right now.

Wiring it up!



Schematic



The code

The setup routine is what the arduino runs as soon as you give it power or press reset.

`void setup()`

The loop routine runs over and over again as long as power is available to the arduino

`void loop()`

The first thing you do is to initialize pin 13 as an output pin with the line

`pinMode(13, OUTPUT);`

In the main loop, you turn the LED on with the line:

`digitalWrite(13, HIGH);`

This supplies 5 volts to pin 13. That creates a voltage difference across the pins of the LED, and lights it up. Then you turn it off with the line:

`digitalWrite(13, LOW);`

That takes pin 13 back to 0 volts, and turns the LED off. In between the on and the off, you want enough time for a person to see the change, so the delay() commands tell the Arduino to do nothing for 100 milliseconds. When you use the delay() command, nothing else happens for that amount of time.

`

```
//This example code blinks an Led on the Arduino
// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
// initialize the digital pin as an output.
 pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
 delay(100);               // wait for a second
 digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
 delay(100);               // wait for a second
}
```

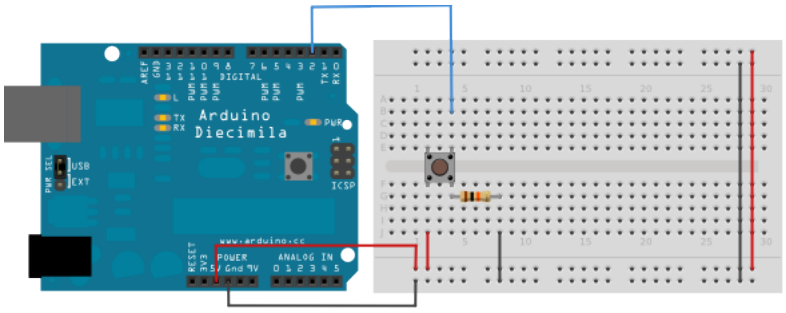
`

Once you have written the code in the IDE, you can send it to your Arduino by pressing on the upload button in the top right of the IDE, it looks like an arrow pointing to the right. You can also verify your code by pressing the check mark next to it. This will look at your code and tell you if there are any errors.

Adding a switch

Once you feel comfortable with the previous code and understand how it is working, you can now learn how to use a switch with an Arduino.

The circuit

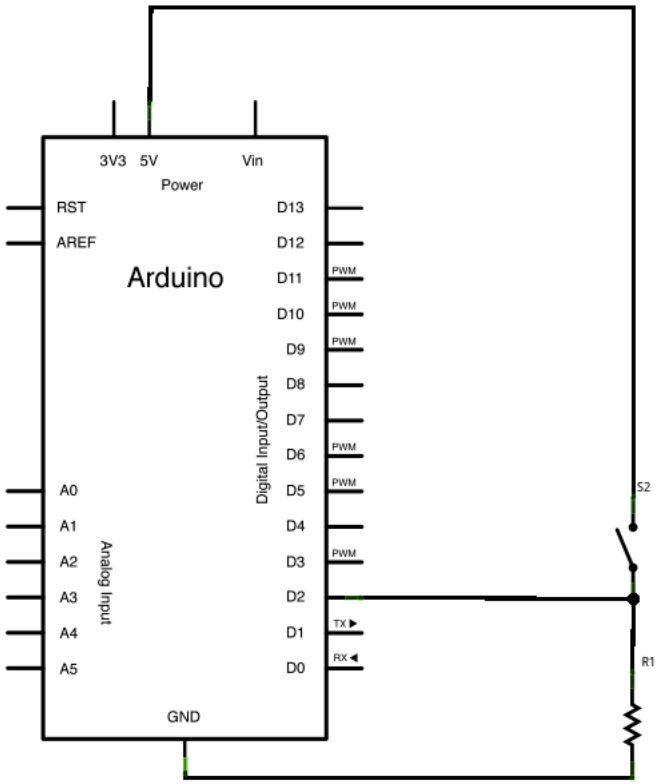


Connect three wires to the Arduino board. The first two, red and black, connect to the two long vertical rows on the side of the breadboard to provide access to the 5 volt supply and ground. The third wire goes from digital pin 2 to one leg of the pushbutton. That same leg of the button connects through a pull-down resistor (here 10 KOhms) to ground. The other leg of the button connects to the 5 volt supply.

Pushbuttons or switches connect two points in a circuit when you press them. 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 reads as LOW, or 0. When the button is closed (pressed), it makes a connection between its two legs, connecting the pin to 5 volts, so that the pin reads as HIGH, or 1.

If you disconnect the digital i/o pin from everything, the LED may blink erratically. This is because the input is "floating" - that is, it doesn't have a solid connection to voltage or ground, and it will randomly return either HIGH or LOW. That's why you need a pull-down resistor in the circuit.

The schematic



The Code

In the program below, the very first thing that you do will in the setup function is to begin serial communications, at 9600 bits of data per second, between your Arduino and your computer with the line:

```
Serial.begin(9600);
```

Next, initialize digital pin 2, the pin that will read the output from your button, as an input:

```
pinMode(2,INPUT);
```

Now that your setup has been completed, move into the main loop of your code. When your button is pressed, 5 volts will freely flow through your circuit, and when it is not pressed, the input pin will be connected to ground through the 10-kilohm resistor. This is a digital input, meaning that the switch can only be in either an on state (seen by your Arduino as a "1", or HIGH) or an off state (seen by your Arduino as a "0", or LOW), with nothing in between.

The first thing you need to do in the main loop of your program is to establish a variable to hold the information coming in from your switch. Since the information coming in from the switch will be either a "1" or a "0", you can use an int datatype. Call this variable sensorValue, and set it to equal whatever is being read on digital pin 2. You can accomplish all this with just one line of code:

```
int sensorValue = digitalRead(2);
```

Once the Arduino has read the input, make it print this information back to the computer as a decimal value. You can do this with the command Serial.println() in our last line of code:

```
Serial.println(sensorValue);
```

Now, when you open your Serial Monitor in the Arduino environment, you will see a stream of "0"s if your switch is open, or "1"s if your switch is closed. `

```
// digital pin 2 has a pushbutton attached to it. Give it a name:
int pushButton = 2;

// the setup routine runs once when you press reset:
void setup() {
// initialize serial communication at 9600 bits per second:
Serial.begin(9600);
// make the pushbutton's pin an input:
pinMode(pushButton, INPUT);
}

// the loop routine runs over and over again forever:
void loop() {
// read the input pin:
int buttonState = digitalRead(pushButton);
// print out the state of the button:
Serial.println(buttonState);
delay(1);        // delay in between reads for stability
}
```

`

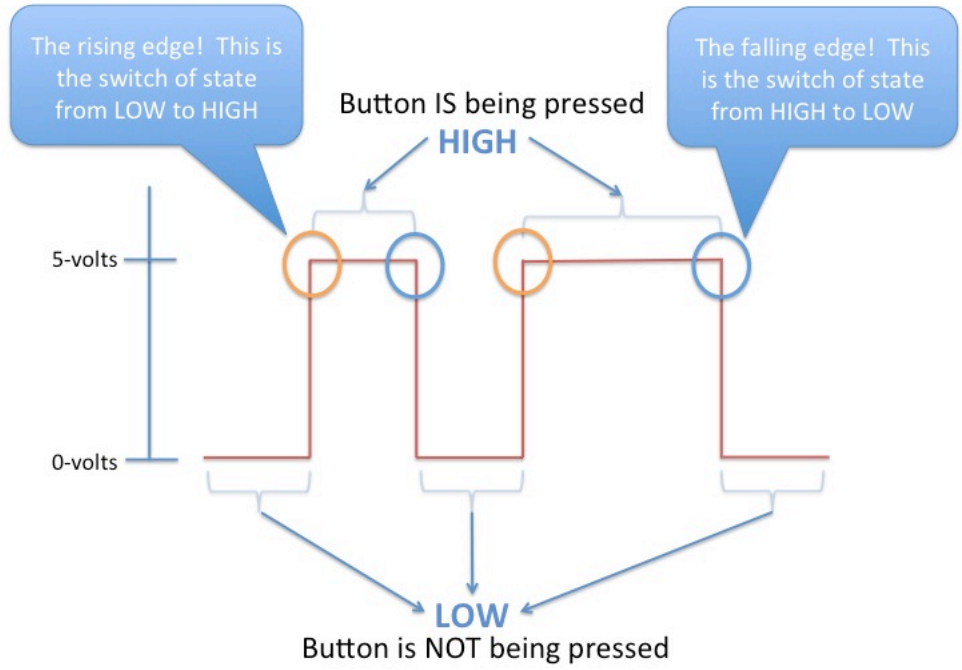
Applying What youve learned so far, Lighting an LED with a switch

In the last example, your learned how to employ a button with your Arduino. When you pressed and held it down – then things could happen, and when you released it, different things could happen. But there lies a blaring problem if you just want an on/off switch – namely who is going to hold the button pressed for you when you get tired?

There are lots of buttons out there – some will hold themselves down – you press it, and it sticks and maintains contact between two wires. So if we just wanted an on/off button, then this is a physical solution to the “Who will hold the button?” dilemma.

The button in the last example was a push button – a type of momentary switch that only connects two wires when you hold it down. What this lesson seeks to explore is how we can use a momentary pushbutton to work as an on/off switch by use of clever code.

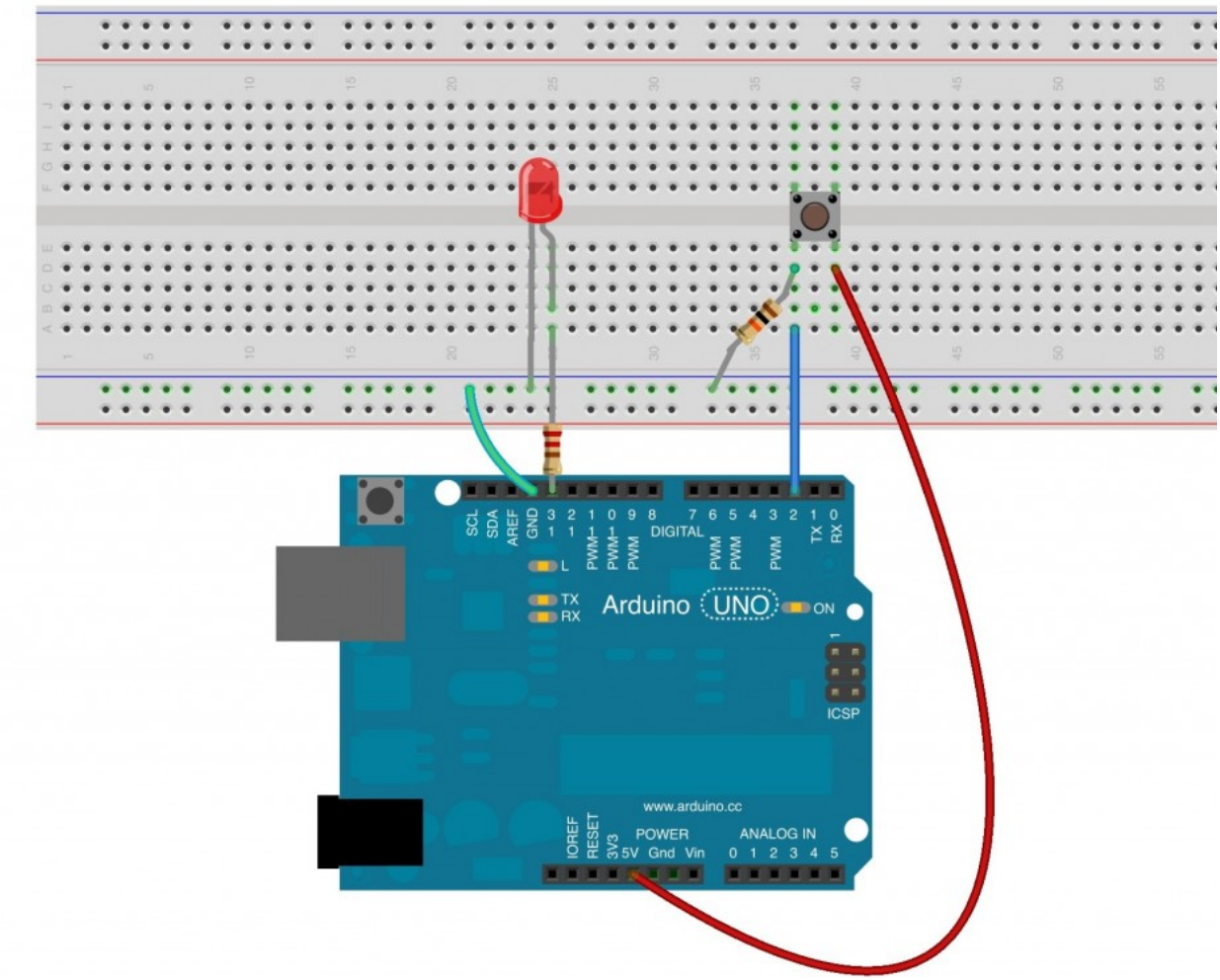
Lets think about happens when you press a button. In this example (and the last) we have a digital pin connected to 5 volts through a pushbutton. When we push the button, the 5-volts is applied to the digital pin. So at one moment there is 0 voltage at the pin, and at the next moment there is 5-volts at the pin. When you release the button – the pin goes back to 0 voltage. Consider the figure below.



What we will do is write a program that says – “when the voltage changes from 0 to 5-volts, then do something, other wise don’t do jack”. This change in voltage is referred to as an edge – and what this program will do is edge detection.

Now an on/off button is useful – but to keep things interesting this program will require four button presses to turn on an LED. Adding this layer of complexity allows us to explore another interesting programming tool called the modulo operator.

The setup



The code

This is the most complicated sketch we have endeavored to understand so far. Remember the best way to keep long programs staring in your mind is to break them up into manageable chunks. First we approach the Variables, then we approach the setup(), and then we consider the loop().

As programs get more involved, you will need to start breaking the loop() into separate chunks – consider parts as separate functional units. You could think – "Ok, this if statement accomplishes this task, or this for loop will increment for this procedure." If you don't mentally Kung-Fu chop the code up – then trying to juggle all the different moving parts in your head will get unwieldy.

The circuit:

- pushbutton attached to pin 2 from +5V
- resistor attached to pin 2 from ground
- LED attached from pin 13 to ground (or use the built-in LED on most Arduino boards)

`

```
// this constant won't change:

const int  buttonPin = 2;    // the pin that the pushbutton is attached to

const int ledPin = 13;       // the pin that the LED is attached to
```

```
// Variables will change:

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() {

// initialize the button pin as a input:

pinMode(buttonPin, INPUT);

// initialize the LED as an output:

pinMode(ledPin, OUTPUT);

// initialize serial communication:

Serial.begin(9600);

}

void loop() {

 // read the pushbutton input pin:

 buttonState = digitalRead(buttonPin);

 // compare the buttonState to its previous state

 if (buttonState != lastButtonState) {

// if the state has changed, increment the counter

if (buttonState == HIGH) {

  // if the current state is HIGH then the button

  // wend from off to on:

  buttonPushCounter++;

  Serial.println("on");

  Serial.print("number of button pushes:  ");

  Serial.println(buttonPushCounter);

}
else {

  // if the current state is LOW then the button

  // send from on to off:

  Serial.println("off");

}

}

 // save the current state as the last state,

 //for next time through the loop

lastButtonState = buttonState;

// turns on the LED every four button pushes by

// checking the modulo of the button push counter.

// the modulo function gives you the remainder of
```

```
// the division of two numbers:

if (buttonPushCounter % 4 == 0) {

digitalWrite(ledPin, HIGH);

 }
 else {

  digitalWrite(ledPin, LOW);

 }

}
```

`

Let's consider the variables declared and initialized. The variables used as pins are qualified as constants and all the others are used to track the state of the button.

`const int buttonPin = 2; // the pin that the pushbutton is attached to`

`const 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`

These are well worded variables – the names describe the function of the variable. Some define pin numbers, some a counters, some are state trackers. These are all common functions of variables used in microcontrollers.

The setup() for this program is also a "standard" fair – we need to set the modes of the pins and initiate serial communication with the serial port. We use the pinMode() and Serial.begin() functions accordingly.

`

```
void setup() {

pinMode(buttonPin, INPUT); //  initialize the button pin as a input

pinMode(ledPin, OUTPUT);  // initialize the LED as an output

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

}
```

`

Notice that we almost always use variables to define pin numbers – or any number for that matter – even if we only use the variable once. Why not just type the value of the pin in the pinMode() function then?

Because variables build in flexibility. Maybe right now you only use the hardcoded number once – but what if you realize that using it later down the program is advisable – now you have 2 hardcoded numbers – and before you know it by the end of the program instead of having just one hard coded number you have it typed in 5 times. Now if you want to change the value you will have to track down all the hard coded numbers – and I bet you will miss one – I always do.

As a rule of thumb, unless you absolutely, positively know that you will not change the value and that it will only be typed once – then make a variable for it. The baud rate that is used for Serial.being() at 9600 is a pretty stable number, an example of where hardcoding makes sense. There are few examples like this – because variables are better than numbers.

Moving on to the loop(). We start by sampling the state of the digital pin where we have the pushbutton attached. We want to know from the very start – is the button being pressed at this instant?

`buttonState = digitalRead(buttonPin); // read the pushbutton input pin`

The digitalRead() function will return a HIGH or LOW which we now conveniently have stored in the buttonState variable. The first thing we will want to check is "has the state of the pin changed since we checked it last?" We do this with the condition of an if statement.

`if (buttonState != lastButtonState) {`

We just stored the current button state, and now we compare this with the lastButtonState using the NOT operator which is != . The != means "not equal to". So all this condition is saying is "If the current button state does not equal the previous button state, then do something." So if the button state is the same, this condition is not met, and the of statement will be skipped. It only cares about when state has changed. Now you might be wondering what lastButtonState is – well we initiated it at the top as 0, which is equivalent to LOW.

If this sketch was running on your Arduino for a couple seconds, and you pressed the button, this if statement would get executed – because the buttonState had been LOW and when you pressed the button, it went to HIGH. Now when you release the button, the state of the pin will go from HIGH to LOW – this also will engage the if statement. These are called state changes, and so far we have a state change deterctor…So what happens when it detects a state change?

The first thing we encounter in this if statement is an if/else statement.

`

```
if (buttonState == HIGH) {

// if the current state is HIGH then the button

// went from off to on:

buttonPushCounter++; // this just adds one to the value

Serial.println(“on”);

Serial.print(“number of button pushes:  “);

Serial.println(buttonPushCounter);

}

else {

// if the current state is LOW then the button

// went from on to off:

Serial.println(“off”);

}
```

`

Our first condition on this if statement is …

`if (buttonState == HIGH)`

This is checking to see if the button went from LOW to HIGH – in other words, the button went from not being pressed to being pressed. If this happens we do two things…

We increment the buttonPushCounter (the ++ just adds 1 to the variable) We print out some information to the serial monitor The stuff we print out just lets us know what the program is up to – it tells us how many times we pressed the button and what the current buttonState is at.

Now if the button was released – it went from HIGH to LOW, then the else statement will handle that – and all it does is print some info to the serial monitor that lets you know the current button state is LOW.

So everything we have talked about so far in the loop() can be summed up as follows:

Did the button get pressed or released? If the button was pressed: Increment the buttonPushCounter variable by 1 Print out some info about the state on the button and the number of times pushed 3. If the button was released:

Print out the state of the button That’s all there is to these big nested if statements. The next line of code we encounter is immediately after the close of these nested if statements – and what it does is update the lastButtonState variable:

`lastButtonState = buttonState; //assign the current button state to the last button state`

Now when the loop() starts again it will compare the lastButtonState with a newly sampled buttonState variable. In this way, we are always comparing what just happened to the button to the current state of the button.

The final block of code in this sketch is what will manage turning on and off the LED. Recall that it will take four button presses to turn the LED on. That why we were racking button presses. So if the button has been pressed four times – turn the LED on, otherwise, turn it off. We can implement this with an if statement.

`

```
if (buttonPushCounter % 4 == 0) {

digitalWrite(ledPin, HIGH);

} else {

digitalWrite(ledPin, LOW);

}
```

`

The condition used in this if statement is a little funky and new at this point. There is a percent sign symbol, this is called the modulo operator.

`if (buttonPushCounter % 4 == 0)`

To understand what this condition means we need to take a close look at the Modulo operator. The modulo operator returns the remainder of an integer division. With integer division, we do not use decimal points or fractions but we are left with remainders.

So if you divided 2 into 5, what would be the remainder?

5/2 = 2 remainder 1

So 2 goes into five twice with one left over.

What if you divide 5 into 17?

17/5 = 3 remainder 2

Ok, what if you divide 1 by 4? Now we are dealing only with integers here – not decimals or fractions.

1 / 4 = 0 remainder 1.

So four does not go into one at all – its too big – so the remainder is 1 (1 is what remains of the dividend).

2 / 4 = 0 remainder 2

3 / 4 = 0 remainder 3

4 / 4 = 1 remainder 0

So let's look at the if condition again…

`if (buttonPushCounter % 4 == 0)`

Recall that the variable buttonPushCounter is keeping tally of how many times we have pressed the button – so this condition asks "If I divide the number of times the button has been pressed by 4, is the remainder equal to zero?" The only time this condition will be met is when is when 4 divides evenly into the pushButtonCounter – and there is no remainder. So the ice cream scoop perfectly gets all the ice cream – non left over. The values 4, 8, 12, 24 or any multiple of 4 will satisfy this requirement.

What the modulo operator allows us to do is maintain a cycle.

If you follow along you can see that every fourth time, the modulo operator starts back at zero. In this way we can cycle an event however often we want.

Lets consider this final if statement one last time.

`

```
if (buttonPushCounter % 4 == 0) {

digitalWrite(ledPin, HIGH);

} else {

digitalWrite(ledPin, LOW);

}
```

`

So if the button has been pushed 4 times, turn the LED on – otherwise turn it off. This brings us to the end of the loop().

Knowing how to employ edge detection (also known as state change detection) can be useful for many applications. – it doesn't just apply to pressing buttons. Understanding when the modulo operator can help you is also a great little trick to keep up your sleeve.

Debouncing

In the last lesson you may have noticed that your button counts weren't exact – sometimes if you pressed the button once, it would register two or even three presses. Maybe you pressed the button four times in a row and it only registered twice. So if you would stop swearing at me I will happily explain.

There is a thing called bounciness – very technical I know – and it relates to the physical properties of buttons. When you press a button down, it may not make contact to both sides at the exact same moment – in fact, it may make contact on one side – then both – and then the other side – until it finally settles down. This making and breaking contact is called bouncing. It is not a manufacturing defect of the button – bouncing is implicit in most every physical switch.

It all happens in a matter of milliseconds – but your microcontroller is moving so fast that it will detect a transition between two states every time your button bounces. This is why your button count from the last lesson could be sporadic at times – it would register dubious bouncing state changes.

This lesson will explore one way to address debouncing code. Basically what we do is record a state change and then ignore further input for a couple milliseconds until we are satisfied the bouncing has stopped. This filters out the noise of a bouncy button.

The setup, the same as the previous lesson

The setup for this is the same as the previous lesson, so dont go tearing apart everything you just put together.

the Code

`

```
//initialize and declare variables
const int ledPin = 13; //led attached to this pin
const int buttonPin = 2; //push button attached to this pin
int buttonState = LOW; //this variable tracks the state of the button
// LOW if not pressed, HIGH if pressed
int ledState = -1; //this variable tracks the state of the LED, negative if off, positive   if on
long lastDebounceTime = 0;  // the last time the output pin was toggled
long debounceDelay = 50;    // the debounce time; increase if the output flickers
void setup(){
 //set the mode of the pins...
 pinMode(ledPin, OUTPUT);
 pinMode(buttonPin, INPUT);
}//close void setup
void loop(){
  //sample the state of the button - is it pressed or not?
 buttonState = digitalRead(buttonPin);
 //filter out any noise by setting a time buffer
 if( (millis() - lastDebounceTime) > debounceDelay){
//if the button has been pressed, lets toggle the LED from "off to on" or "on to off"
if( (buttonState == HIGH) && (ledState < 0) ){
  digitalWrite(ledPin, HIGH); //turn LED on
  ledState = -ledState; //now the LED is on, we need to change the state
    lastDebounceTime = millis(); //set the current time
}
else if( (buttonState == HIGH) && (ledState > 0) ){
  digitalWrite(ledPin, LOW); //turn LED off
  ledState = -ledState; //now the LED is off, we need to change the state
    lastDebounceTime = millis(); //set the current time
}//close if/else
  }//close if(time buffer)
}//close void loop
```

`

We head off this sketch with a handful of variables. Some used to define pins…

`const int ledPin = 13; //led attached to this pin`

`const int buttonPin = 2; //push button attached to this pin`

Others made to track the state of the button and the state of the LED

`int buttonState = LOW; //this variable tracks the state of the button, low if not pressed, high if pressed`

`int ledState = -1; //this variable tracks the state of the LED, negative if off, positive if on`

And finally some long variables to keep track of time – which when measured in milliseconds can become a real big number rather swiftly.

`long lastDebounceTime = 0; // the last time the output pin was toggled`

`long debounceDelay = 50; // the debounce time; increase if the output flickers`

Keep in mind that the basis of this debounce sketch is to silence input from the pushbutton at pin 2 after its first bit of input. So when you first press the button, the first time the Arduino registers that contact is made – it will take this reading from pin 2, and then ignore further input until 50 milliseconds has elapsed. That is why we need these time tracking variables.

The setup() for this sketch is rather simple – we are only setting the modes of the pins…

`

```
void setup(){

pinMode(ledPin, OUTPUT);

pinMode(buttonPin, INPUT);

}//close void setup
```

` The loop() is where things start to get interesting. You might have started to notice, that the first thing many sketches do is check the state of a pin – so every time through the loop we get a new sample of data from a pin and we assign this data to a variable – that way we have the most current conditions to work with. This sketch follows the same motif; we begin by checking the state of pin 2 to see if the button has been pressed or not:

`buttonState = digitalRead(buttonPin); //sample the state of the button – is it pressed or not?`

This is done with the familiar digitalRead() function which takes the pin number you want to check and returns either a HIGH or LOW, depending on what voltage is being “seen” at the pin. In this circuit, when the pushbutton is pressed, 5-volts is applied to pin 2 (HIGH), otherwise the pin is at ground voltage (LOW).

The next thing we normally do is test this value we just sampled against some type of condition. Not for this example – in this example we want to filter out the sample we just took based on when we received the last sample. So if the new sample came in just 1 millisecond after the last sample – we will ignore it. If it came in 2 milliseconds after the last sample, we will ignore it too. In fact, we only want to accept a sample that was taken at least 50 milliseconds after the last sample. How do we implement this as a condition? We use the microcontrollers internal clock with the function millis():

`if( (millis() – lastDebounceTime) > debounceDelay)`

This condition takes the current time and subtracts it from the last time a legitimate input was received it then checks to this if this span of time is greater then a preset threshold which is named debounceDelay. It basically says “Has enough time passed for me to even consider a new input?”.

This is the gate, the filter, that blocks the noise of a bouncing button. Once we know a reasonable amount of time has passed, we will accept the input and begin to process it.

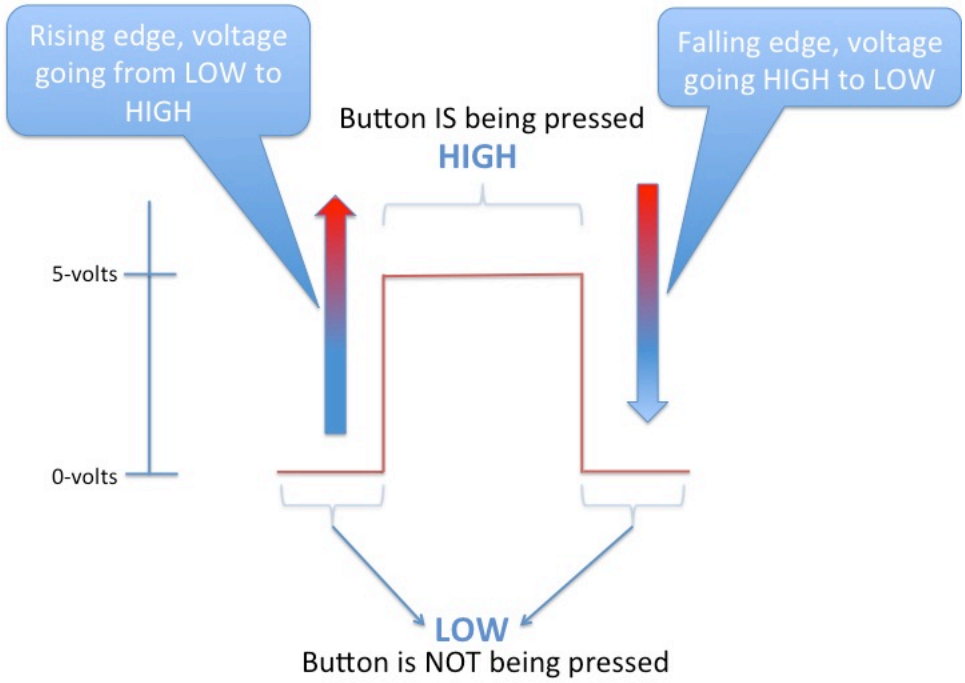
We use an if/else statement to do some further filter.

`

```
if( (buttonState == HIGH) && (ledState < 0) ){

digitalWrite(ledPin, HIGH); //turn LED on

ledState = -ledState; //now the LED is on, we need to change the state

lastDebounceTime = millis(); //set the current time

}
```

`

We are only interested in when the LED goes from LOW to HIGH – that is the rising edge of the input – when the button is pressed – this initiates the rising edge – when the button is released, this starts the falling edge.



The “if statement” checks these two conditions:

Is the input from pin 2 HIGH? Is the LED off? This is done by checking the sign of the ledState variable. You can see that we have multiple conditions that must be met – we use two ampersands (&&) to join these two conditions together:

`if( (buttonState == HIGH) && (ledState < 0) )`

This condition says “Is the button pressed AND is the LED off?” If so, then execute the if statement – if one of these condition is not met then skip this part. Both conditions must be met for the if statement to run.

Well if the button is pressed and the LED is off, then we want to toggle the LED on – and we will also want to update the state of the LED from off to on and we will want to update the lastDebounceTime.

`

```
if( (buttonState == HIGH) && (ledState < 0) ){

digitalWrite(ledPin, HIGH); //turn LED on

ledState = -ledState; //now the LED is on, we need to change the state

lastDebounceTime = millis(); //set the current time

}
```

`

We use digitalWrite() to apply high voltage to the LED at pin 13, this takes care of turning on the LED. We multiply the ledState variable by a -1 to change its sign from negative to positive (remember that we agreed if ledState was negative it means the LED is off, and if ledState is positive then the LED is on). Finally we update the lastDebounceTime to the current time using the millis() function again.

So now when we release the button, the LED will stay on – all we did was toggle the LED from off to on with a button press. Now what happens when we press the button again? We want the LED to turn off then. To do this we are still only concerned with the rising edge of the input – we want to know when button if first pressed again – but this time we want to address the scenario when the button is pressed and the LED is already on. The else if statement that follows the previous if statement does just that:

`

```
else if( (buttonState == HIGH) && (ledState > 0) ){

digitalWrite(ledPin, LOW); //turn LED off

ledState = -ledState; //now the LED is off, we need to change the state

lastDebounceTime = millis(); //set the current time

}//close if/else
```

`

Since the condition requires the buttonState to HIGH and the ledState to be positive (On), then we can easily toggle the LED off by writing digital pin 13 low.

Notice how this “if/else statement” has multiple conditions. The else is not just a catch all – it has a specific requirement itself. The general form of these if else statements is as follows:

`

```
If(condition){

Do something;

}else if(condition) {

Do something else;

}else if(condition) {

Do something else;

}else{

do this if no condition is met;

}
```

`

in this example we do not have a final else statement that is a catchall – we have set very specific conditions and we only want to act on these conditions. In this way we ignore the input that comes from the falling edge, when the button is released and the voltage at pin 2 goes from HIGH to LOW.

Now the next time through the loop – we take a sample – but this sample is going to be ignored, because it is coming too soon after the toggle of the LED – so it will be ignored and all the other inputs will be ignored until we reach the time threshold we set.

Now if you still have bouncing issues with your button, try increasing the debounceDelay variable form 50 to 100. This will ignore input even longer – but there is a price to pay. What if someone wants to rapidly toggle the LED by pressing the button very fast? This is where you will run into trouble if you make the debounceDelay too long. If you are making a video game controller – this could be a definite issue!

Try this on your own

- Try increasing and decreasing the debounceDelay time and observe the effect
- Add an LED to Pin 12 and change the code so that every time you press the button the LEDs toggle between each other. (i.e. one is on when the other is off)