Getting started with the Arduino

Charles West

April 12, 2014

1 The beginning

It began as thunder cleaved the sky and a dark figure labored on a windswept hill. Bracing against the elements, Massimo Banzi pitted his will against the order of the cosmos to tear open a scar in reality. Reaching into that scar, his face contorted from the effort and pain, Banzi drew forth something foreign to our reality. Blinking away the rain pelting his face, he stared in horror at his ruin of a hand. However, cradled in the crooked claw was a shard of light.

It had been worth it. From the powers primordial, he had torn a shard of the force of creation. It didn't look like much, just a tiny bobble. But in that bobble lay endless potential. The potential merely awaited the thoughts of men to give it form. With a cry of triumph, Massimo cast it into the air and sent it flying to the four corners of the Earth. It alighted to the minds of many mortals, who crafted the spirit brothers and sisters. In return, the bobble, in its many forms and guises, gave mortals the power to make their dreams come alive.

And that is where Arduinos come from (sort of, the real story is much more awesome).

The full story can be found here: `http://spectrum.ieee.org/geek-life/hands-on/the-making-of-arduino/0`

What the Arduino lets you do is write a page of text that specifies how the part of the world should work. Its reach doesn't extend very far into our reality (just a little part), but in that part, you have near absolute control and can rewrite how it works on a whim. With a little practice and some knowledge, you can gain skill at rewriting those laws and extend the Arduinos reach (letting you control more). Revel in your (micro) cosmic power!

Specifically, the Arduino can be plugged into your computer via USB and a C/C++ program uploaded into it via an IDE (or other ways). This program specifies how the Arduino should relate its outputs (either 0 volts or 5 volts) to time and its inputs. On its own, this isn't very powerful. It is a very small and isolated portion of reality that we have made our dominion. Part of the artistry (and skill) lies in figuring out how to extend this portion of reality so that it touches and affects the outside world.

Lets look at some examples of people that have done just that....

2 Examples

Have you seen a LED cube? A popular project has been to use an Arduino to control 512 LEDs to make small hologram like images. Not only can you see such a project already built, you can read the tutorial on how to do it yourself!

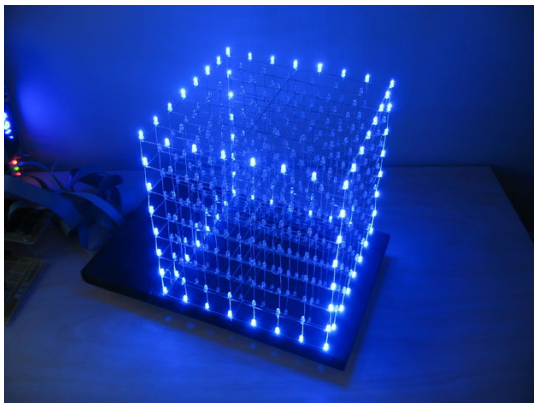

Figure 1: 8x8x8 LED Cube

`http://www.instructables.com/id/Led-Cube-8x8x8/`

Sick of not having someone to play chess with? Build your ultimate chess buddy. This maker used an Arduino to make a chess board that could sense all of the pieces and move them at will.

Figure 2: Chess playing robot

`http://www.instructables.com/id/How-to-Build-an-Arduino-Powered-Chess-Playing-Robo/`

Normal chairs got you down? Want to be able to drive around a segway without all of that pesky standing? Try building one of these self balancing chairs.

Figure 3: Sitway

`http://www.instructables.com/id/SITWAY/`

If those projects are too tame for you, you could build a fire breathing pony instead.

Figure 4: Fire breathing pony

`http://www.instructables.com/id/Make-a-Fire-Breathing-Animetronic-Pony-from-FurRea/`

3 Arduino Pros and Cons

The Arduino is not the cheapest or the most powerful board. It has one major feature that separates it from all the other boards. There are millions of people using Arduinos, contributing examples, tutorials and online help. This means that for 95+% of the projects you are going to do, there are multiple tutorials on how to do it posted online (including code) and lots of experienced people happy to help. In fact, if you are in the Raleigh area, there is a nearby group known as TriEmbed (formerly Triangle Arduino User Group) which regularly holds tutorial sessions and answers people's questions. They meet on the second Monday of each month in room 1021 of Engineering Building 2 at NC State University. There is also a very well documented library and set of examples to make doing things easier. These factors combine together to make a project "just work" much more often with an Arduino than most other boards. In addition, most hobbyist projects don't require really powerful processors or extreme power efficiency and the simplicity of chip makes it much easier to get started with.

The fact that the Arduino is open source and popular has two other effects. You can find a lot of "Arduino compatible" boards and "shields". Different Arduino compatible boards offer different advantages. You can find boards that are identical to the original Arduino for about 1/2 to 1/3 the price of the original. You can also find boards which are tiny (great for weight sensitive stuff such as quadcopters) but offer a nearly identical programming and output interface. This means that you can program on an Arduino and easily transport your code to different physical form factors.

Shields are electrics boards that are specifically designed to be hooked onto an Arduino (and only an Arduino) and provide extra functionality. Since there are so many people using the Arduino and the specs are open, there are a amazing selection of shields which offer a dizzying variety of applications. From a functional standpoint, this means that for a given application, there is probably a shield that has been made to make at least part of it much easier. The tradeoff is often one of money and space vs time. The shields often cost as much or more than the Arduino and stack on top. Sometimes a little skill can make a shield unnecessary but it is not always worth it to trade 8+ hours of labor for $25 (especially if you are in a hurry).

4 Getting started (Exercise 1)

The first step to getting up and running with an Arduino is to install the Arduino IDE. This is a free piece of software that will let you write code and upload it to your Arduino. Just head over to the site below and follow the instructions that are appropriate for your operating system:

`http://arduino.cc/en/main/software`

Because of the great community of people using Arduinos, there are a whole lot of examples of how to do things. Many of these examples have been integrated into the IDE so that you can easily take a look. It is traditional for people learning to program to first make a program that says Hello world!. The microcontroller version of that is to make an LED flash on and off. So after you have finished installing the IDE software, fire up the IDE and plug in your Arduino. It is time to load your first program!

To open the blinking LED example: Go to File → Examples → Basics → Blink

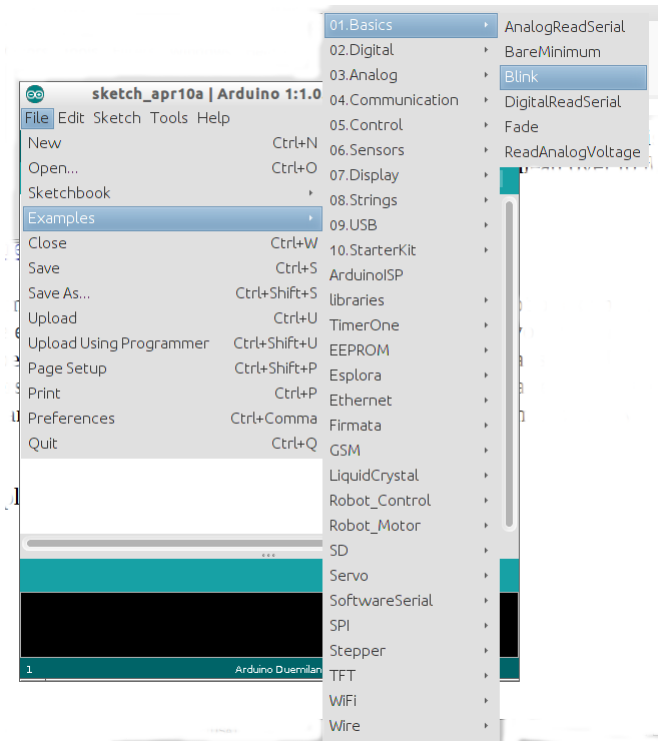

Figure 5: Menus for blink example

The program is reproduced here for discussion:

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

  This example code is in the public domain.
 */

// 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(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

In Arduino sketches, there are always at least two functions that are called. The setup function is run when the microcontroller powers up and is used to configure stuff that needs to be set once. After that, the loop function is called again and again as long as power is supplied to the Arduino.

In the blink example, we set one of the pins on the Arduino so that it is setting a voltage rather than listening to see what is applied (which is what it does by default). The particular pin that we are setting to an output is connected to the red LED that comes installed with the board. This means that when the pin (pin 13) is set to a high voltage (5 volts) that the LED will turn on. When we want to turn the LED off, we set the voltage low (0 volts).

To produce the blinking effect, we have the LED get turned on, then wait 1000 milliseconds (1 second), turn it off, then wait another 1000 milliseconds, then repeat (forever). The Arduino is really fast at doing simple things (like turning a light on and off), so we could shorten the delay until it is blinking so fast that it seems steady (but dimmer) when we look at it.

5 Exercise 2

Next, lets try getting a little more fancy with the LEDs. We are going to use our breadboard to connect several LEDs to the Arduino so we can make animated patterns.

Breadboard is really handy because it allows you to connect parts together electrically without having to solder them together. The holes in breadboard are connected as shown in the figure below. Generally, the two horizontal bars

on the top and bottom are used for high voltage (5 volts for an Arduino) and ground (0 volts) and the vertical bars are used to link components together.

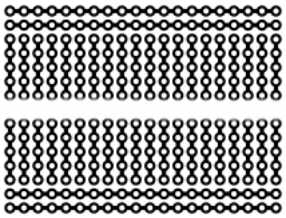

Figure 6: Bread board connections

We want to arrange it so that the positive terminals of our 3 LEDs are connected to pins 2,3 and 4 respectively. With LEDs, the longer lead is the one that is positive. Push the longer lead into the spots marked 2, 3, and 4 on the Arduino. Next we want to hook the power and ground bars on the breadboard so they correspond with the power and ground for our Arduino. Find the spot marked GND (ground) and connect it to one of the bars marked with a minus on the breadboard. Then find the spot marked +5 volts and connect it to the bar marked with a +. Next, push the other lead of the LEDs into the bar marked with a minus that has been connected to ground. This will allow the electrical current to flow out of the Arduino, into the LEDs and back into the Arduino. Take a look at the figure on the next page to see how the final result should look.

The next thing to try is making the LEDs light up in a 123123... pattern. You can either try figuring this out yourself, or take a look at the example in the source code. It is very similar to the code we used to turn on a single LED, just with a few more delays and lines of code. Another fun pattern to try is the Cylon/Knight rider pattern. Try making a 1232... pattern or take a look at the example.

6 A brief digression

Lets step back for a moment and look at what we have done.

We entered a few lines of code to describe how things should be The led should blink every two seconds, pressed a button and the LED started blinking. And blinking... and it will keep blinking as long as power is applied to the machine. The same goes if we build something more complicated like a self balancing chair or a fire breathing skeletal robot pony. We took a little bit of our thoughts and manifested an agent that will tirelessly work to implement the idea in the physical world. We do have to be very careful about how we specify those thoughts and our agent is limited in what it can do to achieve our goals. Even so, this is a powerful thing.

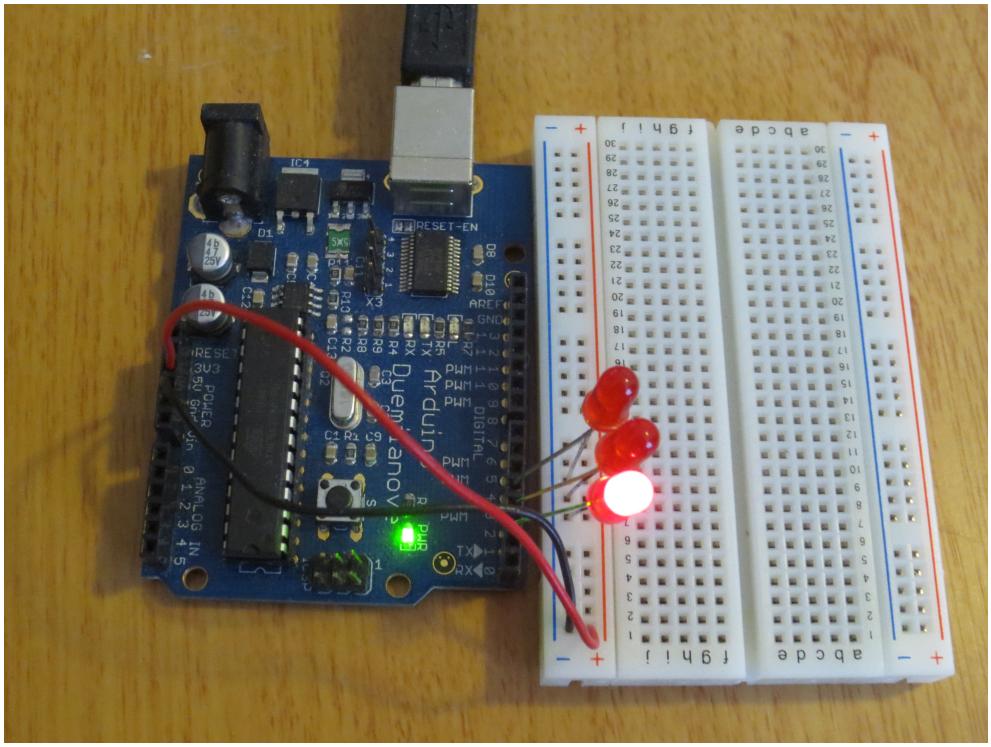

Figure 7: Breadboard connections for 123 blinking

7 Exercise 3

7.1 Physical setup

For our first sensor, lets connect a button. We'll set it so that it makes the an Arduino pin go to 5 volts normally but drop down to zero when we press down on the button. So that we can see what the Arduino is thinking, we'll set it such that when the button is depressed the LED turns off.

A push button is a switch that is only closed (connected) when we are pushing down on it. Otherwise it pushes back up and disconnects the two sides. If we connect the sensing pin on the Arduino to on side of the switch, connect +5 volts to the same side and connect ground to the other side of the switch, it should set the pin to +5 volts when the button isn't pressed and 0 volts when it is. There is only one (HUGE) design flaw there. If the button is pressed, +5 volts is connected directly to ground without any real resistance to restrict how much current flows. In the best case, this would result in your Arduino freaking out and either shutting down or doing unpredictable things. In the worse case this could fry your board.

To fix this problem, we need to add something something that restricts how much current flows. The sensing pins on the Arduino don't draw much current at all (you can usually get away with assuming 0 current draw), so if we just have a tiny trickle of current will let us set our voltages like we want. We will set this up by connecting the sensor side of our switch to +5 volts using a 10 Kohm resistor. The resistor restricts the maximum amount of current that our circuit can draw according to Ohm's law (V=IR). Since our circuit is a 5 volts and the resistance is $10*10^3$ ohms, the most current our circuit will draw is V/R = 5 volts / 10,000 ohms = .0005 amps (which is much better than approaching infinity).

From a practical standpoint, we have to use a few extra wires to connect things because the switch takes up a bunch of room on the breadboard and doesn't leave too much room for other connections. The final configuration looks something like this:

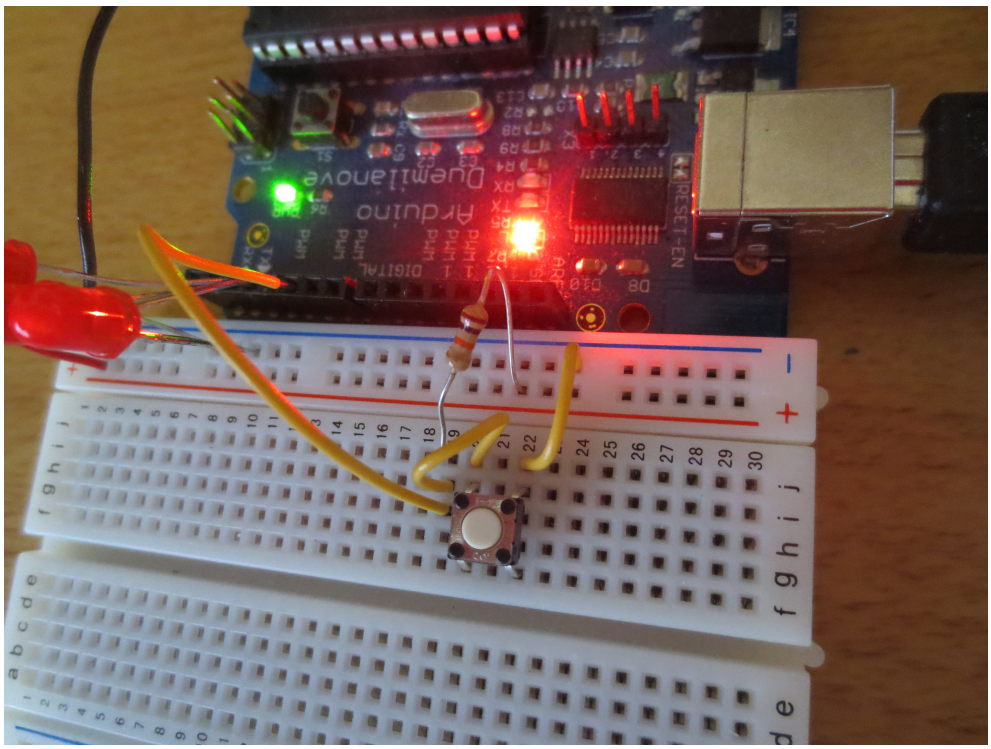

Figure 8: Breadboard connections for switch input

7.2 Programming setup

The code for the switch example is as follows:

```
// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;
int value = 0;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
  pinMode(5, INPUT);
}

// the loop routine runs over and over again forever:
void loop()
{
  //Read in the value from the switch
  value = digitalRead(5);
  //Set the LED to the value from the switch
  digitalWrite(ledPin, value);
}
```

We set the sensor pin (pin 5) as an input and keep the LED pin (pin 13) as an output. We then have the Arduino eternally alternate between reading the value (either 1 or 0) that is sensed on the input pin and setting the LED pin accordingly.

8 Exercise 4

Our setup worked pretty well for turning the LED on and off, but there is a catch that our eyes aren't fast enough to see on their own. When we press the button down, it doesn't smoothly go from 5 volts to 0 volts. Instead, it mechanically bounces really fast as we push down. The net result is that our Arduino sees a bunch of on/offs rather than just one. To have the nice switch behavior we expect (where we can just press a button once and it will toggle something), we have to debounce the button. There are two ways that people usually do this.

Adding a capacitor to the positive side of the switch will make it take a little time for the voltage to drop. This effectively reduces the voltage variation that the Arduino sees until it just sees one transition. However, this requires adding another component to the circuit.

Since the bouncing only occurs when we start pressing the button down (therefore we want at least one toggle), the other way to deal with the issue is to have the Arduino ignore any switch transitions that happen for a certain (fairly short) period of time after the first transition. This allows us to have debouncing without any addition components at the cost of a little more software complexity.

It has been mentioned before that the Arduino has excellent documentation. Now we are going to make use of it! There is a excellent tutorial on how to debounce an Arduino on the Arduino website (public domain) and we will simply modify that example to accomplish our goal of debouncing our switch. This code lets us reliably toggle the LED just by pressing the button. We don't have to modify the physical setup at all.

The link for the tutorial is as follows:

`http://arduino.cc/en/Tutorial/Debounce`

You can also just search the Arduino site.

The (barely) modified code is as follows:

```
/*
 Debounce

 Each time the input pin goes from LOW to HIGH (e.g. because of a push-button
 press), the output pin is toggled from LOW to HIGH or HIGH to LOW.
There's
 a minimum delay between toggles to debounce the circuit (i.e. to ignore
 noise).

 The circuit:
 * LED attached from pin 13 to ground
 * pushbutton attached from pin 2 to +5V
 * 10K resistor attached from pin 2 to ground

 * Note: On most Arduino boards, there is already an LED on the board
 connected to pin 13, so you don't need any extra components for this example.


 created 21 November 2006
 by David A. Mellis
 modified 30 Aug 2011
 by Limor Fried
 modified 28 Dec 2012
 by Mike Walters

 This example code is in the public domain.

 http://www.arduino.cc/en/Tutorial/Debounce
 */

// constants won't change. They're used here to
// set pin numbers:
const int buttonPin = 5;     // the number of the pushbutton pin
const int ledPin = 13;       // the number of the LED pin

// Variables will change:
int ledState = HIGH;         // the current state of the output pin
int buttonState;             // the current reading from the input pin
int lastButtonState = LOW;   // the previous reading from the input pin
```

```
// the following variables are long's because the time, measured in miliseconds,
// will quickly become a bigger number than can be stored in an int.
long lastDebounceTime = 0;  // the last time the output pin was toggled
long debounceDelay = 50;    // the debounce time; increase if the output flickers

void setup() {
  pinMode(buttonPin, INPUT);
  pinMode(ledPin, OUTPUT);

  // set initial LED state
  digitalWrite(ledPin, ledState);
}

void loop() {
  // read the state of the switch into a local variable:
  int reading = digitalRead(buttonPin);

  // check to see if you just pressed the button
  // (i.e. the input went from LOW to HIGH),  and you've waited
  // long enough since the last press to ignore any noise:

  // If the switch changed, due to noise or pressing:
  if (reading != lastButtonState) {
    // reset the debouncing timer
    lastDebounceTime = millis();
  }

  if ((millis() - lastDebounceTime) > debounceDelay) {
    // whatever the reading is at, it's been there for longer
    // than the debounce delay, so take it as the actual current state:

    // if the button state has changed:
    if (reading != buttonState) {
      buttonState = reading;

      // only toggle the LED if the new button state is HIGH
      if (buttonState == HIGH) {
        ledState = !ledState;
      }
    }
  }

  // set the LED:
  digitalWrite(ledPin, ledState);

  // save the reading.  Next time through the loop,
  // it'll be the lastButtonState:
  lastButtonState = reading;
}
```

9 Exercise 5: Pulse Width Modulation (PWM)

9.1 Theory

So far, we can turn something small on and off and sense when a switch is open or closed. Lets see if we can get a little more fancy. It would be really cool with we could turn things partially on or off (like a dimmer). As it turns out, that is something that we can do without any additional hardware. We use a trick called pulse width modulation (often referred to as PWM). Here's how it works:

With the blinking LED example, we have the LED turn on for one second then turn off for one second (cycling forever). If we made the delay period really short and the LED flickered faster than our eye could notice (roughly > 60 cycles per second), then it would seem like the LED was just glowing steadily.

How bright would it look when it is flickering like that? If it is on and drawing full power about half the time, then it would make sense that it would be putting out about half the amount of light that it normally does. If our eyes average what goes on when they sample light, it would make sense that it would be about half as bright. As it turns out, that is about what it looks like.

OK. But what would happen if we made it so that the LED was only powered on 10 percent of the time (while still flickering faster than we could see)? Since it is only drawing full power 10 percent of the time, it would seem about 10% as bright.

That is pulse width modulation in a nutshell. You choose a frequency fast enough that the system/components you are working with don't notice the flicker and then vary the percentage of the time the voltage is on to make the voltage/power seem like a percentage of what it normally is. Raw, this works for things like LEDs (if the only function is for humans to watch it) or controlling the speed of a motor.

There is a catch, but it is a small one. If you look at the output on an Oscilloscope (which lets you see high frequency stuff), you will see exactly what the PWM signal looks like. It will be a bunch of regularly space rectangles at high frequency. If you feed that into a radio raw, it will make your signal have all sorts of high frequency components that you don't want instead of a smooth voltage curve. In an application that can actually use high frequencies (or at least pass them along), the approximation breaks down if you use raw PWM signals. Fortunately, you can run the PWM output through something called a low pass filter (which you can make with a few simple components) and that will make the circuit give out voltages pretty close to what you are trying to send out (so that 10% power PWM = (10%) * 5 volts = .5 volts).

9.2 Practice

Now that we know what PWM is, lets try it! We could just make a program that sets the LED to a specific brightness, but it could be difficult to tell how bright it is without another LED around. For the fun of it, we are going to try something a little more complicated. We are going to make an LED heartbeat,

which means the LED smoothly goes from off to on and then back again.

The code for this exercise is as follows:

```
// 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() {

  //Rising heartbeat
  //Use a unsigned 16 bit counter, so we can have values over 127
  //Do a thousand cycles, adding one to i each time
  for(uint16_t i=1; i<1000; i++)
  {
  digitalWrite(led, LOW);     // turn the LED off by making the voltage LOW
  delayMicroseconds(1000-i);            // wait for a 1000 - i microseconds
  digitalWrite(led, HIGH);    // turn the LED on (HIGH is the voltage level)
  delayMicroseconds(i);                 // wait for i microseconds
  }

  //Falling heartbeat
  //Use a unsigned 16 bit counter, so we can have values over 127
  //Do a thousand cycles, adding one to i each time
  for(uint16_t i=1; i<1000; i++)
  {
  digitalWrite(led, HIGH);    // turn the LED on (HIGH is the voltage level)
  delayMicroseconds(1000-i);                // wait for a 1000 - i microseconds
  digitalWrite(led, LOW);     // turn the LED off by making the voltage LOW
  delayMicroseconds(i);                 // wait for i microseconds
  }
}
```

In this code we use a few different new things. The delayMicroseconds function is a little different from the delay function in that it does waits for microseconds (10^{-6} seconds) rather than milliseconds (10^{-3} seconds). We use the shorter delay function because we want the frequency of our PWM to be pretty fast so that we don't notice it. Another new thing that is used is the for loop. It lets use loop a specific number of times by incrementing a counter variable and checking how high it has gotten.

In this specific example, we start the counter at value 1 ($i = 1$) and increment it at the end of every loop ($i++$). At the beginning of each loop, we check to see if i is less than 1000 ($i < 1000$). If that isn't true ($i >= 1000$), then we exit the loop. We use a special type of integer variable for i (uint16_t) so that it uses 16 bits to store the number. Normal integers in Arduino programming only use 8 bit and support + and -, which means that the max value for them is +127. Since we want to count to 1000, that isn't enough and we have to use a bigger

type.

Inside the for loop, we hold one of the states for i microseconds and then the other for $1000 - i$ microseconds. This means that as i goes from 1 to 999, we smoothly interpolate between being in state 1 most of the time and state 2 most of the time. The first for loop has us start with the LED mostly off and then build up to the LED mostly on. The second for loop starts with the LED mostly on and then ends with it mostly off. Together, they form a smooth rising and falling heartbeat.

10 Exercise 6: Analog to Digital conversion

Our output capabilities have been upgraded, so now it's time to upgrade our inputs. We can now (approximately) output variable voltages, so lets see if we could sense them and respond accordingly. The Arduino has a Analog to Digital converter (generally referred to as ADC). This lets it sense voltages between 0 volts and 5 volts and convert it into a digital representation. This is really handy when we want to sense analog things like temperature or the conductivity of something. The main problem with using the ADC to sense things is that sensors generally give different resistances depending on what they sense rather than different voltages. What we need is a way to convert resistances into voltages. Happily, this problem was solved a long time ago. We use something called a voltage divider.

The simplest form of a voltage divider is just two resistors and some wires. It makes use of how voltages across two resistors in series (one after the other) work. If you put a voltage on two equal value resistors that are in series and test the voltage in the middle, you will find that the voltage is half of the voltage you put across the combination. If you change up the resistor values so that they aren't even, then you will find that the voltage is divided as follows: Voltage in middle = (Voltage across combination)*(resistance of the second resistor / (resistance of the second resistor + resistance of the first resistor)). In essence, if the second resistor supplies 60% of the total resistance, it will get 60% of the total voltage.

This implies that we can put something with a variable resistance in series with something that has a fixed (and known) resistance and get a voltage between 0 volts and 5 volts that is related (if not linearly) to the resistance. You can normalize in software to get what you are trying to measure from the ADC value since you know the relationship.

Sometimes you just want a nice linear change in voltage. You just want to be able to turn a knob and get a smooth linear change. As with so many other things, the shared desire of mankind for this capability has lead to a very nice and inexpensive solution for it. It is called a potentiometer. It is a little round knob with 3 terminals. You just stick it in the bread board, connect 5 volts to the right side, ground to the left side and the ADC input to the middle (as shown below). As you turn the knob you get a linearly varying voltage going from 0 volts to 5 volts. Pretty cool, right?

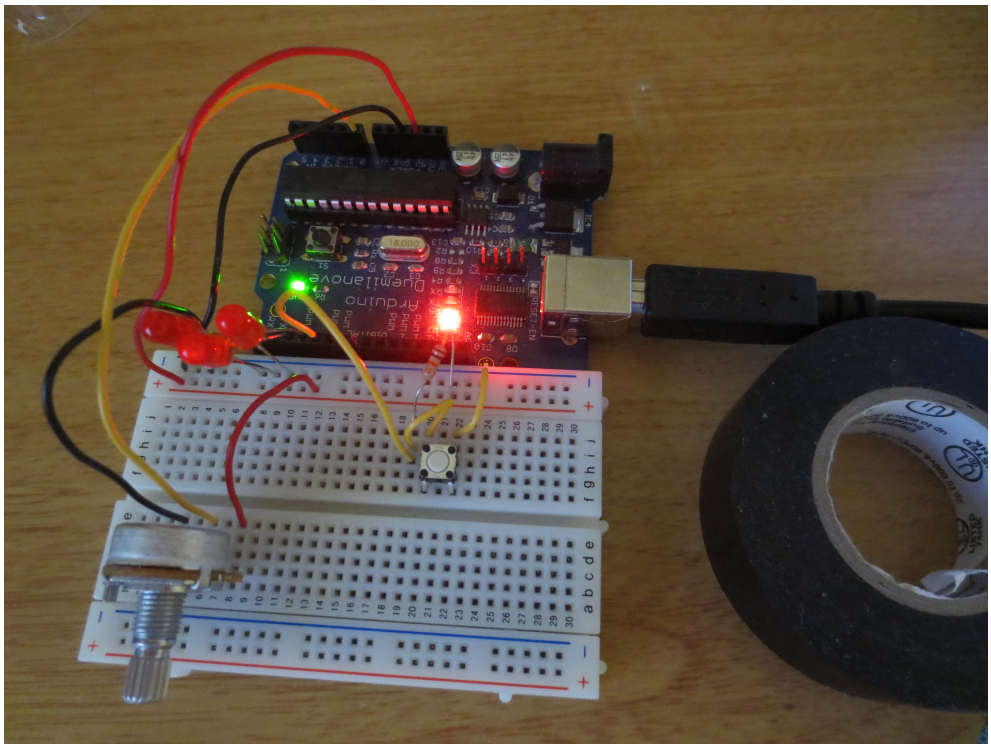

Figure 9: Breadboard connections for ADC with potentiometer

How it works is that it has a sliding contact in the middle. As you move the wiper across, the first resistor value gets bigger and the second resistor value gets smaller. The net result is a linear change in the potential.

Lets try it out. Wire up the configuration as shown in the figure and fire up the code shown below (if you are new at programming, I would suggest you manually type it as it give you a chance to learn to correct common coding errors and examine the code closely). This is yet another example of great code the community has made for the Arduino.

```
//Code retrieved from http://www.arduino.cc/en/Tutorial/Potentiometer
/* Analog Read to LED
 * ------------------
 *
 * turns on and off a light emitting diode(LED) connected to digital
 * pin 13. The amount of time the LED will be on and off depends on
 * the value obtained by analogRead(). In the easiest case we connect
 * a potentiometer to analog pin 0.
 *
 * Created 1 December 2005
 * copyleft 2005 DojoDave <http://www.0j0.org>
 * http://arduino.berlios.de
 *
 */
```

```
int potPin = 0;     // select the input pin for the potentiometer
int ledPin = 13;    // select the pin for the LED
int val = 0;        // variable to store the value coming from the sensor

void setup()
{
  pinMode(ledPin, OUTPUT);  // declare the ledPin as an OUTPUT
}

void loop() {
  val = analogRead(potPin);     // read the value from the sensor
  digitalWrite(ledPin, HIGH);   // turn the ledPin on
  delay(val);                   // stop the program for some time
  digitalWrite(ledPin, LOW);    // turn the ledPin off
  delay(val);                   // stop the program for some time
}
```

At the start of the loop, the program reads in the value of the voltage from the potentiometer (commonly known as a pot). It then does the normally hello world blink with the frequency determined by the value of the ADC. This can range from pretty slow to faster than our eyes can see. You may notice that the number given to the analogRead function doesn't correspond to the pin number we connected. This is because not all of the Arduino pins are connected to the Analog to Digital converter. If you look at the board, you will see that there are a few pins marked ADC. These are the ones we are addressing. In this particular example, we are using ADC pin 0.

11 Exercise 7: Interrupts

So far, our programs have done exactly one thing at a time. You might think that that is the only way things can be done when you have just one processor. Fortunately, the Arduino and other microcontrollers have peripherals that run independently of the main processor and this gives us the ability to multitask. These peripherals consist of devices such as timers and transition detectors. Timers can be used to trigger interrupts periodically (and do some other things). The transition/edge detectors can be used to trigger an interrupt when one of our pins go from high to low or low to high.

OK. Wonderful. So what is an interrupt? An interrupt is a function that has been registered to be called when a specific event happens. In the case of a timer, this means that the interrupt function can be called periodically. When the interrupt is called, it halts the execution of the main loop and does its stuff, then sets the main loop back running like nothing ever happened. Ideally, this means that we can write our main loop code without having to pay much attention to what is going on with interrupts (letting interrupt code and main loop code run pretty much in parallel). As with so many things, this works most of the time but there are a few catches.

The main loop is suspended while interrupt code runs. This means that if we want to minimize the effect the interrupts have on the main loop code, we

want the interrupts to be as short as possible. Other interrupts are also delayed when an interrupt is running, which is another good reason to make them run fast. Since the main loop is delayed when the interrupt code runs, this means that our delay calls can be a bit longer than we expected. In generate the delay is pretty small (fast interrupts), but it is something you have to account for if you need accurate timing (though you would probably be better off using a timer interrupt rather than the main loop in that case).

Lets try using an interrupt. Specifically, we are going to use a nice set of library functions that a Arduino user wrote to make a function an interrupt for timer 1. In this example, we make it so that the LED blinks every second without tying up the main loop at all.

The first step is to load the library. Go to Sketch $\rightarrow$ Import Library $\rightarrow$ Add Library. Navigate to the library file in the source code and double click on the TimerOne.zip file. Make sure the directory the file is in doesn't have any - characters or the IDE will complain and not load the library.

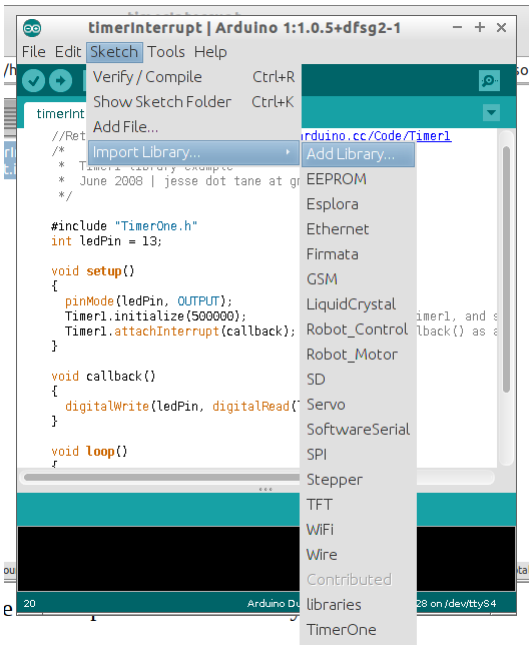

Figure 10: Loading the timer library

Next load or copy the code below:

```
//Retrieved from http://playground.arduino.cc/Code/Timer1
  /*
   *  Timer1 library example
   *  June 2008 | jesse dot tane at gmail dot com
   */

  #include "TimerOne.h"
  int ledPin = 13;

  void setup()
  {
    pinMode(ledPin, OUTPUT);
    Timer1.initialize(500000);          // initialize timer1, and set a 1/2 second period
    Timer1.attachInterrupt(callback);  // attaches callback() as a timer overflow interrupt
  }

  void callback()
  {
    digitalWrite(ledPin, digitalRead(ledPin) ^ 1);
  }

  void loop()
  {
    // your program here...
    delay(1000);
  }
```

In the timer example, we first set up the timer object and set its period (Timer1.initialize(500000)). We then tell it that the function entitled callback is an interrupt function it should call when the timer goes off. We then define the callback function so that when it is called it will toggle the LED by setting it to the opposite of whatever its current value is (digitalRead(ledPin) $\wedge$ 1 evaluates to 1 if the LED is set to 0 and 0 if the LED is set to 1). Lastly we have our main loop, which just delays and otherwise does nothing (but could do anything, since it doesn't have to worry about toggling the LED).

In practice, interrupts are great for collecting data without burdening the processor too much and doing stuff like PWM without needing to tie up the main loop. In more advanced embedded applications (using boards/chips other than the Arduino), it is pretty common to have almost all of the functionality carried out by interrupts and to turn off the processor to save power whenever there isn't an interrupt running. There are a lot of things that you can do with interrupts, and the online documentation has a lot of resources if you are interested in learning more.

Along those lines, several of the examples were written in such a way to show the concept rather than to give the easiest way to implement it. In general, it is not advisable to tie up your main loop doing PWM. There are a great many convenience functions that the Arduino library has to offer that makes many of these common tasks (such as PWM) really easy. As you move forward with your mastery of doing things with Arduinos and Embedded systems in general, I would advise you to read the documentation associated with the libraries you work with. For the Arduino in particular, documentation and community can make your life easier and enable you to be able to do so much more.

This is a good place to start:

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

`http://arduino.cc/en/Tutorial/SecretsOfArduinoPWM`

`http://playground.arduino.cc/Learning/Tutorials`

Also, don't forget that there is a local user group:

`http://triembed.org/`

These next sections are going to lightly touch on a bunch of different subject areas in an attempt to give you some idea where to go next as you do things (or encounter things).

12 USB communication

The Arduino has the capability to communicate over USB. It takes some work to get the USB data to do something useful on your PC, which is why it wasn't covered in depth in this tutorial.

However, if you would like to be able to print text and numbers from the Arduino to a terminal where you can see it (so that you can see the values from the ADC and so on), check out the following tutorial from Adafruit:

`http://www.ladyada.net/learn/arduino/lesson4.html`

13 Places to buy electrical parts

Finding the parts you need for a project can sometimes be half the battle. In the USA, most people buy their parts online. There are a few big names that most people use. The author doesn't have any endorsement deals or real preference where you shop, but hopes this information will be useful.

Digikey and Mouser are two huge electronics supply stores. They are some of the companies that are commonly used when you are buying a lot of parts. That said, they still offer hobbyist quantities as well. They use to charge more for shipping (forcing you to use UPS or Fedex) but have recently offered domestic shipping that is competitive with any of the others. In general, they are a good option of you know the part number you want (particularly if you are buying a lot of stuff).

Digikey: `http://www.digikey.com/`

Mouser: `http://www.mouser.com/`

The trick with the big companies is that it can be quite difficult to go from I want a nice sensor that I can plug into my bread board to measure PH to finding the part that you want. Their menu systems lets you sort by technical specifications but doesn't offer much in the way of advise.

Adafruit and Spark Fun are the opposite. They are (mostly) more expensive than Mouser and Digikey but offer a lot of support in figuring out what to buy and what would work well in your project. If you are trying something new or want ideas for cool project, looking through Adafruit or Sparkfun and see what stuff catches your eye. They also offer a bunch of tutorials on how to get going with each part. You can either support their efforts to help hobbyists by purchasing from them or get the part number from the data sheet for the part and buy it from Digikey or Mouser.

Sparkfun: `https://www.sparkfun.com/`

Adafruit: `http://www.adafruit.com/`

14 Transistors

Darlington transistors are a handy component for a large variety of projects. They act similar to a switch that is opened and closed by applying a electrical current to one of the 3 pins. Since there is a limit to how much power you can draw from an Arduino (and not all parts work at 5 volts), it is really handy to be able to switch things on and off. It allows you to power the devices you control directly from a power source, bypassing the Arduino while still retaining control. You can use them to turn motors on and off, as well as other high power things such as solenoids.

The TIP120 is a commonly used part, but I am also quite fond of the RFP12N10L. Both have a similar physical package, but I believe the RFP is a little more rugged.

One thing that you should always keep in mind when working with transistors is that voltages are relative. If you connect a separate power supply to

the source and drain of the transistor and just hook the pin of the Arduino to the gate, you will be unpleasantly surprised. The voltage of the Arduino doesn't mean anything to the power supply if the ground of the power supply isn't connected to the ground of the Arduino. This means that it is like you didn't connect anything to the transistor gate (it's floating) which results in unpredictable/not good behavior.

The solution is to connect the ground of the Arduino to the ground of the power supply. This is called having a common ground. There are two catches, however.

When a motor stops or you turn off a solenoid, you can get a quick peak of negative voltage. This can reset your Arduino whenever current is applied (sometime many times per second) and make it act really weird. The easiest solution is to stick a reasonably large (4700 uF isn't bad) capacitor on the power/ground rail for the Arduino (if using an electrolytic capacitor, make sure you connect the positive and negative terminals correctly, just like an LED). This give a bit of a storage buffer that helps smooth out the spikes you get from motors and solenoids.

The other catch is that your power supply is often at a much higher voltage than 5 volts. This means that if you mess up when you connect things, you can damage components and potentially fry circuits. The solution is to be really careful and see if you can test your circuit with lower voltage first. This can give you a better chance to get it right when you turn on the juice. When working with higher voltages, remember to take your time and make sure it is correct. The author recently completely fried an Arduino and destroyed two USB ports on his computer because he tried to connect a high voltage circuit in a hurry. Its similar to the old machining adage: Measure twice, cut once.

You can find more information on using Darlington transistors with the Arduino here:

`http://www.instructables.com/id/Use-Arduino-with-TIP120-transistor-to-control-moto/`

15 Servo motors

Servo motors have a PID controller in them, which is basically a fancy way of saying that they try to get to a certain rotation and then hold that position. What is nice about servos is that they are generally fairly strong and will resist any force that tries to move them from their set position. This means that you can tell your turret to point at 90 degrees and be fairly certain that it will point that direction even if you don't have a sensor watching it. One trade-off with servos is that they generally have a limited range of rotation (often +- 180 degrees).

You control servos with PWM. Each servo generally has a specific frequency they expect and if you deliver PWM with that frequency then you can set the position the servo tries to get to by setting the on percentage (simulated voltage) of the PWM.

There are Arduino libraries to help you work with servos, such as this one:
`http://arduino.cc/en/reference/servo`

Here is an example which controls one servo:
`http://playground.arduino.cc/Learning/SingleServoExample`

16 H-Bridges and DC Motor control

H-Bridges are generally used to control DC motors. DC motors spin continuously with a torque and direction dependent on the voltage they are driven with (generally meaning that you can control how hard they work with PWM). If you reverse the voltage, most DC motors will start spinning the opposite way.

H-Bridges are made with a set of 4 transistors can allow you to set the direction (polarity) of a voltage as well as whether it is on or off. Wikipedia offers a fairly good explaination of how they work:
`http://en.wikipedia.org/wiki/H_bridge`

You can make H-Bridges by hand simply by correctly connecting a set of individual transistors. However, this takes a bit of work and the cost of the individual transistors can add up (though the benefit is that they can generally handle a lot of power). It is more common for people to by premade H-bridges and integrate the premade chips into their board. This also helps deal with the fact that if you send the wrong inputs to a hand made H-bridge you can turn it into a short circuit (and probably fry your parts).

A good tutorial on using a premade H-Bridge chip:
`http://itp.nyu.edu/physcomp/Labs/DCMotorControl`

17 Divides in electronic part types

17.1 3.3 volt parts vs 5 volt parts

There are two types of electronics components that you will find for sale today. The ones that run at 5 volts and the ones that run at 3.3 volts. Modern commercial electronics are trying really hard to be low power to save energy for so that a user can save both money and battery life. In general, power usage goes up with the square of the voltage (assuming the same resistance). This means that components that run at 3.3 volts can save a significant amount of power. The net result for industry is that most new and modern parts for professional electronics run a 3.3 volts rather than 5 volts.

The net result for you is that a 3.3 volt part in a 5 volt circuit will likely damage the part and work incorrectly. This means that you should check the proper operating voltage for a part (particularly one that was developed more recently) before buying the part and sticking it in your circuit. The other thing you should watch out for is that more and more parts and boards are making the switch to 3.3 volts. You might want to switch your projects over in a few years as well.

17.2 Through hole parts vs surface mount parts

The other big divide that you are likely to run into is through hole parts vs surface mount parts. Modern electronics try to squeeze as much functionality as possible into as little space as possible (while also using as little material as possible). This and other factors have shifted modern electronics manufacturers away from using through hole parts which require holes to be drilled in the printed circuit board for them to be soldered and to using surface mount parts which can be soldered directly on the surface of the printed circuit board.

Hobbyists have not, by and large, followed this trend. There are two big reasons for this. Surface mount parts can't just be connected using breadboard. They generally require you to design and manufacture a printed circuit board, which takes time and money (though much less than it use to). The other reason is that surface mount parts are often TINY. It can take a microscope, experience and a very steady hand to solder some of those parts. In general, the author would say that there are a lot of advantages to surface mount parts for professional quality boards, but you might want to get the hang of doing things with through hole components first.

18 Raspberry Pi VS the Arduino

This guide would be remiss if it did not mention the other side of popular hobbyist electronics. While the Arduino takes a page or two of code and dedicates all of its processing power to it, there are other boards that run modern operating systems and have as much complexity as a PC.

The Raspberry Pi was the first single board computer of this type to achieve wide spread popularity. For a cost similar to the Arduino (though it requires a bunch of peripherals to get it working, so it is more like 2-3 times the cost), this computer can run a actual Linux desktop with an HDMI output. In contrast to the Arduino, with its 8 bit 16 MHz processor, the Raspberry Pi runs with a screaming 32 bit 800 MHz processor. It would seem like the obvious choice would be to get a Raspberry Pi instead of an Arduino.

The truth is that the Arduino and the Raspberry Pi are good for different things and are best used in combination.

The fact that the Raspberry Pi runs an operating system makes it difficult to have precise timing over its digital outputs. In general, operating systems are suppose to take care of scheduling, so getting something like a PWM wave out of the Raspberry Pi can take some serious work. To get the full power of the Raspberry Pi, you need to have a fairly good understanding of how to tweak Linux, which takes some time and effort. In addition, the Raspberry takes a lot of power to run and is likely to drain a few batteries in just a few hours. In contrast, the Arduino could chug on for days.

On the flip side, you aren't likely to see the Arduino displaying HDMI video. And while you can get some audio out of the Arduino, you aren't likely to see it decompressing .mp3s any time soon. Also, it takes some fairly serious fiddling

(or more likely, a shield) to get the Arduino to talk over a network. These are all things that the Raspberry Pi can do with ease. In the end, the Arduino is a microcontroller and is good at doing the sort of things microcontrollers are good at (PWM, analog sensing, etc) and the Raspberry Pi is a PC and is good at the things PCs are good at (processing video, high intensity computation, networking, audio generation). By hooking the two together using USB, you can get the best of both worlds.

You can learn more about the Raspberry Pi here:

`http://www.raspberrypi.org/`

A popular board that tries to tread the middle ground is the Beagle Bone Black. It offers a similar output pin profile to the Arduino and a set of subprocessors that can be used to try to do low level time sensitive processing such as PWM. It is a bit more expensive than the Raspberry Pi, but offers a little better processing power.

You can learn more about it here.

`http://beagleboard.org/Products/BeagleBone+Black`

One of the extremes in this direction is the O-Droid U3, which has been created by a Korean company. Physically, it is smaller than the Raspberry Pi. It costs about 3 times as much and the company makes you buy $25 shipping from Korean (Fedex, fast but pricey). It also draws about twice as much power as the Raspberry Pi. On the other hand, it has a 32 bit 1.7 GHz quadcore processor and is estimated to be about 8-13 times as fast as the Raspberry Pi (depending on the operation type). The community around the O-Droid U3 isn't nearly as big as those for the Raspberry Pi or the Beagle Bone Black, but if you need a lot of computing power in a small and reasonably cheap package, it seems a pretty good choice.

You can find more information about it here:

`http://hardkernel.com/main/products/prdt_info.php?g_code=G138745696275`