Lights! Speed! Action!
Fundamentals of Physical Computing for Programmers

Erik Brunvand
School of Computing
University of Utah

v1.0, April 26, 2013

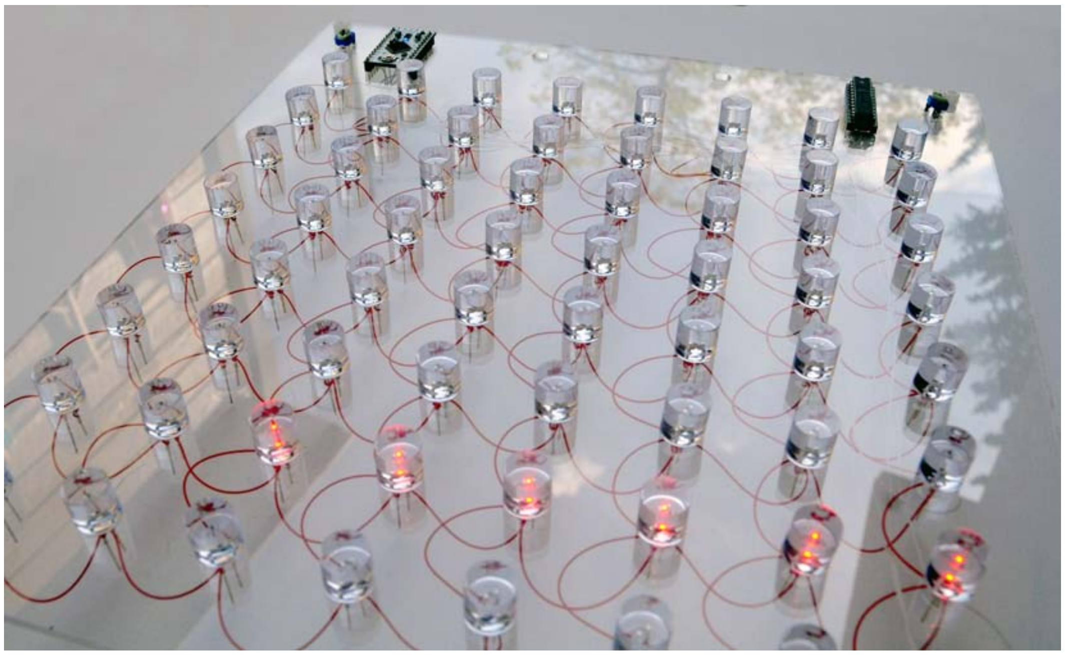


Figure 1: Serpente Rosso, Erik Brunvand 2012, acrylic, LEDs, wire, circuits, and computer control

Contents

Abstract

The definition of computer graphics as used by artists in new media and kinetic areas of the arts is much more expansive than simply rendering to a screen. A visit to the SIGGRAPH art gallery, for example, will showcase a wide variety of uses of physical computing, embedded control, sensors, and actuators in the service of art. This course is for programmers, educators, artists and others who would like to learn the basic skills and intuition necessary to incorporate physical components into their computing systems.

The course is targeted at programmers with little or no electronics background. We start with basic electronics concepts as they are used with these components. We then cover a variety of sensors that provide information about the physical environment (light, motion, distance from objects, flex, temperature, etc.), programmer-controlled lights (LEDs), and programmer controlled motion (servos, motors). We will describe the use of these components in the context of the Arduino microcontroller, but the skills learned will be general and should transfer easily to a variety of other computing platforms.

Although there will be a few simple formulas in this material, the strong focus will be on practical usage and common sense applications in real circuits. In fact, for those physicists in the audience, there are places where a common-sense or rule-of-thumb description will be used that may not be completely accurate in terms of electrostatic or quantum mechanical reality. Please take these models for what they are: intuitive descriptions that help understand the situation in practice, even though they may gloss over some second order effects. After taking this course you should feel more comfortable in selecting, wiring, using, interfacing, and controlling a variety of simple physical computing components. Following a few simple rules of thumb, and knowing what questions to ask, should keep you from blowing up too many components. These physical computing components can allow you to add physical computer graphics to your repertoire. That is, real, physical machines controlled by computers that make graphical marks!

About the Author

Erik Brunvand is an Associate Professor in the School of Computing at the University of Utah in Salt Lake City, Utah. His research and teaching interests include the design of application-specific computers, graphics processors, ray tracing hardware and software, asynchronous systems, and VLSI.

Starting in 2009 he has co-developed and taught an arts/tech collaborative course with a colleague, Paul Stout, in the Department of Art and Art History at the University of Utah entitled Embedded Systems and Kinetic Art (`www.eng.utah.edu/~cs5789/`). As an artist he is a printmaker, co-founder of Saltgrass Printmakers (a non-profit printmaking studio and gallery in Salt Lake City: `www.SaltgrassPrintmakers.org`), and also works in mixed-media computer-controlled kinetic arts.

elb@cs.utah.edu
`www.cs.utah.edu/~elb`

Course Schedule

5min: Introduction and motivation

20min: Section1: Electronics Fundamentals

- Charge, Voltage, Current - the water model
- Ohms Law - applications to physical circuits
- Kirchoffs Laws - physical arrangements of components
- Practicalities of wiring and powering circuits
- Arduino Platform

15min: Section 2: Lights! - LEDs

- Electrical properties
- Solutions for lots of LEDs

20min: Section 3: Speed! - Environmental Sensors

- Switches
- Resistive sensors
- Analog sensors
- Analog to digital conversion

20min: Section 4: Action! - Servos and motors

- Hobby servos
- DC motors
- Stepper motors

10min: Conclusions and Context

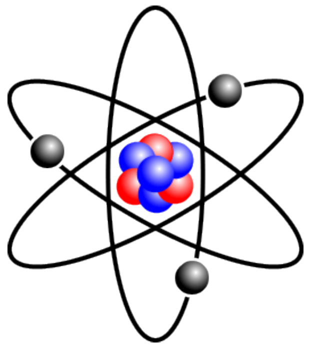
- - Lightening review of kinetic art and physical computing, and how this all relates to a broad definition of computer graphics in the fine arts.

These course notes have far more information than can be presented in a 90min course. The live course will present the high level concepts, and these notes serve as the details.

Chapter 1
Electronics Fundamentals

Why review basic electronics? You've probably seen all this in another class you took as an undergraduate or even in high school. However if you haven't used this information lately it's likely that you could use a review. Many of the physical computing components that we'll be talking about have specific voltage and current requirements, and the platform that you're using to control them (like the Arduino, for example) has certain electrical limitations. By reviewing basic electronics we'll make sure that we're all on the same page when it comes to asking the right questions about electrical interfacing, and also knowing what the right questions are in the first place.

Electricity is a term that encompasses a wide variety of physical phenomena related to the behavior of subatomic charged particles and electromagnetic fields. For our purposes we're mostly concerned about the movement of charge in response to an electric field. Where does the charge come from? That comes from an inherent property of subatomic particles. In the Bohr theory of the atom[1], a positively charged nucleus (positively charged protons and uncharged neutrons) is orbited by negatively charged electrons. The charged particles interact with each other: particles with the same charge repel each other, and particles with different charges attract each other. It is this electrostatic attraction/repulsion that keeps the electrons orbiting the nucleus and holds the atom together. This is, of course, quite an oversimplification, but it is a fine first-order model to understand charge and charge movement. Why is charge movement important? Because that's the essential part of electricity that we're concerned with. The movement of charge in a conductor is the source of power and functionality of pretty much all the physical computing components we'll be talking about. The movement of charge past a fixed point in a conductor is electrical **current** and is measured in **amperes** or **amps**. The force that causes the charge to move is measured in volts. Let's get some definitions out of the way:


By Halfdan [CC-BY-SA-3.0], via Wikimedia Commons

Conductor: A material where the electrons are weakly bound to the other atoms in the material so that when a force is applied (electric field), the electrons can move from atom to atom (electrical current can flow). Copper is an example of a good conductor.

[1]Niels Bohr, Danish physicist, 1882 - 1962, Nobel Prize in Physics, 1922.

Insulator: A material where the electrons are so tightly bound to the other atoms that it takes a huge amount of force to get them to move around. For practical purposes, the electrons don't move, so current doesn't flow. Glass is an example of a good insulator.

Current: The amount of charge moving past a point in a conductor. If you could count atoms as they move by a point in a conductor, you could directly compute the current. Charge is measured in **coulombs**. One coulomb of charge is the amount of charge on 6.241×10^{18} electrons. That's a LOT of electrons. Fortunately there are a LOT of electrons floating around in a conductor. Copper, for example, has 1.38×10^{24}free electrons per cubic inch.

Ampere: Also known as an **amp**, this is the unit of current in an electrical circuit. An ampere is one coulomb of charge passing a point in one second.

Voltage: The electromagnetic force that causes charge to move, measured in **volts**. Voltage is actually a form of potential energy. It's the amount of energy required to move one coulomb of charge from one point to another. Voltage is related to other measures of energy such as the joule. Raising the voltage of one coulomb of charge by one volt takes one joule of energy. The high voltage node in a system is often abbreviated as **VDD**. This is somewhat obscure notation based on the "Drain to Drain Voltage" in transistor based circuits.

Ground: This is an arbitrary point in an electrical circuit that is defined to have a voltage of 0. This is arbitrary because voltage is a relative term. Two points in the circuit are defined by their difference in voltage (their difference in potential energy). But, this is just a difference in energy. There is no absolute zero. We simply define one point in the circuit to be zero volts and measure voltage relative to that point. In your house, this point is actually the ground. There is a metal stake in the basement of most houses that goes into the ground and is defined as the arbitrary zero volt reference for your house. That's why it's called ground! Ground is often abbreviated GND.

Power: Measured in **watts**, power is simply volts times amps (equivalently joules/sec). This is instantaneous power. Power over time has slightly more complicated measures to smooth out the differences over time.

Big Idea #1

Charge moving in a conductor (current) under the influence of an electrical force (voltage) is the main electrical activity that we're interested in. This phenomenon powers LEDs, makes motors move, and is the property that we'll sense in a sensor to measure our environment. Causing current to flow, and controlling that current, is one of our main goals!

1.1 The Water Analogy

One way to get an intuitive feel for current and voltage is to think of electricity flowing in a conductor like water flowing in a river. The electrical current is very much like the water current. The voltage is like slope that provides the potential energy that lets the water flow. A higher slope provides more energy to move the water than a gentle slope. Like water, you can think of interesting situations with high and low current, and high and low power (voltage). Elaborating on this model a little bit:

High Current, High Voltage: This is like Niagara Falls: There is lots of water and lots of energy being dissipated. In electronics this is like a high-tension, high-voltage power

Table 1.1: A water analogy for systems with high/low current and voltage

	High Current	Low Current
High Voltage	By JohnnyAlbert10 [Public domain], via Wikimedia Commons	By Tomaszp [CC-BY-SA-3.0], via Wikimedia Commons
Low Voltage	USGS image, via Wikimedia Commons	By User:Ruhrfisch [GFDL] via Wikimedia Commons

transmission line. Long-range power transmission lines might be at 100-500 kilo-volts (kV) at 100 amps or more. Shorter lines might be 50kV at 10kA. Yikes! This implies kilo-Watts and Mega-Watts of power.

Low Current, High Voltage: Imagine Angel Falls in the dry season - the water is falling from a huge height (lots of voltage), but there isn't much of it (low current) so it doesn't really hurt when it falls on your head. Static electricity is a great example. When you rub your feet on the carpet and then touch a doorknob, you might be dissipating 10,000 or 20,000 volts! But this voltage is at very low currents so you can see the spark, but you don't get hurt. Energy dissipated ranges from 1millijoule to 50millijoules (a millijoule is a thousandth of a joule) (1 watt is 1joule/sec).

High Current, Low Voltage: Like the Mississippi river - lots of water, but moving very slowly. Still, there's lots of energy in that system! An arc welder is an electrical example: it might use a step-down transformer to bring the line voltage down to 20-40v, but at 600amps which can weld metal. Another example is a graphics chip by NVIDIA or ATI. Their high-end chips consume 200-300watts, but with a power supply of around 1v. That implies 200-300 amps of current (at low voltage) going into those chips!

Low Current, Low Voltage: like a small creek that doesn't do major work, but has just the right amount of energy to make a pleasant babbling sound. Think embedded computing here. Your phone runs for days on a battery - it's consuming tiny current (milliamperes or mA) at low voltages (1v). It's still doing a lot of stuff, but trying to consume as little power as possible doing it.

1.2 Ohm's Law

Ohm's law [2] describes the relationship between voltage, current, and a property of physical materials called resistance. Resistance, as the name implies, is the property of a material to resist the flow of electricity. The precise relationship is $V = IR$ where V is the potential difference between two points in the circuit (measured in volts), I is the current flowing between those points (measured in amps), and R is the resistance of the material (measured in ohms, often using Ω as a symbol).

Resistance to the flow of charge is similar to friction in a mechanical system. Or, using the water analogy, you can think of this as the diameter of the pipe through which the water (current) flows. A narrow pipe restricts the flow of water (current), whereas a larger diameter pipe lets the current flow more freely. One ohm is defined as the resistance in a system when one volt of energy produces one amp of current.

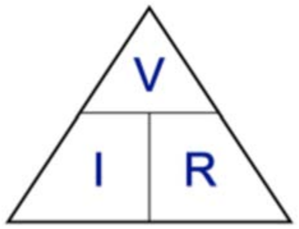
Big Idea #2

Ohm's law tells us about the relationship between voltage, current, and resistance. Suppose you want to keep the voltage the same, but reduce the current in a circuit? Ohm's law tells us we can do this by increasing the resistance. This is known as a current limiting resistor and we'll see it again soon. Very useful!

Big Idea #3

Ohm's law can tell us how to compute any one of voltage, current, and resistance, if the other two are known. You can use Ohm's triangle to remind yourself how to do this. Cover the unit you're looking for with you finger, and the remaining parts of the triangle tell you how to compute that unit. For example, if you want to compute the resistance in a circuit, block out R, and you're left with V/I as the way to compute that resistance.

Ohm's law makes a great deal of common sense if you think about the water analogy again. Imagine that you have a system of water running through a pipe with some force. If you make the pipe smaller, but don't increase the force (voltage), then the current must go down because of the smaller pipe (more resistance). This is the effect of a current-limiting resistor in a system - to reduce the current while the voltage remains the same.



In the same situation, if you want the same amount of current to flow through the smaller pipe, you need to increase the force (voltage) to force that current through the smaller pipe. This is used when a particular current is required in a circuit (a component that requires a fixed current). You then know how to set the voltage to achieve that amount of current.

Resistors are electrical components that look like little cylinders with colored bands printed on them, and wire connections on each end. The colored bands tell you how much resistance each component has. Reading the colored bands is a bit of an art form. There are many calculators on the web that can help you decode this. A couple examples are:

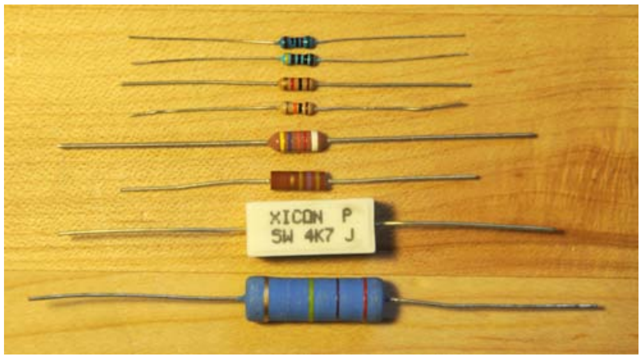
`www.hobby-hour.com/electronics/resistorcalculator.php`

`www.hobby-hour.com/electronics/resistorcalculator.php`

Resistors are not directional - it doesn't matter which end is connected to + and which end to -. They just act like smaller or larger pipes for the current to flow through. If you draw a resistor in a circuit schematic you usually use a zigzag symbol or a rectangle that looks like the right side of Figure 1.1. The R1 and R2 are the component names. In a complex schematic it's important to give each of the components identifiers so that you

[2]Named for Georg Simon Ohm (1789-1854), a German physicist and mathematician.

can refer to specific components by name.



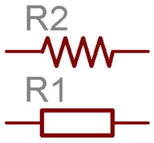


Figure 1.1: Some examples of resistors, and the schematic symbols used to represent resistors. Different physical sizes of resistors are related to their power-handling capability.

As a practical matter, the ranges of current, voltage, and resistance that you're likely to encounter in physical computing circuits areas follows:

Current: useful quantities for our circuits range from amps to milliamps (mA). A mA is a thousandth of an amp: 1mA = .001A. Amps are usually denoted as i or I in circuits for reasons that nobody is completely sure about.

Voltage: useful quantities for our circuits range from a few 10's of volts to millivolts (mV). Again, a mV is a thousandth of a volt: 1mV = .001V.

Resistance: useful quantities range from ohms, to kilo-ohms (kohm or kΩ), to megaohms (Mohms or MΩ). A kilo-ohm is 1,000Ω and a megaohm is 1,000,000Ω.

Some practical examples before we move on:

- **Q:** How much current does a 1200watt toaster draw from a 120v power line?

  A: If $P = VI$, then $I = P/V$, so I (amps) = 1200watts/120v = 10A of current.

- **Q:** If you have a 220v electric stove top, and the heating element has 30Ω of resistance, how many amps go through that heating element?

  A: $V = IR$, so $R = V/R$ = 220v / 30Ω = 7.33A

- **Q:** How much power does that heating element dissipate?

  A: $P = VI$ = 220v 7.33A = 1,613.33 watts

  A2: $P = VI$, and $V = IR$, so substituting in the first equation,
  $P = I^2R$ = $(7.33)^2 \times 30\Omega$ = 1,613.33 watts. Whew!

Before we move on, we should make clear one confusing aspect of voltage, current, and resistance. Current is measured in terms of positive current. That is, it is measured in terms of charge moving from the more positive point in the circuit (typically the power supply) to the more negative point in the circuit (typically ground). BUT, the actual things that are moving are the negatively charged electrons!

Confusing Concept #1

So, the current that we measure as positive current is moving in the opposite direction as the actual charge that is moving. Believe it or not, this confusion goes back directly to Ben Franklin.

Yes, THAT Ben Franklin. During his experiments with electricity, he arbitrarily chose to define one type of charge negative, and the other positive. Later, when batteries were more common, it was natural to consider current flowing from the positive to the negative terminal of the battery. Only many years later did we discover that the actual subatomic particles that were moving were the electrons, and they moved in the opposite direction as the defined current. What a mess! But, we're stuck with it now...

Just remember: current is measured as positive charge moving from the higher voltage in the circuit to the lower voltage. But, if you want to confuse yourself, remember that although you're thinking about positive charge moving to ground, inside the conductor it's actually the other way around: negative charge is moving towards the higher voltage!

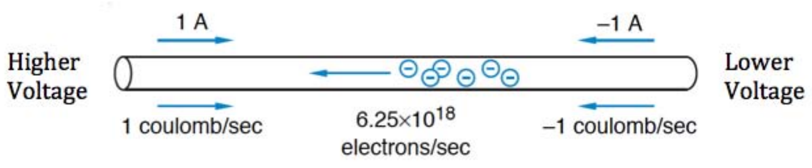


Figure 1.2: Conventional current (positive current) moves from + (higher voltage) to - (lower voltage). The actual majority charge carriers are negatively charged electrons moving from - to +.

Big Idea #4

If that's too much, just ignore it. Current is positive and moves from the higher voltage (the + end of the battery, or Vdd) to the lower voltage (the - end of the battery, or Ground).

1.3 Kirchhoff's Laws

Kirchhoff's laws[3] tell us about how voltage and current behave in an electrical circuit that has series and parallel connections of components through nodes and loops. First let's make sure we know what a node and a loop are in a circuit.

Big Idea #5

A node in a circuit is any part of the circuit that is connected together through a conductor. All points on the node are at the same electrical potential (voltage) because they are connected. As long as you connect all the pieces of the node together through a conductor, the physical circuit will be the same as the schematic. A loop is a path that starts at a node, passes through connected components, and ends up at the same node at which it started. With a battery in the circuit the loop can start at the + end of the battery and end at the - end. Without a battery, a loop can be from the voltage source (5v say) through components to ground (0v). That may not look like a loop, but it is because the battery node is implied by the 5v (+) and 0v (-) in the circuit.

Kirchhoff's Voltage Law (KVL): The sum of voltages around a loop is zero. This is like saying that if you are hiking and you start at the parking lot, go up a hill, down a hill, and end up back at the parking lot, you will have a net altitude gain/loss around the loop of zero. You will have gone up just as much as you went down because you ended up right where you started.

Kirchhoffs Current Law (KCL): The sum of the currents going into and out of a node is zero. This is like saying that all the current that flows into a node must also flow out. This is like a river splitting into three branches: the water upstream that comes into the split must all flow out through the combination of the output branches.

[3]Gustav Robert Kirchhoff (1824 - 1887), German physicist.

In a way, these are obvious laws, but to some extent that's their point: that current and voltage obey the obvious rules that seem to make sense.

There's a big idea hiding in Kirchhoff's Voltage Law: If the battery supplies a certain amount of voltage (say 5v), then KVL tells us that all of that voltage is gone when you get back to ground (0v). Where did it go? It got used up through each of the components in the loop. There is a *voltage drop* across each of the components, and those voltage drops add up, in this case, to 5v. How much voltage drops across each component? Ohm's Law tells us (combined with Kirchhoff's Current Law). If all the components in the loop are in series, then they all see the same amount of current (KCL tells us that the current can't sneak out through magic). If all the components in the loop see the same current, then the voltage across each component is related to Ohm's Law: $V = IR$.

Big Idea #6

The voltage drop across a component is directly related to that component's resistance. This is the basis of a voltage divider. A voltage divider is a series connection of resistors. KVL and KCL tell us that the resistors will see the same current, and the total voltage drop across both resistors will equal the total voltage. Ohm's law tells us that the drop across each resistor is directly related to its resistance.

The circuit for a basic voltage divider is shown in Figure 1.3. There are two resistors in series between power (Vdd) and ground (GND). The voltage will *drop* across both resistors in proportion to their relative resistances.

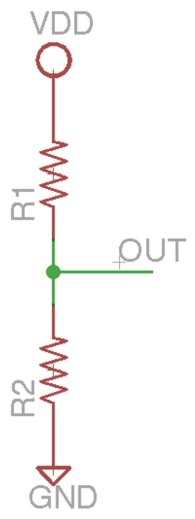


Figure 1.3: A simple voltage divider with two resistors

Q: If Vdd is 5v, R1 is 100Ω, and R2 is 200Ω, what is the voltage at OUT (with respect to ground at the bottom of the circuit)?

A: $I = V/R$ for the whole circuit
$5v/(100\Omega + 200\Omega) = .0167A$
$V1 = I \times R1 = 0.167A \times 100\Omega = 1.667v$ (Vdd to Vout)
$V2 = I \times R2 = 0.167A \times 200\Omega = 3.333v$ (Vout to GND)
Check result: $1.667 + 3.333 = 5v$ (total drop)

Voltage dividers are extremely useful circuits that allow us to divide a voltage into many parts, or to compute how much voltage is dropped across a specific component. We'll see them later when we talk about sensors!

Another very useful practical result of Kirchhoff's Laws relates to current in a branching node. In a voltage divider, we use the fact that all the series-connected components see the same current - there's nowhere else for that current to go. But, if there is somewhere else for it to go, it gets split up in proportion to the resistance of the branches. This makes sense back in our water analogy: if there is a large input pipe (lots of current) branching into three smaller pipes (resistors), then the current is split in proportion to the diameters of the smaller pipes. If all the smaller pipes are the same diameter, the current is split into thirds.

This will come in handy when we talk about things like LEDs where we want a specific amount of current to flow through each component. Kirchhoff's and Ohm's laws can tell us exactly what we need to know!

1.4 Practicalities of Wiring and Powering Circuits

Circuits can be wired up using any conductive material, but insulated wires are the easiest and most common materials used. A schematic tells you about the logical organization of the electrical components. It uses standard symbols that represent electrical components and lines that represent electrical nodes. It's a recipe for how you should assemble the physical circuit.

The physical realization of that circuit could look very different, but as long as you connect the components into the same set of nodes and loop as in the schematic, it will have the same function as the schematic. For example, the following is an extremely simple schematic that describes two components connected in series: an LED (the triangle-ish component) and a resistor (the zig-zag). The schematic says that you should connect one terminal of the resistor to Pin 10 of the Arduino Controller, and the other end of the resistor to one end of the LED. Finally, the other end of the LED should be connected to GND (ground).

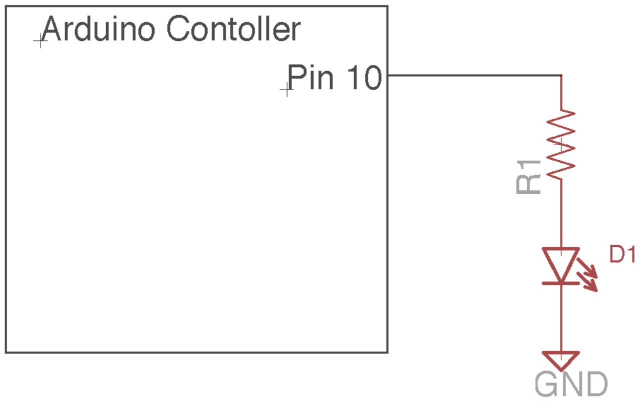


Figure 1.4: A simple schematic diagram showing a large component on the left (Arduino Controller) connected through Pin 10 to a resistor in series with an LED to ground.

To make this circuit you would find physical examples of all the main components (the Arduino Controller, the resistor, and the LED) and use wire to connect them together. This schematic doesn't specify the resistance value of the resistor, or the color or electrical specifications of the LED. In a real circuit that information would be annotated on the schematic, or in a supplemental document that references the component names R1 and D1. You could solder wires and components together to make a nice solid permanent bond, and in fact you probably want to do this for your finished product. But, for prototyping and testing, it's nice to have a less permanent, and less complex solution.

One way of doing this is with a solderless breadboard or prototyping board. This is a board that has lots of little holes in it in which you can poke a wire. Inside the board are connections such that all the wires plugged into one row will be electrically connected (that is, that row is one electrical node). Using these breadboards, you can prototype a circuit by plugging and unplugging wires and components into the holes, and you can quickly change things to try out new ideas, or fix bugs.

In this breadboard, the vertical columns on either side of the board that are marked with red and blue lines are vertical buses. Any wire plugged into that column (e.g. the columns marked in Figure 1.6) will be connected to any other wire plugged into that column. The

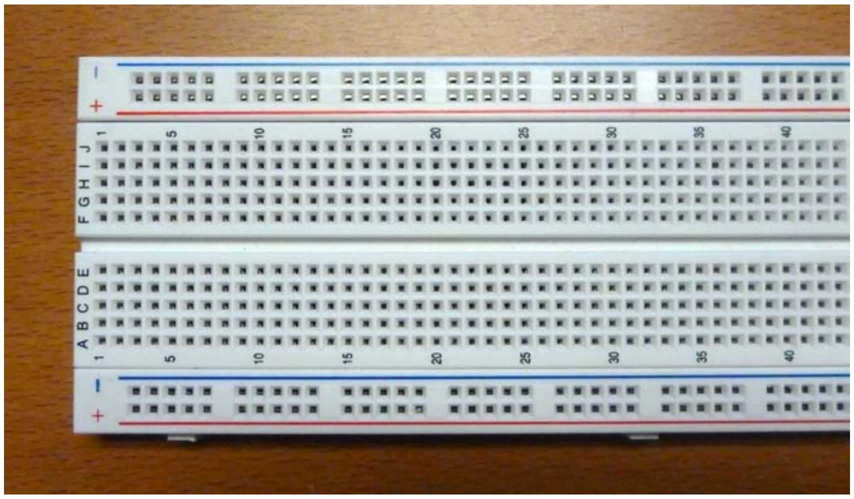


Figure 1.5: A solderless breadboard. The holes in the breadboard are on 0.1in centers so they fit standard integrated circuit pins that also have that spacing. To make connections, 22AWG solid core wire can be inserted into the holes to make connections.

entire column is one electrical node. These columns are typically (but not always) used for power and ground connections, so they're marked with + and - in case you want to use them for that purpose.

In this board, shown in Figure 1.5, the five holes in a row marked with abcde are also connected as a node. That is, any wire plugged into row 5, column a will be connected to any other wire in columns b, c, d, or e of that row. Likewise, the f, g, h, i, and j holes are also connected in a row. There is no connection across the valley in the middle of the board. This is designed so that integrated circuit packages (chips) can be placed in the board with legs on either side of the valley. All sorts of other components can also be placed in these breadboards.

As a very simple example, Figure 1.7 shows the same circuit we saw before with the resistor and LED. In the physical version of this circuit there is a green wire connecting the GND connection of the board on the left (an Arduino board - we'll see that in a minute) to the blue bus column. The red LED has one leg in the blue bus column, and the other leg in row 5 column A. The resistor has one leg in row 5, column E (and is thus connected to the LED leg), and the other leg in row 8, column A. Also connected to row 8 (column A) is a purple wire going back to the blue board into the connection labeled 10. Although it doesn't look identical, this is a correct physical realization of the schematic using a solderless breadboard.

We'll see further examples of how breadboards are used to connect components as the course progresses. For now, there's another Big Idea:

When you connect multiple devices together, all the ground terminals of all the devices MUST be connected together. Ground is the zero-volt reference for your system. If you don't connect them all together, then they might all have different notions of zero volts. That's a very bad thing! Notice in the preceding extremely simple example, the GND connection of the board on the left is connected to a GND bus in the breadboard on the right. Nice!

Big Idea #7

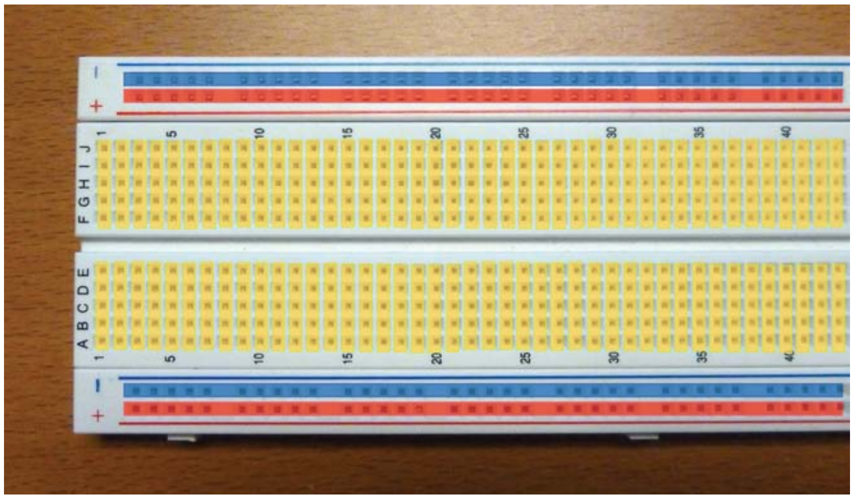


Figure 1.6: A solderless breadboard annotated to show what sets of pins are connected together internally to the breadboard. The holes in the rows at top and bottom are connected into buses. Every hole under the red and blue lines are connected internally. The sets of five holes in yellow are also connected. Wires or pins put into any of the holes covered by a yellow square will be connected together.

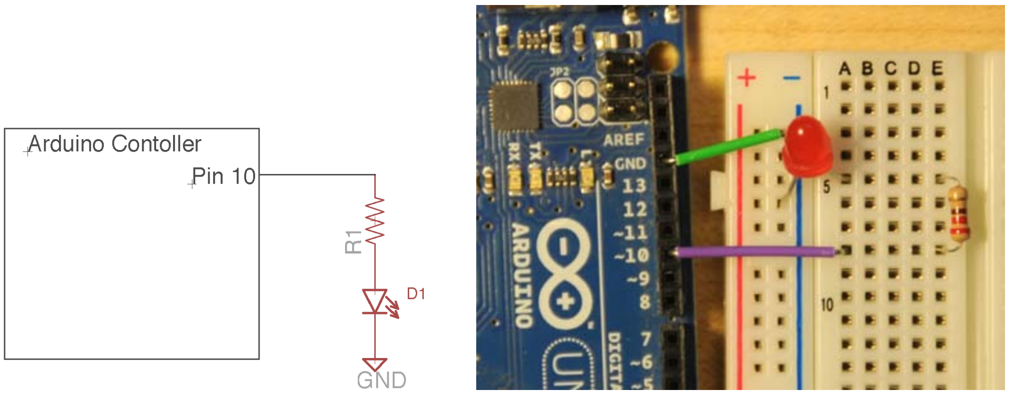


Figure 1.7: A schematic on the left, and the physical realization of that schematic on the right using a solderless breadboard. The purple wire goes from pin 10 of the Arduino to row 8 of the breadboard. The resistor connects row 8 to row 5. The LED is connected from row5 to the GND bus, and finally the GND bus is connected to the GND pin on the Arduino with the green wire.

1.5 The Arduino Platform

When we get to the part about interfacing electrical components to a computer controller, we need a computer controller to connect to. There are lots of choices here. You could use your desktop computer or your laptop, although those aren't really optimized for attaching physical computing components like LEDs, motors, and sensors. You could use a micro-controller chip directly - there are hundreds to choose from: ARM, Atmel AVR, Freescale 68HC11, Intel 80C51, MIPS, PowerPC, Microchip PIC, or TI MSP430 just to name a very few. But, that's a pretty complex endeavor - these chips aren't at all trivial to connect in a system, and it can be tricky to figure out how to upload programs once you have designed a system around them.

The easiest solution is to find an embedded computing board that is already designed for you, along with programming support and data uploading and downloading infrastructure. There are quite a few of these boards available. One popular option is the Arduino. The name *Arduino* corresponds to (at least) two different things: A small embedded controller board based on an AVR 8-bit micro (currently the ATmega328 on the common Duemilanove and Uno boards), and a programming environment (a somewhat basic Integrated Development Environment (IDE)). The IDE is designed to support the controller board and is based on gcc (actually a version of gcc called avr-g++). The Arduino web site introduces things in this way:

> Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It's intended for artists, designers, hobbyists, and anyone interested in creating interactive objects or environments.

Both parts of Arduino come originally from a group of folks in Italy led by Massimo Banzi but has since made the transition to a web-based open-source project (both open source SW and HW) with the main Arduino team scattered all over the world. The term "open source hardware" means that all the hardware design files (schematics and board layouts) are available free of charge so that users can study, understand, and modify the hardware to their own liking.

The main web site for all things Arduino is `www.arduino.cc`. It is from this site that you can download the official Arduino IDE, and where all the officially supported software packages are available. It's also where some of the text in this document was borrowed from. It's definitely the first stop when looking for Arduino information. There is also an unofficial site with even more web-community-supplied Arduino goodies at `www.freeduino.org` (titled "The World Famous Index of Arduino & Freeduino Knowledge"). I'll include a more comprehensive list of Arduino-friendly web sites a little later. An Arduino assembled from raw parts by hand will cost around $6. You can also easily buy pre-assembled boards for a reasonable price (around $30). One example is shown in Figure 1.8. This is that standard Arduino form factor and is about the size of a credit card.

Other form factors include tiny Arduinos that fit in a breadboard (Arduino Nano and Arduino Mini), to really really tiny versions that are basically a "backpack" for the ATmega328 chip (Ardweeny), to "mega" versions that use a beefier version of the ATmega chip (the ATmega2560) that has twice the memory and lots more I/O pins, and even a version called the LilyPad which is designed to be sewn into clothing for wearable applications. You can see an overview of the different "official" boards at `arduino.cc/en/Main/Hardware`, and more of the user-designed boards at `arduino.cc/playground/Main/SimilarBoards`.

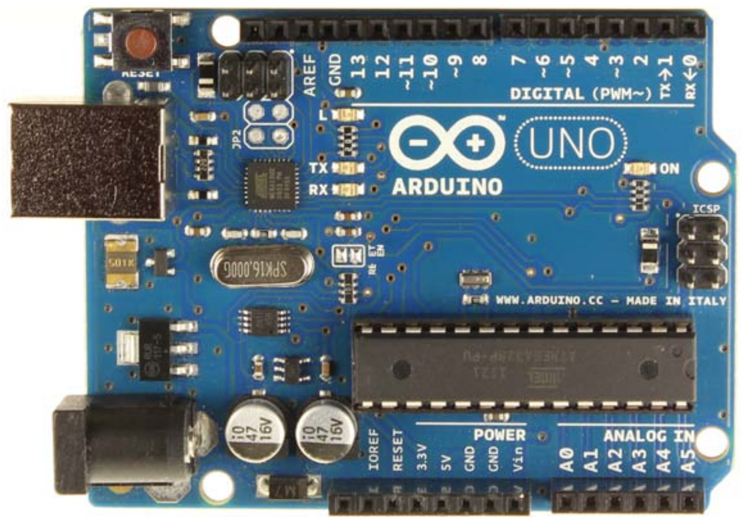


Figure 1.8: An Arduino Uno embedded controller board.

The Arduino comes in a variety of versions, but they're all very similar in capability, and they all work with the Arduino IDE. On the Arduino board shown in Figure 1.8. The black connectors at the top of the picture are for 14 digital input/output pins (numbered 0 through 13). These pins can be used either as inputs (e.g. for sensing the voltage on a switch) or for outputs (e.g. for driving 5v or 0v to an LED) On the bottom of the picture are connectors for power (5v and 3.3v), Ground, and six analog inputs (numbered A0 through A5). Analog inputs are for sampling continuous voltages between 5v and 0v. We'll see in the sensor section how useful these are for environmental sensing. The silver box on the top left of the picture is the USB connection, used both to power the Arduino and to upload/download data. The black connector on the bottom left is an auxiliary power connector, and the large black chip is the microcontroller itself. This course isn't really about using the Arduino, although we will use it for controller examples

The best thing about Arduino is that it's designed to be extremely simple to use with physical computing components. LEDs (Chapter 2) and servo controls (Chapter 4) can be connected directly to the digital I/O pins. Larger motors can be controlled through the digital I/O and a switching transistor (see Section 4). Environmental sensors can be connected easily through the analog inputs (Chapter 3). Programs can be written in C/C++ using the Arduino IDE and uploaded from your PC/Mac or laptop using a USB cable. All in all a very spiffy little system!

1.5.1 Arduino Electrical Overview

These specifications may not make complete sense at the moment, but they should be a useful reference for later. You can also find information about specific Arduino boards (all of which may have slightly different specs) on the Arduino web site http://www.Arduino.cc. This overview is for the main generic Arduino boards that you're likely to see (e.g. the Duemilanove and Uno).

Power to the Arduino board: You can supply power to your Arduino board either through the USB cable that you use for uploading code, or through a separate power supply using a 2.1mm center-positive male connector. If you use the separate power supply you can use anything in the range of 7-12 volts, although 9v is usually considered the optimal power supply. Higher than 9v causes the voltage regulator on the Arduino to get warm (or hot!).

Note that you can also use a 9v battery to power the Arduino. You can easily find 9v battery connectors (snaps) that have a 2.1mm center-positive male connector that mates with the power input of the Arduino.

Voltage on digital pins: Each of the 14 digital pins on the Arduino can be configured to be an output where the Arduino is driving the voltage on the pins, or an input where an external circuit is driving the voltage on the pins. In either case the range of acceptable voltages is 0v to 5v.

Digital pin usage restrictions: Of the 14 digital pins, a few have potential restrictions or special uses:

- Pins 0 and 1 are also used for UART communication to the host so it's good practice to avoid using them for general circuit connections unless you need them. If you do need to use them for general I/O, it's sometimes necessary to disconnect them during program upload, and then connect the wires again after the program has been uploaded. You should not use them at all if your own program is using the Serial communication capability.
- Pins 2 and 3 can also be used for external interrupts. This is a bit of an advanced feature that we won't cover, and you don't have to worry about using these as general I/O pins if you're not using the interrupt feature.
- Pins 3, 5, 6, 9, 10, and 11 can be configured to provide 8-bit pulse width modulation (PWM) using the `analogWrite()` function. If you're not using this function, you can use them as general I/O pins.
- Pins 10, 11, 12, and 13 are used for hardware-controlled SPI communication (Serial Peripheral Interface) using the SPI Library. If you're not using this library, or communicating to external devices using this protocol, you can use them as general I/O. As a side note, this restriction is only if you're using the hardware-supported SPI. There is also a software-only SPI library that uses any of the 14 digital pins.
- Pin 13 has a built-in LED connected to it. Whenever you drive pin 13 high the LED will be lit, and when you drive pin 13 low, it will be off.

Current through digital pins: The digital pins on the Arduino can provide or receive a max of 40mA/pin. Providing or receiving current to a high voltage is known as sourcing the current, and providing/receiving current to ground is known as sinking the current. Additionally there is a limit of 200mA max total among all 14 pins. This means, for example, that you can't simultaneously sink 40mA from all 14 pins!

Voltage on analog inputs: The analog to digital converter (ADC) that is receiving the analog inputs on pins A0 through A5 can handle voltages in the range of 0v to 5v. The Arduino's ADC has a resolution of 10 bits which means that the voltage range of 0v to 5v is mapped to digital values from 0 to 1023.

Analog pin usage restrictions: Analog pins A4 and A5 are used for I^2C (also known as the two-wire interface (TWI)) communication with external devices using the Wire library. If you're not using this communication library, or communicating to external devices using this protocol, you can use them as general analog inputs.

Power for external devices: The Arduino can pass through the power signal for use with external devices through 5v and 3.3v pins. These power signals come from on-board voltage regulators on the Arduino. If you're using the USB connection for power there is a max of 500mA total for the entire system including the Arduino and external devices (limited by the USB specification). If you're using the 2.1mm power connector and a 7-12v power adaptor the max power around 800mA (perhaps a little more if you attach a heat sink to the regulator). The 3.3v connection provides a max of 50mA. This is smaller because of the limit on the separate voltage regulator used to generate the 3.3v signal.

Arduino Memory: The ATmega328p that is the microcontroller on the Duemilanove and Uno boards is an 8-bit processor, but the C/C++ compiler handles the use of larger data types when it generates code. This microprocessor has 32KB of flash memory for storing code, of which 2KB is used by the boot loader (0.5KB on the Uno). This means that your code's binary (produced by the Arduino IDE compiler) must fit in the remaining Flash memory space (30KB on Duemilnove, 31.5KB on Uno). It also means that because the code is in flash, once you upload your code to the Arduino it stays there even when you cycle power. The ATmega328p also has 2KB of SRAM (used for variables by the compiler) and 1KB of EEPROM (accessible using the EEPROM library).

1.5.2 Arduino Software Overview

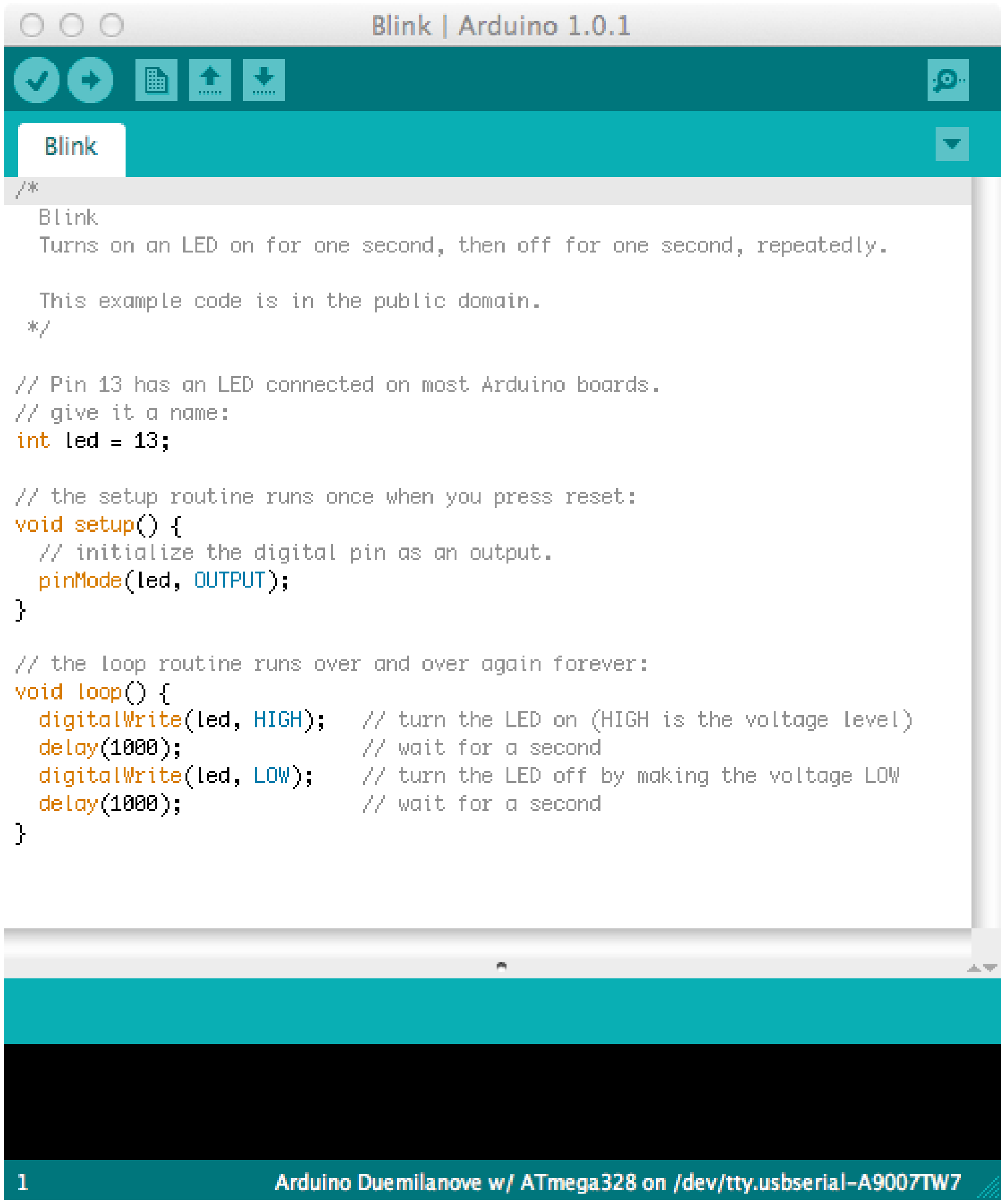
The Arduino software environment consists of two main parts: an integrated development environment (IDE), and a set of built-in functions and libraries that allow the user to easily write C and C++ programs that interact with the I/O ports on the Arduino hardware. The Arduino integrated development environment (IDE) 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. An image of the Arduino IDE interface, with a very simple example program, is shown in Figure 1.9. The text editor uses syntax coloring to help identify parts of your program syntax. As you can see in the figure, C and C++ keywords (including the built-in functions added by Arduino) are colored orange. Constants that are defined in the included header files are colored blue. Other program text is black. The IDE is free and available on the `www.arduino.cc` web site. It runs on Windows, Mac, and Linux.

One note: in Arduino-speak programs are known as *sketches.* This means that your sketches go in your *sketchbook* folder when you save them. I'm not fond of this terminology so I'll continue to call them programs. This section is a non-comprehensive overview of the Arduino software environment. For a more complete reference see the Arduino web site, or one of the many books available on Arduino programming. These notes will include code listings as the electrical components are introduced.

The control buttons on the Arduino IDE are the following:

▷ **Verify/Compile –** This runs the gcc compiler on your program and returns any errors or warnings in the status area at the bottom of the window.

□ **Stop –** Stops the serial monitor (a text window opened on the host), or unhighlights other buttons.



```
/*
  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
}
```

```
1                    Arduino Duemilanove w/ ATmega328 on /dev/tty.usbserial-A9007TW7
```

Figure 1.9: The Adruino Integrated Development Environment (IDE). The program shown is Blink. This is the program that is installed in a newly purchased Arduino board by default. It blinks the built-in LED on pin 13 of the board at 1Hz (one blink per second). Note the use of the two required functions setup() and loop().

New – Creates a new blank program (sketch) in the IDE editor.

Open – Presents a menu of all the sketches in your sketchbook (all the programs in your local program directory). Clicking one will open it in the current window. On some versions os the Arduino IDE (due to a Java bug), this menu will not scroll. So, if you need to open a program that is late in the list, use the **File** → **Sketchbook** menu instead.

Save – Save your program. The default location (sketchbook) used to save programs is set in the **preferences** under the **Arduino** menu.

Upload to your Arduino board – This button compiles your code and uploads it to your Arduino board through the USB interface.

Serial Monitor – This opens the serial monitor on the host. This is a text-based window that can be printed to using Arduino functions **print(value)** and **println(value)**, and that can be used to send ASCII data back to the Arduino.

Additional commands are found in the five menus: **File**, **Edit**, **Sketch**, **Tools**, and **Help**. These are documented in the Arduino IDE and on the Arduino web site. Some of the more important menu choices are:

File **Sketchbook** This opens a menu into your local program directory so that you can select one of your previously saved programs (sketches).

Examples This contains the example programs that come with the IDE. These can be very useful to see how various libraries are used.

Edit **Copy** Copy program text for pasting into another program

Copy for Forum Copy in a format suitable for posting to the Arduino form including syntax coloring

Copy as HTML Copy in HTML format suitable for embedding in web pages.

Sketch **Show Sketch Folder** Open a window to the folder that contains the program (sketch) you're currently working on.

Import Library Add the appropriate **#include**< ... > lines for libraries that are in your path.

Tools **Board** Select which Arduino board you're going to compile for, and upload to.

Port Select which USB port you will be using to upload the program to the Arduino board.

Help These menu choices open a browser onto the arduino.cc page that has information about the listed topic. The **Reference** topic, for example, opens the page that has the Arduino built-in function reference.

The standard programming environment for Arduino is C/C++ through the Arduino IDE and avr-g++. The basic C++ language is augmented with a set of functions that are built in to the Arduino IDE.

Arduino application structure: Arduino requires that all programs (sketches) consist of two top-level functions. These functions can be seen in Figure 1.9 that shows a very simple Arduino program. The functions are:

- `void setup()` - called exactly once to set things up. This function is typically used to set the direction of digital pins (input or output), and initialize libraries if needed. This function never returns a value so it is always declared as returning void.
- `void loop()` - runs and loops forever after `setup()` finishes. This is a fundamental part of the reactive nature of a typical Arduino application. These applications either repeat their behavior over and over, or repeatedly wait in a forever-loop for some external event and then react to that event. The external event is often waiting for some value on the external pins, either digital or analog. This function also never returns a value. In fact, it never returns at all because its behavior is to be an endless loop. It is also always declared to return void.

  An exception (no pun intended) to putting the program behavior in the `loop()` function is if you're using interrupts. If the program activity is all handled in the interrupt service routines, it's possible to have an empty `loop()` function, but you still have to have that empty function in your program.

Arduino data types: These are basically C/C++ data types, but remember that the C standard is purposely vague on certain things, like the bit width of an int. The basic data types are listed below for the Duemilinove and Uno boards, but there are more (e.g. unsigned types).

- `boolean` (1 bit - true/false)
- `char`(signed, 8-bits)
- `byte` (unsigned, 8-bits)
- `int` (signed, 16 bits)
- `long` (signed, 32 bits)
- `float` (32 bit fp)
- `double` (also 32 bit fp)

Arduino functions for digital pins: These functions all operate on any of the 14 digital pins (numbered 0 through 13). Recall that C/C++ is case sensitive, so be careful with case in the function names.

- `pinMode(pin_number, mode)`
  - Sets the direction of that digital pin
  - mode can be `INPUT`, `OUTPUT`, or `INPUT_PULLUP`
  - If mode is `INPUT_PULLUP` an internal pullup resistor is enabled on the input. More on this later in Chapter 3.
  - Typically used within the `setup()` function.
- `digitalRead(pin_number)`
  - Reads the voltage currently on that digital pin (pin should be configured to `INPUT` mode)
  - Returns `HIGH` (1) if the voltage on the pin is around 3v or greater (up to 5v)
  - Returns `LOW` (0) if the voltage on the pin is around 2v or lower

- `digitalWrite(pin_number, value)`
  - value can be `HIGH` (1) in which case the pin is forced to 5v, or or `LOW` (0) in which case the pin is forced to 0v (pin should be configured to `OUTPUT` mode)
- `analogWrite(pin_number, value)`
  - This function is only meaningful on digital pins 3, 5, 6, 9, 10, and 11 on the Duemilanove and Uno boards
  - value is a number between 0 (fully off) and 255 (fully on).
  - A number between 0 and 255 will vary the duty cycle of the pulse width modulation on the pin which acts as a partial on value in a variety of cases

Arduino functions for analog pins: these pins are connected through a 10-bit analog to digital converter (ADC) so that analog voltages between 0v and 5v on the pins will return an int between 0 and 1023.

- `analogRead(pin_number)`
  - Analog pins may be referred to as pin A0 through A5, and it's good practice to do so to differentiate them from the digital pins

Arduino functions for delays and timing: These functions are used to deal with timing in Arduino programs.

- `millis()`
  - Returns the number of milliseconds (thousands of seconds) since the program started as an unsigned long
  - This number overflows (goes back to 0) after about 50 days of running
- `micros()`
  - Returns the number of microseconds (millionths of a second) since the program started running as an unsigned long
  - This number overflows after about 70 minutes
  - On a 16MHz processor like the Duemilanove and Uno this has a resolution of 4 microseconds (i.e. the number is always a multiple of four)
- `delay(ms)`
  - Delay the program by busy-looping for ms milliseconds
  - Argument can be up to an unsigned long
- `delayMicroseconds(us)`
  - Delay the program by busy-looping for us microseconds.
  - Argument is an int, and the largest value for which an accurate delay can be produced is 16383
  - Accuracy isn't guaranteed for arguments of 3 or less

Arduino functions for communication to host: The Arduino has a UART on-board that it uses to communicate with the host. This UART is used for uploading the program code, and also for general communication with the host once the program is running. The Arduino IDE has a built-in terminal that can be used to display data from the Arduino, and for the user to send characters from the host to the Arduino.

Note that it looks like you've connected your Arduino to the host with a USB cable, and you have. But, the Arduino is actually communicating with a serial protocol layered on top of the USB connection. From the Arduino programmer's point of view, it's just a serial connection.

- `Serial` object - The Serial object is pre-defined in the Arduino IDE, and all the functions used for serial communication are methods called on that object. There are many more methods than the ones described here.
- `Serial.begin(speed)`
  - This is the baud rate for the serial communication.
  - Legal values are 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200.
  - Make sure the terminal you're connecting to (e.g. the serial monitor in the Arduino IDE) is set to the same baud rate
  - This is typically called in `setup()`
- `Serial.print(arg)`
  - Prints data to the serial port as human-readable ASCII text. This command can take many forms.
  - Numbers are printed using an ASCII character for each digit.
  - Floats are similarly printed as ASCII digits, defaulting to two decimal places.
  - Bytes are sent as a single character.
  - Characters and strings are sent as is.
  - An optional second parameter specifies the base (format) to use;
  - permitted values are `BIN`, `OCT`, `DEC`, and `HEX`
  - For floating point numbers, this parameter specifies the number of decimal places to use.
- `Serial.println(arg)`
  - Same as `Serial.print(arg)`, but follows the printed data with a carriage return character (ASCII 13, or '\r') and a newline character (ASCII 10, or '\n')
- `Serial.available()`
  - Returns the number of bytes (characters) available for reading from the serial port.
  - This is data that's already arrived and stored in the serial receive buffer (which holds 64 bytes)
- `Serial.read()`
  - Returns the first byte of incoming serial data from the serial receive buffer (as an int)
  - Returns -1 if no data is available

Arduino helper functions: There are quite a few here, but I'll just mention a few that we'll see in the next sections.

- `random(min, max)`
  - Returns a pseudo-random number between min and max-1
- `map(value, fromLo, fromHi, toLo, toHi)`
  - Remaps (interpolates) the value argument from the range (fromLo, fromHi) to the range (toLo, toHi)
  - Does NOT constrain the value to be within the to range
  - The lower bounds of either range can be higher or lower than the other bounds. You can use that to reverse the range of a number.

  - Uses integer math so it will not generate fractional values
- `constrain(value, a, b)`
  - Constrains the value argument to be between a and b (inclusive)
- `shiftOut(data_pin, clock_pin, bit_order, value)`
  - Sends 8-bits of data on a serial (SPI) link.
  - data_pin: the pin on which to output each bit (int)
  - clock_pin: the pin to toggle once the data_pin has been set to the correct value (int)
  - bit_order: which order to shift out the bits; either `MSBFIRST` or `LSBFIRST`. (Most Significant Bit First, or, Least Significant Bit First)
  - value: the data to shift out. (byte)

Chapter 2

Lights! LEDs

There are a lot of different components you could use to generate light, but LEDs are a great choice because they're cheap, and they use very little power to generate light. An incandescent light bulb, for example, takes much more current than an Arduino pin can provide, but an Arduino digital output pin can easily provide enough current to light up an LED. Plus they come in a wide variety of shapes, sizes, and colors. They're a great way to add some flash to your projects.

LED stands for Light Emitting Diode. They are an example of a more general type of semiconductor device called simply a **diode**. A diode is essentially a one-way valve for current. That is, current can flow one direction through the diode, but not the other. They're used in a variety of situations where the circuit designer wants to make sure that current is only flowing in one direction and never backing up in the other direction.

Big Idea # 8

The two terminals of a diode (light emitting, or not) are the anode and the cathode. Current can flow from anode to cathode, but not the other way, except in certain exceptional situations.

Because they are directional components, there should be some way of identifying which terminal is which in a schematic. The schematic symbol for a diode looks like the left hand side of Figure 2.1. The anode (input) is on the left, and cathode (output) is on the right with the vertical bar. Current can flow in the direction of the triangle, and not back through the bar. Physically, general diodes typically look a bit like resistors - they are little cylinders with wires attached to the ends. Instead of colored bands to indicate value, diodes typically have tiny numbers printed on them. There is also usually a single band on the cylinder that indicates the cathode end. Think of the band on the diode as marking the position of the bar on the diode symbol.

An LED is a diode, so it behaves like a regular diode with respect to current flow. But, it has the added feature that when current does flow from anode to cathode, it lights up. So, the schematic symbol looks like a diode, but adds an indication of the light-emitting feature. It looks like the right hand side of Figure 2.1.

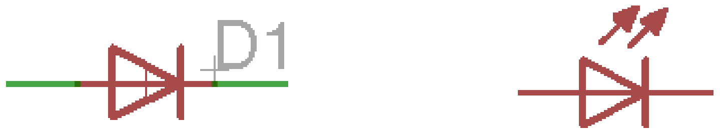


Figure 2.1: Schematic symbols for a general diode and a light emitting diode (LED).

Physically, LEDs come in a wide variety of shapes, sizes, and colors. One important consideration is how to figure out which lead is the anode, and which is the cathode. In a standard through-hole LED (i.e. one that is designed to be inserted through the holes in a circuit board), one lead will be longer than the other. *The longer lead is the anode, and the shorter is the cathode.* The good news is that unless you apply a very large voltage to the leads of the LED you can't really hurt it by putting it in a circuit backwards. That's what diodes are designed for after all, to block current in the reverse direction. So, if you don't remember which lead is which, or if you've cut the leads of the LED to the same length, you can just try it both ways and see which way lights up.

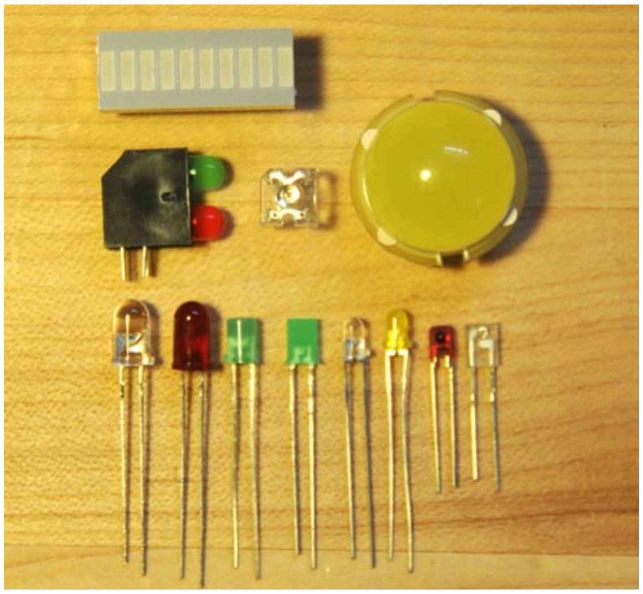


Figure 2.2: A variety of different types of LEDs.

An important electrical consideration for any diode, LEDs included, is the *forward voltage.* A diode does indeed conduct from anode to cathode, but only after the voltage difference from anode to cathode is raised above a certain voltage, known as the forward voltage, or V_f. For many regular diodes, the forward voltage is 0.7v. For an LED, the forward voltage is higher: often between 1.5v and 3.5v. Red LEDs typically have the lowest forward voltage (1.7v is a fairly typical value), and in general the forward voltage goes up as the frequency of the emitted light goes up. So, V_f goes up as LEDs move from up the spectrum (red, orange, yellow, green, blue) with blue LEDs having a V_f of 3.0 to 3.5v. Of course, the best plan is to get the specs for the LEDs you plan to use.

Big Idea # 9

One ramification of the forward voltage is that the LED doesn't conduct until the voltage difference between anode and cathode gets above that value. Another is that because of that behavior, the diode uses up that much voltage once it does conduct. That is, if you put 5v between anode and cathode of an LED with a 2.0v Vf, the voltage at the cathode will be 3v with respect to ground. Said in another way, the LED has a 2.0v voltage drop across the diode.

Another important consideration with respect to diodes is how much current the diode can support. Regular diodes come in a huge variety of current capacities from milliamps to hundreds or thousands of amps. Regular LEDs, on the other hand, are almost always designed to produce maximum light output at around 20mA, and look good when the current is between 10-20mA. You can get super-bright high-powered LEDs that are designed for 800mA or 1000mA (i.e. up to 1A), but the generic inexpensive 3mm or 5mm LEDs that you're likely to encounter will almost certainly be designed for around 20mA.

One question that is critical for using LEDs is how to make sure that the current you're using to drive the LED is limited to the right value. This is a perfect application for Ohm's law from Chapter 1. If we know the total voltage we're putting through the LED, the forward voltage V_f, and the desired current, we can use Ohm's law to compute the correct resistor value for that circuit.

Example: If the Arduino digital output pin drives to 5v, and the LED has a V_f of 2.0v, and a current limit of 20mA, then we can compute the resistor required to make this all work. Remember that the V_f is subtracted from the total voltage (used up by the LED), and 20mA is 0.020A. Ohm's law uses whole-unit values for Ohms, Volts, and Amps.

$R = V_{total}/I$
$R = (V_{source} - V_f)/I$
$R = (5.0v - 2.0v)/0.020A$
$R = 150\Omega$

For this example, you would put a 150Ω resistor in series with the LED to limit the current to the 20mA as shown in Figure 2.3.

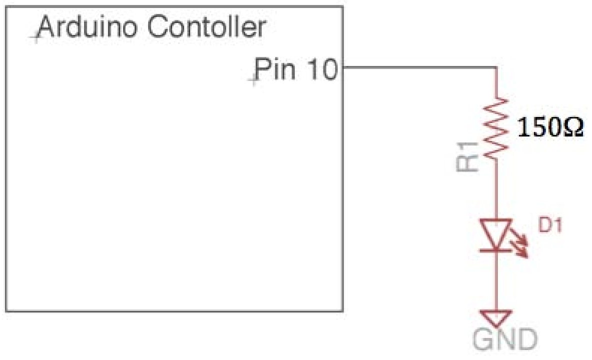


Figure 2.3: Simple LED schematic with current-limiting resistor specified as 150Ω.

Big Idea # 10

LEDs always need a current-limiting resistor. Never connect an LED without computing what the current limiting resistor should be. The only exception to this rule is for special LED drivers that have constant current outputs (we'll see some in the next sections). The value of the resistor can be computed using Ohm's law and the forward voltage of the LED using the equation $R = (V_{source} - V_f)/I_{desired}$ *. Most generic LEDs work well with between 10mA to 20mA of current. When in doubt, a resistor in the range of 220Ω to 470Ω will usually do the trick and be safe.*

As an aside, those odd resistance values represent *standard* resistance values. You can special-order resistors in any ohm rating that exists, but there are a set of values that are considered standard and that are readily available at any electronics shop. Some standard values that are useful for typical current-limiting applications with standard LEDs are 150, 180, 220, 330, and 470Ω.

Recall from Section 1.5.1 that each pin of the Arduino can supply up to 40mA. This means that each of the 14 digital pins can light up an LED, and you can control the on/off of the LEDs using `digitalWrite()` functions. Remember, though, that the max current for the entire Arduino is 200mA. So, if you have 10 LEDs and you will be lighting them up all at once, you'll need to limit the current to 20mA or less per LED to avoid hitting that max current limit!

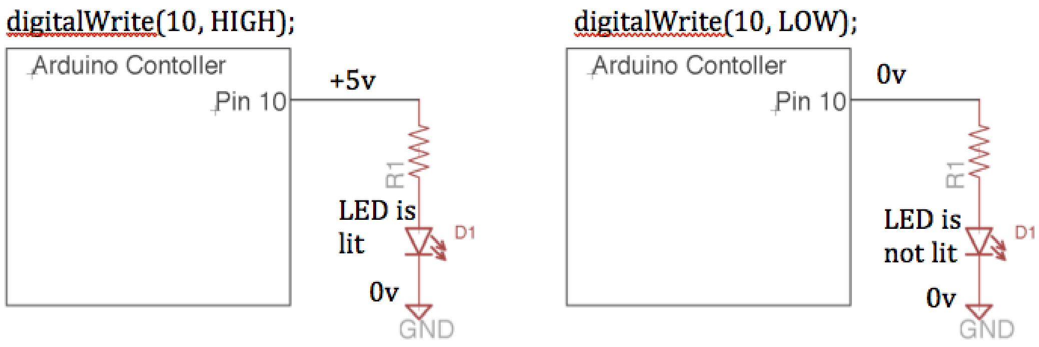


Figure 2.4: Driving an LED from an Arduino pin using digitalWrite(). If the pin is driven high, the LED's anode voltage is higher than the cathode, so current flows and the LED lights up. If the pin is driven LOW, the anode and cathode are at the same voltage (ground) so no current flows and the LED is not lit. Remember that the voltage difference must be greater than the forward voltage (V_f) in order for current to flow. V_f is typically between 1.5v and 3.5v depending on the LED. In this example, the voltage difference is +5v, which is plenty.

Figure 1.9 shows a screen shot of the Arduino IDE with the blink program. This program, the default program loaded in a new Arduino board, sets up digital pin 13 as an output in the `setup()` function, and then flashes the LED connected to that pin once per second. The `digitalWrite()` functions set the pin to a high or low voltage, and the `delay()` function spin-waits and delays the action by the given length of time (1000ms, or 1sec). Pin 13 is a special digital pin in that it has an LED (with the required current-limiting resistor) built into the board, so no external wiring is needed to run this program and see the flashing LED output.

If you wanted to repeat this program, but with a different pin, you would need to wire up an external LED and resistor and connect them to another of the Arduino's pins. This is what is shown in Figure 1.7 with Arduino digital pin 10. The only difference required to the blink program in Figure 1.9 is to change the value of the led variable to 10.

2.1 Multiple LEDs driven from digital pins

One LED is fine as a starting point, but what about multiple LEDs? You could connect a different LED, each with its own current-limiting resistor, to each of the 14 digital pins. You should make sure that you understand the amount of current each LED is consuming so that you don't approach the 200mA Arduino current limit when all the LEDs are lit. You can do this by choosing your current-limiting resistor carefully. The picture in Figure 2.5 shows an Arduino with eight LEDs and eight resistors connected to eight of the Arduino's pins. The resistors are chosen so that each LED is consuming approximately 13mA of current at 5v (the resistors in the picture have bands of brown, red, and red making them 220Ω resistors). So, even if all eight are turned on at one time, the total current will be only 96mA.

There are many ways you could write an Arduino program to light up these LEDs. The easiest would be just to write individual `digitalWrite()` commands for each LED.

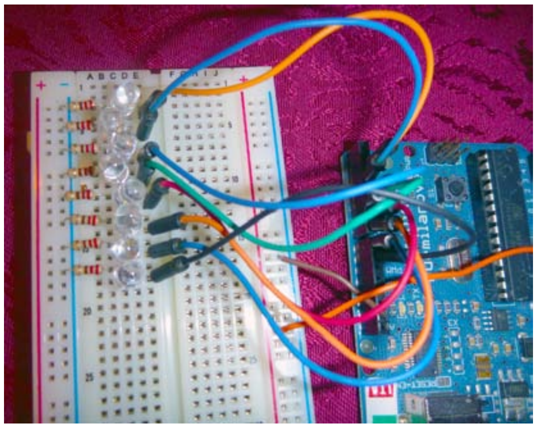


Figure 2.5: An Arduino board connected to eight LEDs, each with a current-limiting resistor. The left terminals of the resistors are connected to the blue ground bus. Out of the frame of this picture, the ground bus is connected to a GND pin on the Arduino. Each LED will come on when the corresponding digital pin is pulled HIGH on the Arduino.

That program would look something like this (additional Arduino example programs are included in the auxiliary materials for this course):

```
/*
 * Eight-LED example #1
 *
 * This example takes a brute-force approach to making patterns
 * appear on the 8 LEDs.
 *
 * Note that the pins used for the LEDs are 2,3,4,5,6,7,8,9.
 * This is because pins 0 and 1 are also shared with the
 * communication pins that allow Arduino to talk with the PC.
 * You could still use them, but you might have to disconnect
 * them during programming, and then connect them again.
 *
 * setup() is the function that runs once to set things up. We'll
 * use it to set all of the LED pins to be OUTPUTs
 */
void setup() {
   pinMode(2, OUTPUT);    // Define all 8 pins as outputs...
   pinMode(3, OUTPUT);
   pinMode(4, OUTPUT);
   pinMode(5, OUTPUT);
   pinMode(6, OUTPUT);
   pinMode(7, OUTPUT);
   pinMode(8, OUTPUT);
   pinMode(9, OUTPUT);
   digitalWrite(2, HIGH); // Start with the LED on pin2 HIGH (ON)
}

/* loop() is the function that repeats the Arduino's behavior over and
 * over when the power is on.
 */
void loop() {
  int delayMS = 100; // Variable to hold the time (in milliseconds) to
                     // delay between parts of the LED pattern.
   // This set of digitalWrite() commands makes (half of) the
   // "Cylon eye" behavior.
   // Start by going from pin2 LED to pin9 LED.
   // Note that the first time through loop(), pin2 is already HIGH
   delay(delayMS);          // Wait for delayMS milliseconds
   digitalWrite(2, LOW);    // set pin 2 LED OFF
   digitalWrite(3, HIGH);   // set pin 3 LED ON
   delay(delayMS);          // Wait for delayMS milliseconds
   digitalWrite(3, LOW);    // set pin 3 LED OFF
   digitalWrite(4, HIGH);   // set pin 4 LED ON
   delay(delayMS);          // Wait for delayMS milliseconds
   digitalWrite(4, LOW);    // etc
   digitalWrite(5, HIGH);
   delay(delayMS);
   digitalWrite(5, LOW);
   digitalWrite(6, HIGH);
   delay(delayMS);
   digitalWrite(6, LOW);
   digitalWrite(7, HIGH);
   delay(delayMS);
   digitalWrite(7, LOW);
   digitalWrite(8, HIGH);
   delay(delayMS);
   digitalWrite(8, LOW);
   digitalWrite(9, HIGH);
   delay(delayMS);
   // For the full cylon eye you would need to light up the LEDs
   // in reverse order here
   // end of loop() - go back and start again
}
```

A slightly fancier version of this program makes use of a separate function to set the state of all eight LEDs based on the value expressed as an 8-bit byte.

```
/*
 *
 * This example uses a separate function to set the LED outputs. This
 * new function can be called by the user each time the LED should be
 * set.
 * This example uses a relatively advanced low-level technique. It
 * codes the LED values in a single byte with each bit of the byte
 * being the 1 or 0 that determines the LED ON/OFF state. This
 * requires the setLEDs() function to pick off each bit of the byte in
 * turn.
 *
 */

/*
 * Define the array to hold the LED pin numbers. It's defined here
 * outside all the functions so that it's a "global" variable
 * and can be seen by all functions.
 */
int ledPins[] = {2,3,4,5,6,7,8,9}; // An array to hold the Arduino pin
                                   // numbers that each LED is
                                   // connected to.

/*
 * setup() is the function that runs once to set things up.
 */
void setup() {
  // Define all 8 pins as outputs...
  // The pins are referenced through the ledPins array
  for (int i = 0; i<8; i++){       // loop eight times
    pinMode(ledPins[i], OUTPUT);   // set each LED pin to OUTPUT
  }
} // end of setup()
```

```
/* loop() is the function that repeats the Arduino's behavior forever
 * while the power is on.
 */
void loop() {
  int delayMS = 100;  // Variable to hold the time (in milliseconds)
                      // to delay between parts of the LED pattern.
  // This set of setLED() commands makes the "Cylon eye" behavior
  //
  // This version takes a single byte as an argument to tell setLEDs
  // what the ON/OFF state is in each step of the pattern. The Arduino
  // syntax for this is B10101010 for a single byte value.
  // There's a subtlety here - B00000001 means that bit 0 of the byte
  // is 1.
  setLEDs(B00000001, delayMS);
  setLEDs(B00000010, delayMS);
  setLEDs(B00000100, delayMS);
  setLEDs(B00001000, delayMS);
  setLEDs(B00010000, delayMS);
  setLEDs(B00100000, delayMS);
  setLEDs(B01000000, delayMS);
  setLEDs(B10000000, delayMS);
  setLEDs(B01000000, delayMS);
  setLEDs(B00100000, delayMS);
  setLEDs(B00010000, delayMS);
  setLEDs(B00001000, delayMS);
  setLEDs(B00000100, delayMS);
  setLEDs(B00000010, delayMS);
  setLEDs(B00000001, delayMS);

  // This is the end of the "cylon" pattern. Remember that loop()
  // starts over after it's done, so the pattern repeats.
} // end of loop() - go back and start again

// This is the function that actually applies the pattern to the LED
// and then delays for the specified amount of time.
// This version takes a single byte as input that holds the ON/OFF
// values for the LEDs. It loops through each bit of that byte to set
// the LED values. The LED pin numbers are held in the global LEDPins // array.
void setLEDs(byte LEDvalues, int delayMS) {
  // You can access each bit of the LEDvalues byte using the
  // Arduino syntax: bitRead(number, whichBit);
  for(int i=0; i<8; i++) { // loop 8 times - i = 0,1,2,3,4,5,6,7
     digitalWrite(ledPins[i], bitRead(LEDvalues, i));
  }

  // Now wait for delayMS milliseconds so you can see the change
  delay(delayMS);
} // End of setLEDs()
```

2.2 Diming an LED with pulse width modulation

An interesting side note on LEDs is that they're not easily dimmable. With an incandescent light bulb if you turn the voltage down, the bulb gets dimmer, and turning the voltage up results in a brighter light (until you go too far and the bulb burns out). This works for incandescent bulbs driven by either DC or AC power.

LEDs work differently. They start to conduct current when the voltage exceeds the V_f forward voltage, but changing the voltage has no effect on the light output because they're current-controlled devices (once the voltage is higher than V_f). There are minor variations in brightness as you increase the current, but mostly they're either on or off. By the way, it's easy to burn out an LED by putting too much current through - another reason for using a current-limiting resistor!

Big Idea # 11

The good news about LEDs is that they turn on and off really fast. So, if you turn them on and off very quickly, they can look dimmer to our eyes. Our eyes are not as quick as an LED, so if they flash quickly they look on, but dim. This technique is known a pulse width modulation or PWM. Using PWM the signal is pulsed on and off very quickly (on the order of 500Hz for Arduino's PWM). This can simulate the effect of being at an intermediate voltage between 0v and 5v by adjusting the percentage of time that the PWM pulses are high and low.

Arduino has a function that controls this type of pulse width modulation called `analogWrite(pin, value)`. The pin argument is the digital pin to control. This must be one of the PWM pins 3, 5, 6, 9, 10, and 11. The value is an int between 0 (fully off) and 255 (fully on). If the value is somewhere between 0 and 255, the PWM signal will be modulated to simulate a voltage between 0v and 5v by adjusting the pulses. When connected to an LED, this can provide a wide range of apparent brightness for that LED. Note that `analogWrite()` is ONLY usable on digital pins, not analog pins, and doesn't really produce an analog voltage, just a simulation of the analog voltage using PWM. Analog pins are used only as inputs on the Arduino.

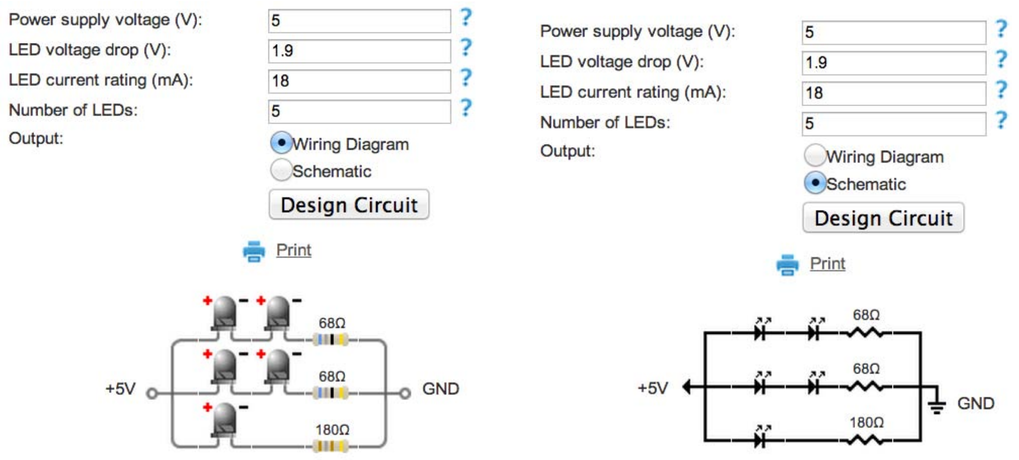
Here's a snippet of code that fades the LED from full off to full on and back again using analogWrite():

```
/* Use analogWrite() to fade an LED */
int ledPin = 10;     // LED on pin 10 (like Figure 4)

void setup()  {
  pinMode(ledPin, OUTPUT); // set pin 10 as output
 }

void loop()  {
  // fade LED from min to max in increments of 5 steps
  for(int val=0; val<256; val+=5) {
     analogWrite(ledPin, val); //sets the value (from 0 to 255)
     delay(30);                //wait a bit to see the effect
  }

  // fade LED from max to min in increments of 5 steps
  for(int val=255; val>=0; val-=5) {
     analogWrite(ledPin, val); //sets the value (from 0 to 255)
     delay(30);                //wait a bit to see the effect
  }
}
```

2.3 Driving lots of LEDs directly from Arduino

Causing an LED to flash, or a handful of LEDs to flash in a pattern, is fun, but what if you wanted to flash a lot of LEDs? What's a lot? 50? 100? 1000? The previous example showed each LED connected directly to a digital pin of the Arduino. Using this technique you could light up a maximum of 14 LEDs. You could technically light up more LEDs by attaching multiple LEDs to each pin of the Arduino. Remember Kirchhoff's current law says that components connected in series see the same current. So, if you connected two LEDs in series, they would each see the same amount of current sourced from the Arduino pin. But, remember that each LED also uses up V_f of the total voltage. So, if the Arduino drives a digital output to 5v, and the V_f of your LED is, say, 2v, then you can't put more than two LEDs in series because each one will use 2v, and after two of them are in series, there's only one volt left. This is not enough V_f to turn on the LED and let current pass through. Also, both LEDs would be on and off at exactly the same time because they're connected to the same digital pin, so it provides more light, but not extra visual complexity.

If you would like to figure out how to connect multiple LEDs together in series and parallel connections, there are a variety of LED calculators on the web that do just that. You can tell these web-based applications how many LEDs you would like to connect, what the power supply voltage is, what the V_f of the LEDs is, and the application will tell you what current-limiting resistor to use, and how to connect the LEDs in a series/-parallel circuit. Example of web based LED calculators are `ledcalculator.net/` and `led.linear1.org/led.wiz`. Examples of using the first calculator listed are seen in the following Figure. I've specified a 5v power supply, V_f of 1.9v, I want 18mA of current, and would like five LEDs. The tool shows me a series/parallel array that works with these specs, and the correct values for the required current-limiting resistors.



Power supply voltage (V):	5
LED voltage drop (V):	1.9
LED current rating (mA):	18
Number of LEDs:	5
Output:	[✓ Wiring Diagram] Schematic

Power supply voltage (V):	5
LED voltage drop (V):	1.9
LED current rating (mA):	18
Number of LEDs:	5
Output:	Wiring Diagram [✓ Schematic]

Figure 2.6: Examples from ledcalculator.net. The example on the left shows the wiring diagram with pictures of the LEDs and resistors. The example on the right is the same circuit, but shown as a schematic.

Note that you could not attach the +5v connection of these circuits to an Arduino digital pin to turn things on and off because each of the three branches of the circuit would consume 18mA, for 54mA total. This would exceed the Arduino's 40mA per-pin max. If you wanted to drive five LEDs from one Arduino pin you would have to specify a current of 13mA (for 39mA total) which would result in current-limiting resistors of 100Ω, 100Ω, and 240Ω for the circuit shown in Figure 2.6.

2.4 Driving lots of LEDs using external LED-driver chips

For situations where you want to drive a lot more LEDs, and drive them each separately on and off, you need external chip support. There are a large number of LED driver chips that you could choose from. These chips are really the way to go if you want to drive lots and lots of LEDs. Note that depending on the total number of LEDs in your system, you may need to provide extra power. Recall that even if you're using, say, a 9v external power supply into the 2.1mm power input, that power connection can provide a max of 800mA or so of total power. If you have enough LEDs that you're bumping up against that power limit, you'll need a separate power supply for the LEDs. Remember that if you're using multiple power supplies, you should tie all the grounds of all the sub-circuits together into one big ground. See Big Idea #7.

2.4.1 74HC595 latched shift register

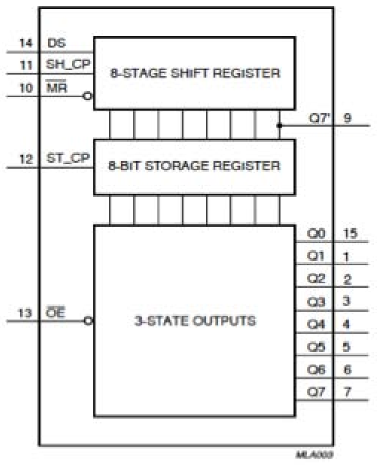
This chip is an 8-bit shift register. That is, it has a single input and on each shift-event it takes the logic value at the input (high voltage for 1 or low voltage for 0) and puts it in an 8-bit register, pushing all the other values one step further into the register. Imagine a pipe filled with ping pong balls that are labeled with a logical value (1 or 0). When you push one more ping pong ball into the pipe, the ball at the end of the pipe pops off and falls out. That's what a shift register is like. You enter data into the register one bit at a time, but you can see all eight bits that are currently in the register on separate output pins (like windows into the pipe looking at the value painted on each ping pong ball).

This shift register is referred to by its number 74HC595. The 74 refers to an entire family of integrated circuits known as the 7400 series that all start their part numbers with 74. The HC is a section of the code that tells in what technology the chip is fabricated. In this case it is made in a High-speed CMOS (HC) process. The 595 is the identifier that says that this is the "8-bit serial-in, serial or parallel-out shift register with 3-state output latches" chip in the 74HCxxx family. It's shorter, but more mysterious, to refer to it as a 74HC595.

To really see how any chip is used you need to look at its data sheet. I'm including data sheets for all the chips mentioned in this course in the additional materials. The pin out of the chip, and the logic diagram of the chip, are shown in Figure 2.7. These diagrams tell the practiced user what's inside the chip and how to connect to that chip. The logic diagram says that there is an 8-bit shift register that uses the following pins:

- DS (pin 14) is the data input to the shift register.
- SH_CP is the clock that determines when new data is shifted into the register. The data is shifted into the shift register on the rising edge of this clock.
- MR is a Master Reset that clears the register to 0. The fact that the signal is shown going into a bubble in the diagram, and has a bar over the MR, means that it is active low. That means that the action of that signal happens when the voltage is low, not when it is high.
- The ST_CP signal takes the data in the 8-stage shift register and transfers it all at once to the 8-bit storage register.
- From there, the 8 data values currently in the register are connected to the output through the Q7 - Q0 signals.
- These signals are 3-state signals which means that you can turn off their voltage drive using the OE signal (again, an active-low signal). If you turn off the drive to those outputs, they act as if they are disconnected from the circuit. (Advanced note - you can dim all of the LEDs driven by the 595 by using `analogWrite()` and PWM connected to this OE pin.)

To use the 74HC595 as an LED driver, you would use the Arduino (for example) to send bits one at a time into the shift register, and then use the ST_CP signal to transfer them to



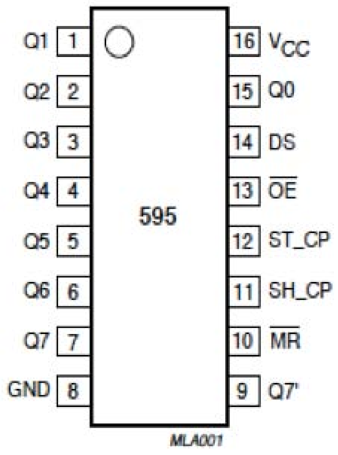


Figure 2.7: Logic diagram and chip pinout diagram for the 74HC595 shift register chip. The pinout shows where the signals are located on the physical package. All integrated circuit packages have some indication of where pin1 is, or where the "top" of the package is. This is typically a recessed dot (as in this picture) or a u-shaped divot in the end of the chip.

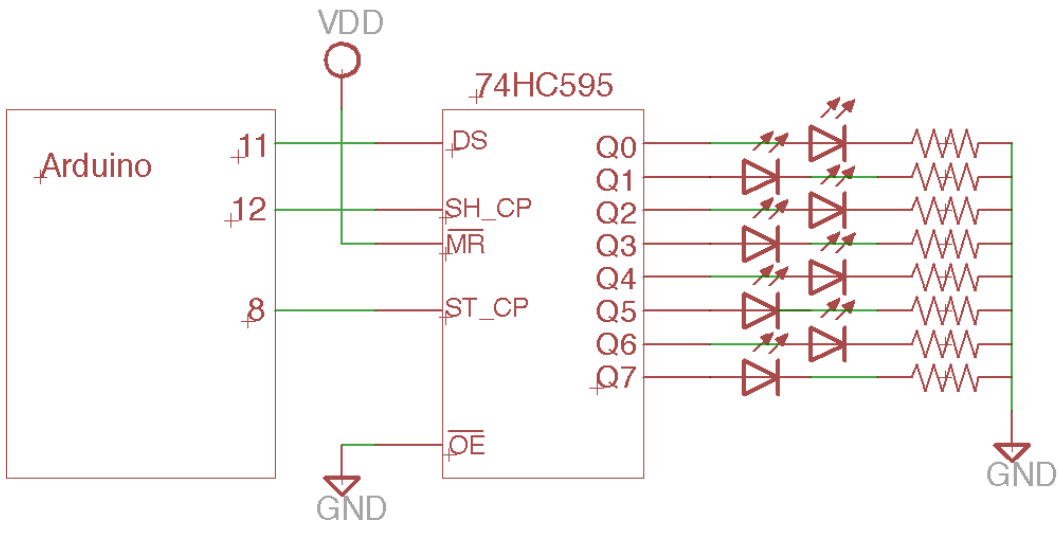


Figure 2.8: A circuit that uses a 74HC595 to control eight LEDs. The data (on or off) for the LEDs are shifted into the 595 from the Arduino one bit at a time. The Output Enable (OE) is tied low (active) so that the LEDs are lit. The Master Reset (MR) is tied high (inactive) so that the internal register is not reset to all 0's.

the output storage register. Then those bits would appear at the Q outputs, and you could put an LED (and a current-limiting resistor) on each of those outputs. If the data in Q3 (for example) is a 0, the LED would be dark, and if Q3 was high, the LED would light up. An example of this circuit is shown in Figure 2.8.

In the circuit in Figure 2.8 the Arduino is driving the data into the 595 on pin 11 of the Arduino, controlling the clock that causes new data to be shifted in on pin 12, and controlling the signal that transfers shifted data to the output register on pin 8. This circuit allows the Arduino to control eight LEDs using only three pins on the Arduino. The MR and OE signals in the circuit of Figure 12 are tied to 5v so that they're never active. This means that you're never resetting the register to all 0's, and you're never turning off the drive to the outputs. Another version of this schematic is seen in Figure 2.9, with a diagram of the physical wiring in Figure 2.10.

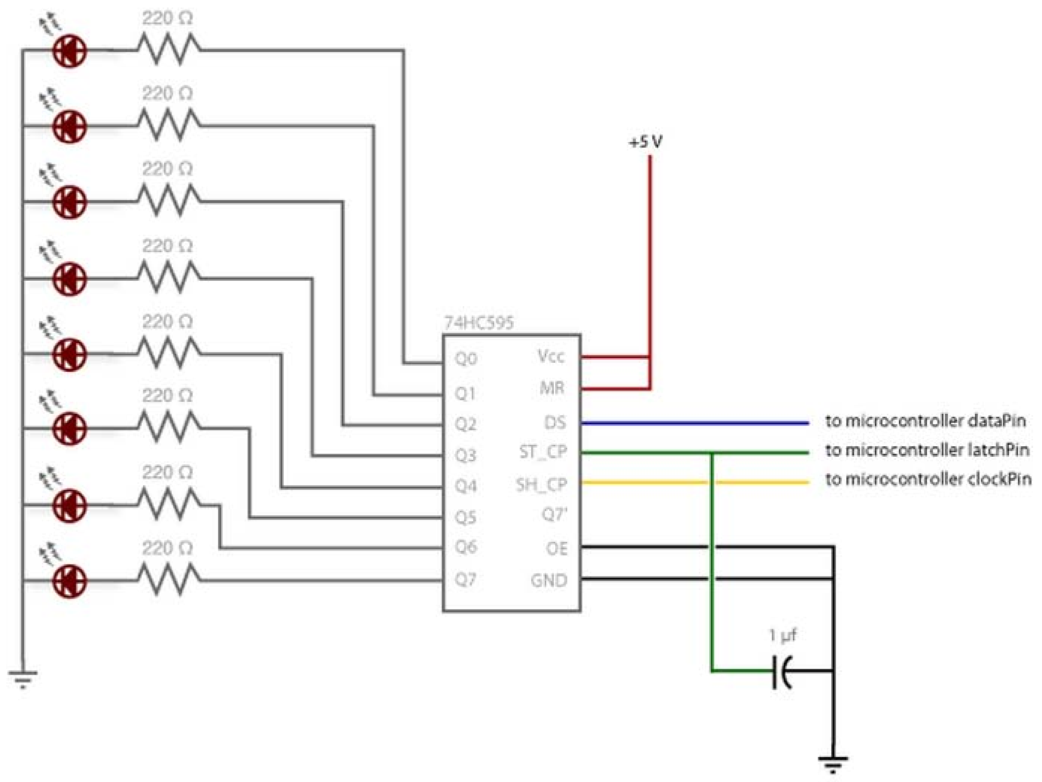

220 Ω
+5 V
74HC595
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Vcc
MR
DS
ST_CP
SH_CP
Q7'
OE
GND
to microcontroller dataPin
to microcontroller latchPin
to microcontroller clockPin
1 µf

Figure 2.9: Another schematic using a 74HC595 latched shift register. This one is from the `Arduino.cc` web site (`arduino.cc/en/Tutorial/ShiftOut`) and features a few extra ideas. Like Figure 13, the MR is tied high (inactive) and the OE is tied low (active). The 1uf component is a capacitor. This is a component that stores electrical charge. It's being used here as a decoupling capacitor between the latching signal and ground to avoid electrical noise on that signal which might capture incorrect data into the shift register. It's a nice feature, but not absolutely required. The circuit will almost certainly work without it. Note that the ordering of the series connection of the resistor and LED has been reversed from Figure 2.8. In this case the resistor connects to the chip and the LED connects to ground. The relative order of the components in a series connection does not matter because Kirchhoff's current law says that series connected components will experience the same current flow regardless of where they are in the series.

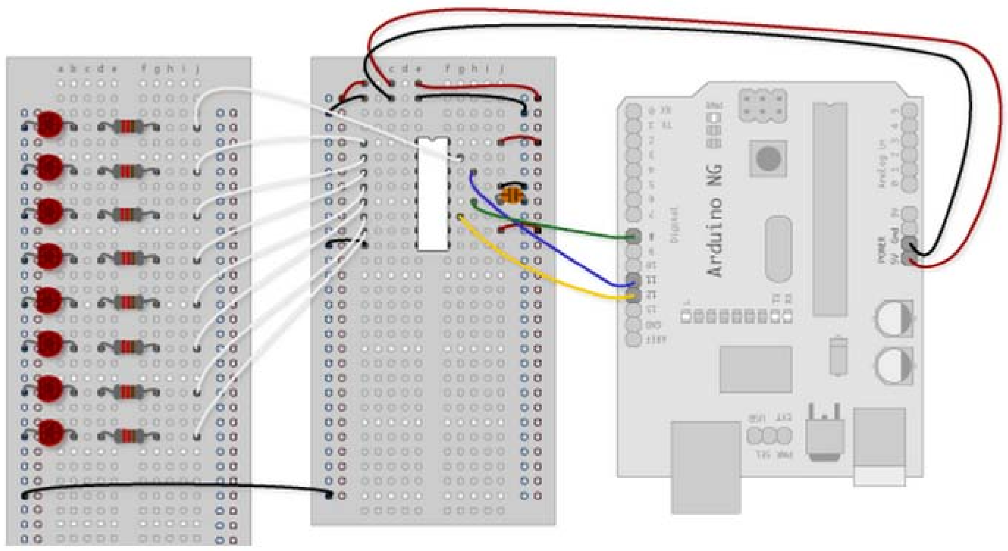


Figure 2.10: Another figure from the Arduino.cc web site (http://arduino.cc/en/Tutorial/ShiftOut). This picture shows the physical wiring of the circuit in Figure 14 with an Arduino on the right and two solderless breadboards on the left. The white chip in the center is the 74HC595. Note that in addition to the signal wires, the chip has been connected to power (+5v) using the red wire in the upper right of the chip, and to ground using the black wire on the lower left. All external chips must be connected to power and ground so that those chips get the power they need to operate.

Also, you could chain these shift registers together because the data that pops off the end of the shift register is on pin Q7. That pin could be connected to the DS pin of a second 74HC595 to chain them together. Now the Arduino could shift out 16 bits of data before capturing it in the output register, and get 16 LEDs for only three Arduino pins.

Code for driving a circuit like the one in Figure 2.8 or 2.9 is shown below. It turns out that this sort of serial data transfer is common enough that it has its own name: Serial Peripheral Interface or SPI (sometimes pronounced "spy"), and it has a special function built in to Arduino called `shiftOut()` that sends 8-bits of data on this type of link (see the Software Overview in Section 1.5.2).

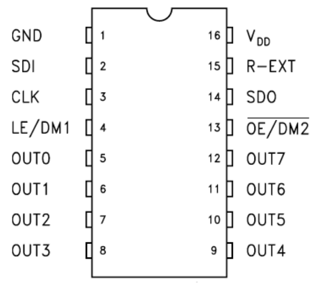
```
/*
 * An example of sending data to an external 74HC595 chip using the
 * shiftOut() function.
 */

//Pin Definitions
int data = 11;    // Pin connected to the 595's data (DS)input
int clock = 12;   // pin connected to the 595's clock (SH\_CP)input
int capture = 8;  // Pin connected to the 595's capture (ST\_CP)input

// set up the three pins from Arduino to the 595 as outputs
void setup() {
  pinMode(data, OUTPUT);
  pinMode(clock, OUTPUT);
  pinMode(capture, OUTPUT);
}

// set the pins to the binary numbers 0 through 255 over and over
void loop() {
  int delayMS = 100;                //Delay between LED updates
  for(int i = 0; i < 256; i++){    // count 0 to 255
     setLEDs(i);                    // update the LEDs to value i
     delay(delayMS);                // delay between updates
  }
}

// use shiftOut() to send 8 bits of data to the 595, and thus
// update the on/off status of each of the LEDs
void setLEDs(int value) {
  digitalWrite(capture, LOW);       //Set the capture pin low (inactive)
  shiftOut(data, clock, MSBFIRST, value); //transfer 8 bits to the 595
  digitalWrite(capture, HIGH);          //set the capture pin high to transfer
}
```

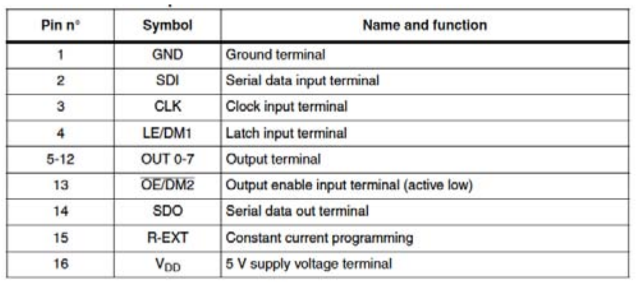
2.4.2 STP08DP05 LED driver chip

The 74HC595 is a great chip - it's inexpensive (around 45-90 cents per chip), and it's easy to use with the shiftOut() function. But, it has a big disadvantage: you still need to use a separate currently-limiting resistor for every LED. That's a lot of extra wiring, but necessary to limit the current to a safe value.

The STP08DP05 chip is very similar to the 595 in function: you can send data to the chip in exactly the same way as the 595 using the `shiftOut()` function. The biggest advantage of this chip over the 74HC595 is that the STP08DP05 has constant current outputs. The description of this chip on the datasheet is Low voltage 8-bit constant current LED sink. This means that there are special drivers on the output pins of this chip that automatically limit the output current to a particular value. The output current is set through a single resistor that sets the output current for the whole chip. The pins of the SPT08DP05 are shown in Figure 2.11. If you wanted to use the same code as for the 74HC595, you would connect the data_pin to the SDI input, the clock_pin to the CLK input, and the capture pin to the LE/DM1 input.

The LEDs can be connected directly to the 8 outputs labeled OUT0 to OUT7. A close examination of the logic diagram (Figure 14) shows that the drivers for the outputs can only pull the outputs low, not drive them high (that's why the chip description is a "constant current LED sink"). That is, the pins can sink current to ground, but not source current to a high voltage. Thus, the LEDs must be connected with their anodes connected to a power source (5v) and their cathodes connected to the STP08DP05 outputs.





Pin n°	Symbol	Name and function
1	GND	Ground terminal
2	SDI	Serial data input terminal
3	CLK	Clock input terminal
4	LE/DM1	Latch input terminal
5-12	OUT 0-7	Output terminal
13	$\overline{OE/DM2}$	Output enable input terminal (active low)
14	SDO	Serial data out terminal
15	R-EXT	Constant current programming
16	V_{DD}	5 V supply voltage terminal

Figure 2.11: Pin assignments and definitions for the STP08DP05

The R-EXT pin on the STP08DP05 is where the current-limiting resistor for the entire chip is connected. The value of the resistor sets the current limit for all the outputs. A

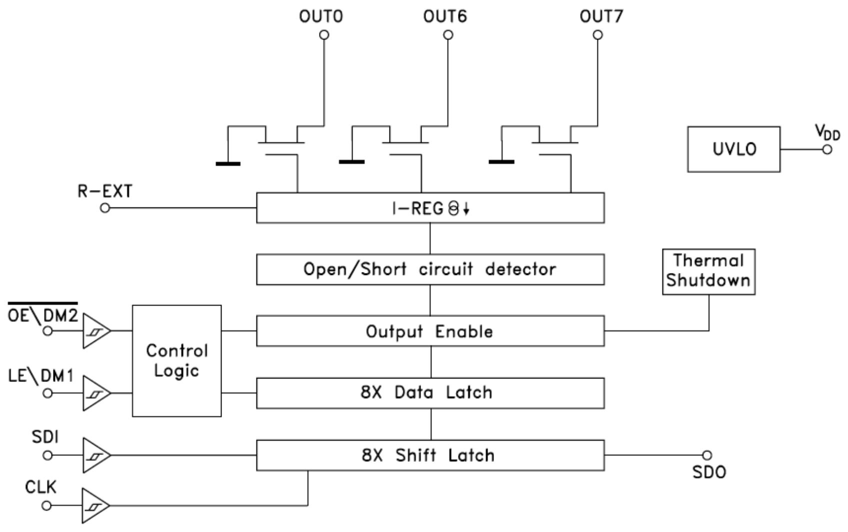


Figure 2.12: Logic diagram for the STP08DP05

table in the STP08DP05 gives a range of values for a range of currents. For example, if you want to limit the output current to 10mA, choose an external resistor of around 1900Ω. The external resistor is connected to the R-EXT pin, and then to ground outside the chip. Using this chip you need only one resistor per chip instead of one resistor per LED. You can also chain these chips together in the same way as you would chain 595 chips together. You pay a little more for the STP08DP05 because of the advantage of the constant current outputs. This chip sells for around $1.50-$2.00 each.

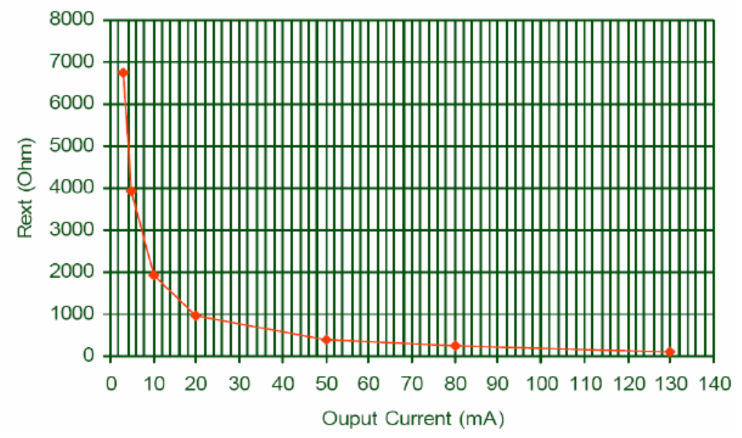


Figure 2.13: A graph of the external resistor value to the current limit on the output pins.

This next set of figures shows a detailed look at wiring up an external STP08DP05 chip on a solder less breadboard. If you've wired things up before I'm sure you can skip it. If you've never wired anything, I though it might be nice to see the whole process.

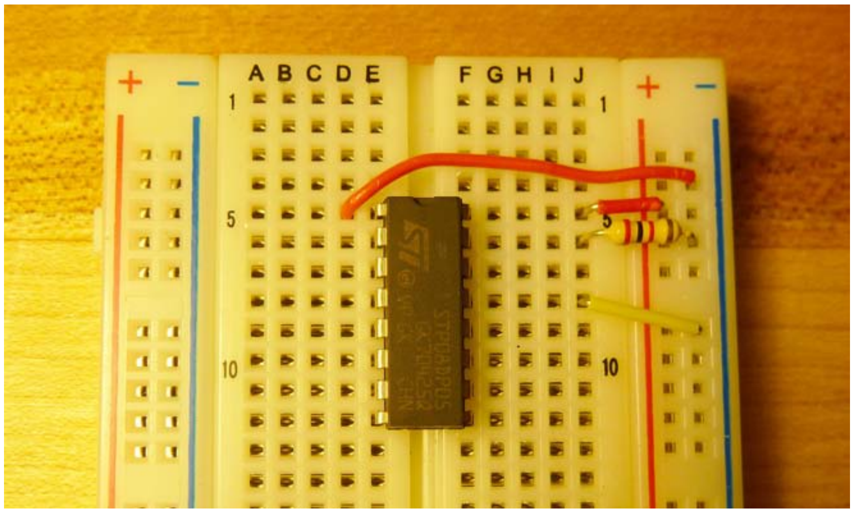


Figure 2.14: First Step: Put the STP08DP05 chip into the solderless breadboard. Note that the "legs" of the chip straddle the gap in the middle of the board. This means that each leg has four other holes that wires can go into and connect to that leg of the chip. If you look carefully in this figure you'll see a U-shaped divot in the top of the chip. This tells you where the top is with respect to the pin numbering. Pin 1 is in the top left, just like in Figure 2.11. The first connections to make are GND (pin 1), Vdd (Pin 16), and OEbar (pin 13). I've tied OEbar to ground so that the outputs are always enabled. I've also put a resistor between the R-EXT pin (pin 15) and ground to set the current limit for all the LEDs. This resistor is 2kΩ.

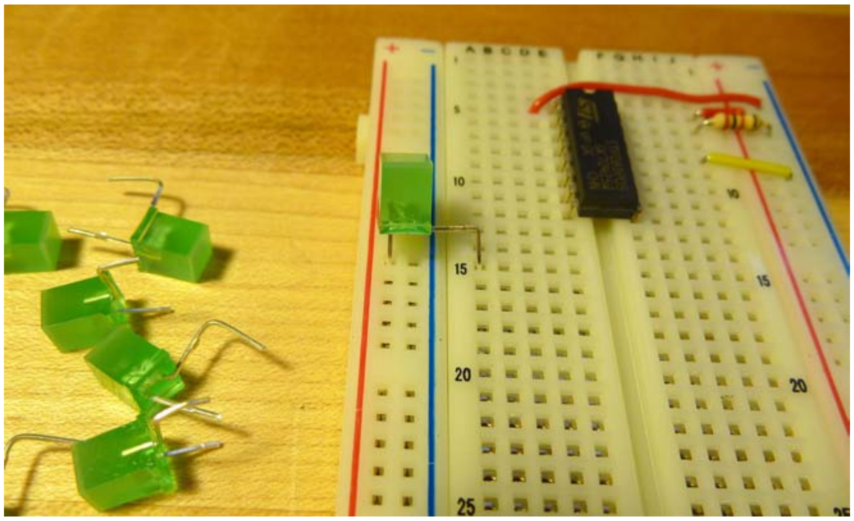


Figure 2.15: Now I'm ready to plug in the LEDs. I've cut the leads so that the LEDs fit snuggly in the board. The Anode of each LED goes to Vdd and the cathode goes to a row that I can then connect to the the STP08DP05.

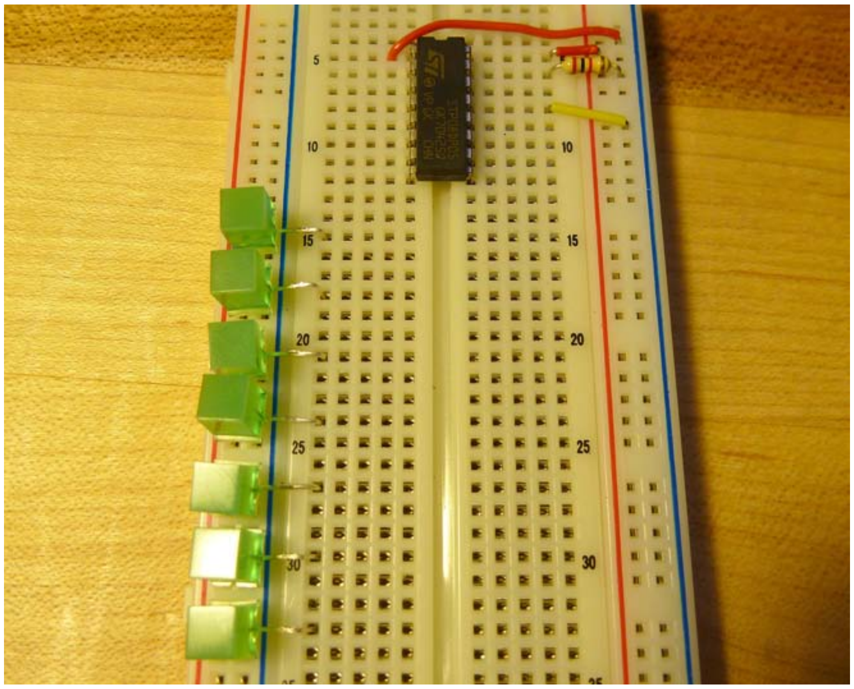


Figure 2.16: All 8 LEDs are now placed in the board. I'm ready to start connecting the outputs from the STP08DP05 to the LEDs.

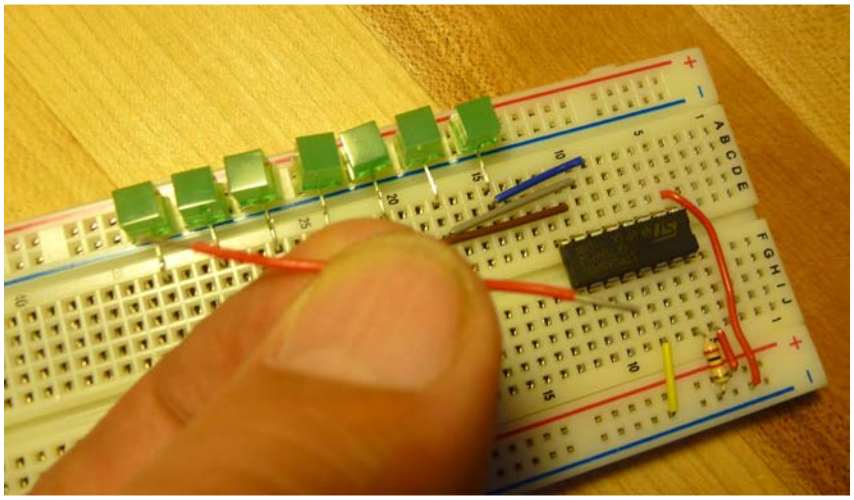


Figure 2.17: I've started to connect Out0 through Out7 from the STP08DP05 to the LEDs. You can see the blue, grey, and brown wires in the board. I'm holding a red wire that I'm about to put in the board. You can see that both ends of this 22AWG solid wire have been stripped a little so that they can poke into the breadboard.

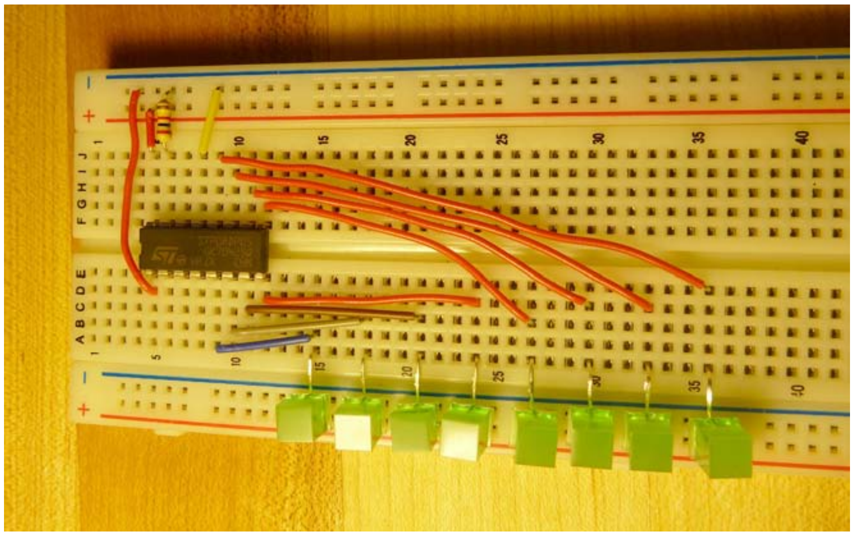


Figure 2.18: All the LEDs are now connected. According to the data sheet, Out0 through Out7 on the STP08DP05 are on pins 5 - 12 on the chip.

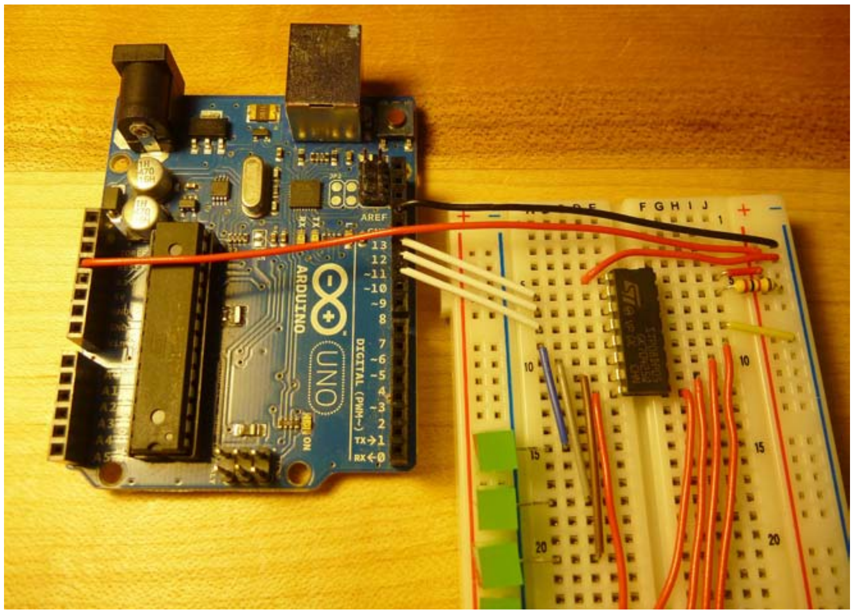


Figure 2.19: Now the Arduino is also connected. The SDI (data-input), CLK, and LE (latch-enable) are connected to pins 12, 11, and 10 on the Arduino. I've also connected GND on the Arduino to the far right ground bus on the breadboard (black wire), and +5v of the Arduino to the far right power bus on the breadboard (red wire). Not seen in this photo, the power bus on the far right is connected to the power bus on the left side of the breadboard out of range of the camera. I'm now ready to run a program like the one on the previous pages to drive values onto the STP08DP05 and have them show up on the LEDs.

2.4.3 MAX 7219/7221 LED driver chip

The STP08DP05 is a great little chip - it can drive 8 LEDs directly while using only three pins on the Arduino. If you daisy-chain multiple chips you can drive even more LEDs without using more pins on the Arduino. However, there's an LED driver chip that can drive even more LEDs - the MAX 7219 and MAX 7221 chips (these chips are essentially identical for our purposes) can drive 64 LEDs from one chip, and these chips also use constant current drivers so you don't need 64 resistors. These chips are actually designed to drive 7-segment numeric LED displays. These are displays like you might find on a calculator.

A seven-segment LED has, you might expect, seven LED segments. These are typically shaped like bars instead of circles, and are organized as a figure eight so they can be can be used to display the numbers 0 through 9. Actually, most seven-segment LEDs have an 8th segment that is the decimal point for each number as seen in Figure 2.20. Each of the eight LEDs in one digit has a separate anode, but is connected to a common cathode. If you want to see a 0, for example, you would set the anodes of segments A, B, C, D, E and F to 1, and the anodes of segments G and DP to 0. Then when you connect the common cathode to ground, current flows and the LED segments light up.

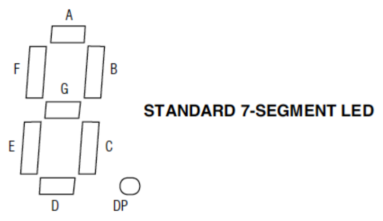


Figure 2.20: A standard seven-segment LED organization (ironically with eight LEDs). These segments can be lit up in various combinations to make the numerals 0 through 9. This figure is from the MAX 7219/7221 data sheet, Maxim Integrated Products.

The reason that the segments have a common cathode is more obvious when you put multiple digits in a row. All of the anodes for each segment in the multi-digit display are connected together and each digit has a common cathode. That is, if you have a 5-digit display, all five of the anodes for the A segments are connected together. The other segments likewise have their anodes connected together. Each digit has a separate cathode that connects all the cathodes in that digit. This way you can cycle through the digits one at a time. Start with all five of the cathodes high. That is, none of the LEDs or digits is lit up. First you set the A through G (plus DP) segments for the first digit, then pull that digit's cathode low to light it up. Then you set the first digit's cathode back to high, change the segment values to the values for the second digit, and pull the second digit's cathode low to light it up. If you do this for all five digits in turn, then each of the five digits can have a different value, but they light up in sequence, not all together.

However, remember the PWM technique for dimming an LED? If you flash LEDs quickly enough, they look like they're on all the time. Your eye integrates the on-time and makes it look like it's all the way on when instead it's flashing too quickly to make out the flashes. That's how multiple digit LED segments work: the driving chip cycles through each digit and does it fast enough that they all look on at the same time, albeit a little bit dimmer than if they really were on all the time. The Max 7219/7221 chip scans digits at 800Hz (800 times per second): Much too fast for your eyes to see the blinking.

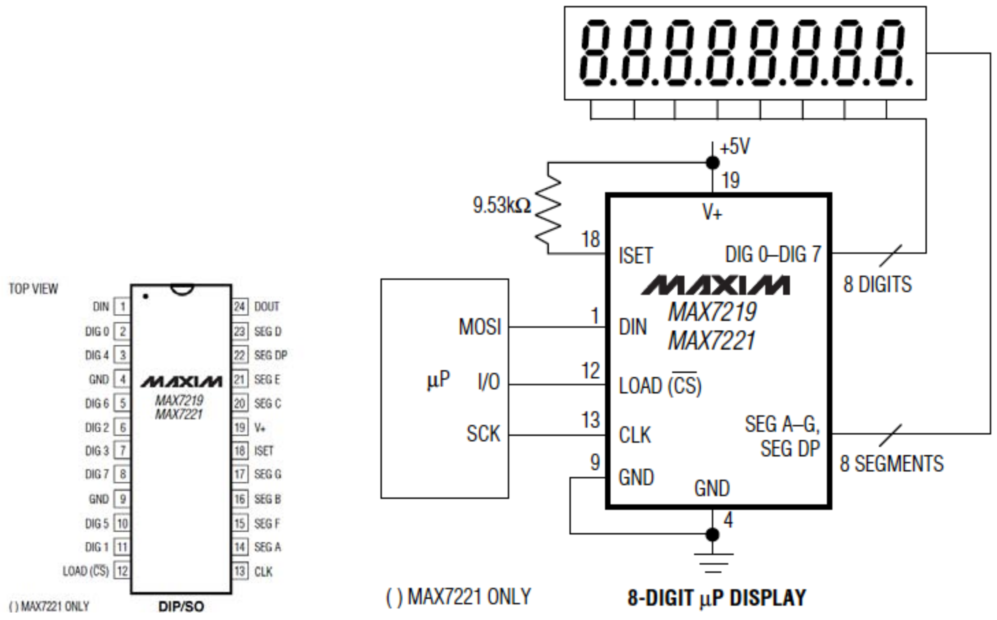


Figure 2.21: The pinout of the Max 7219/7221 chip, and an example of how it is used to drive a, 8-digit numeric (seven-segment) display. Note that the **8-segments** are driving the A-G and DP segments (i.e the anodes of those LEDs), and the **8-digits** are pulling down the common cathodes of each digit in sequence. On the microprocessor side, you would use an Arduino and choose whatever digital pins you like. You could then use `shiftOut()` to send data to the Max chip in the same way as the code for the STP08DP05. Diagrams are from the MAX 7219/7221 data sheet, Maxim Integrated Products.

Figure 2.21 shows how the Max 7219/7221 chip could be used to drive an 8-digit numeric display where each digit in the display is a seven-segment digit (with 8 LED segments). The Max chip handles the sequencing of the digits to make it look like they're all on at once. The user sends data from the microcontroller to the Max chip to tell it what should be displayed on each of the 8 digits. This communication is a little more complex than for the previous chips because you're specifying more things. In addition to telling the Max chip what should be displayed on each of the digits, you can tell the chip how bright things should be, which of the digits should actually be displayed, and a variety of other things. This is all layered on top of the simple SPI serial communication protocol used in the previous chips, so you can use `shiftOut()` to send the data, but the data to be sent is a little more complex. More on that in a moment.

Another thing that you can use the Max 7219/7221 chip for is driving 64 separate LEDs. If you think about what's involved in driving an 8-digit display where each digit consists of eight common-cathode LEDs, that means that the chip is driving a total of 8x8=64 LEDs. It does this by cycling through the LEDs in groups of eight, but it does this at 800Hz so the human eye thinks that they're all on at the same time. So, if you want to light up 64 separate LEDs, and you're willing to wire them up as eight groups of eight cathode-connected LEDs, this is the chip for you.

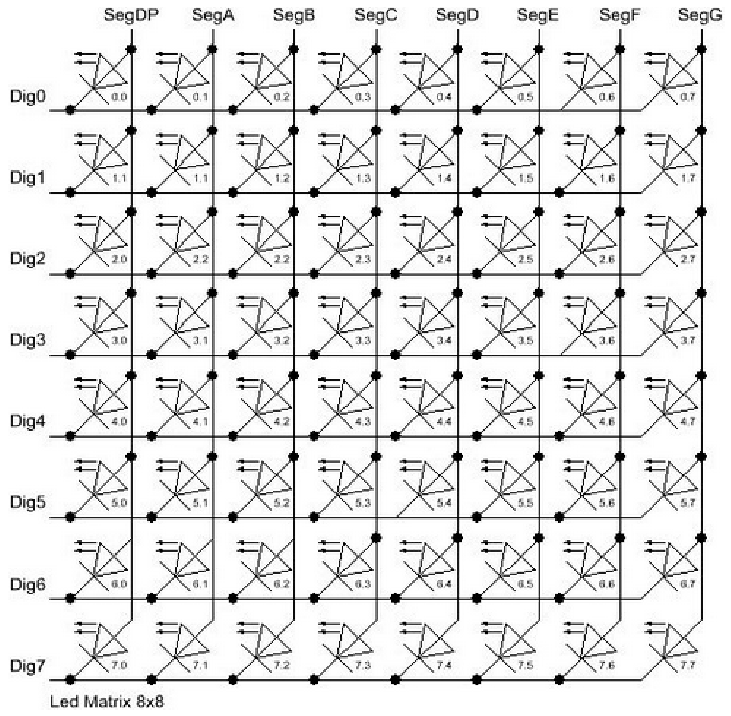


Figure 2.22: An 8x8 matrix of LEDs. Of course, your physical organization of the LED doesn't need to be a square. This is just showing the logical connections that must be made. This figure is from the ledControl library documentation page `playground.arduino.cc/Main/LedControl`

.

Figure 2.22 shows an 8x8 array of LEDs connected in a way that works with the Max 7219/7221. Each row of this diagram has a common cathode for the whole row (like the LEDs in one digit). Each column is anode-connected (like the segments in a digit). Connected in this way, the Max7219/7221 can set each of the 64 LEDs independently and by cycling through the rows it will look like they're all lit (or not) at the same time. This figure shows the LEDs in a square, and of course you could connect them that way. In fact, you can buy pre-assembled LED matrixes that are constructed like this. But, as long as you maintain this logical connectivity, your LEDs could be physically placed in whatever pattern you like. Now it's a simple matter of programming to choose how to light up the individual LEDs to make the patterns that you want to see.

The Max 7219/7221 chips have constant-current drivers so you don't need separate current-limiting resistors on every LED. The current drive is set using one resistor, similar to the STP08DP08. But, the value is different, and in this case the resistor connected to the ISET pin (remember that the symbol for current is I) is connected to Vdd (power), not ground. The table of current-setting resistor values for the Max7219/7221 chips is shown in Figure 22. Note that the values in this table change depending on the forward voltage V_f of your LEDs, and they are in kilo-ohms. You can see where this resistor connects to the chip and to the power supply in the schematic in Figure 2.21.

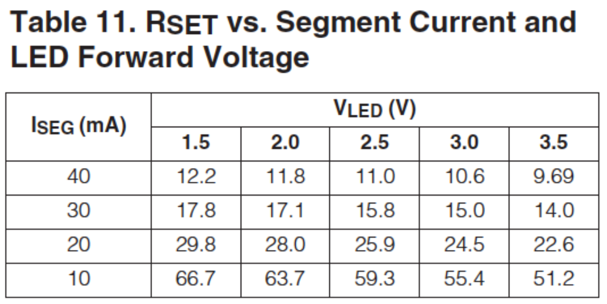


Table 11. RSET vs. Segment Current and LED Forward Voltage

ISEG (mA)	VLED (V)				
	1.5	**2.0**	**2.5**	**3.0**	**3.5**
40	12.2	11.8	11.0	10.6	9.69
30	17.8	17.1	15.8	15.0	14.0
20	29.8	28.0	25.9	24.5	22.6
10	66.7	63.7	59.3	55.4	51.2

Figure 2.23: Current-setting resistor values for the constant current outputs on the Max 7219/7221 LED driver chips. Note that the values in the table are in kilo-ohms. For example, if your LEDs have a forward voltage (VLED in this table) of 2.5v, and you want the current limited to 20mA, you should use a 25.9k resistor as the ISET resistor. Table is from the MAX 7219/7221 data sheet, Maxim Integrated Products

OK, maybe it's not all that simple. The Max7219/7221 requires that you send multiple commands and data to the chip to set it up to control the LEDs. The data sheet has all the details. Luckily, someone has written a great library that captures all that information. In fact, because it's so popular, there are multiple libraries for this chip. You can find them on the Arduino.cc web site in the playground section: `playground.arduino.cc/Main/LEDMatrix`. This has pointers to a number of different libraries that interface to this chip. I like the led-control library documented at `playground.arduino.cc/Main/LedControl`. It allows you to send digits to a Max 7219/7221 connected to a multi-digit display, and also control each LED separately if you're using 64 separate LEDs.

ledControl Library

This library, written by Eberhard Fahle, is a nice API for the Max7219/7221 chip that has the user create an object for each Max chip, or series-connected set of Max chips. User code can then set individual LEDs on or off, set entire rows or columns to an 8-bit value all at once, or display a number on a specific digit of a multi-digit display. The ledControl library, and documentation, is available at `playground.arduino.cc//Main/LedControl`. The example program that comes with the library and that drives patterns onto an 8x8 LED

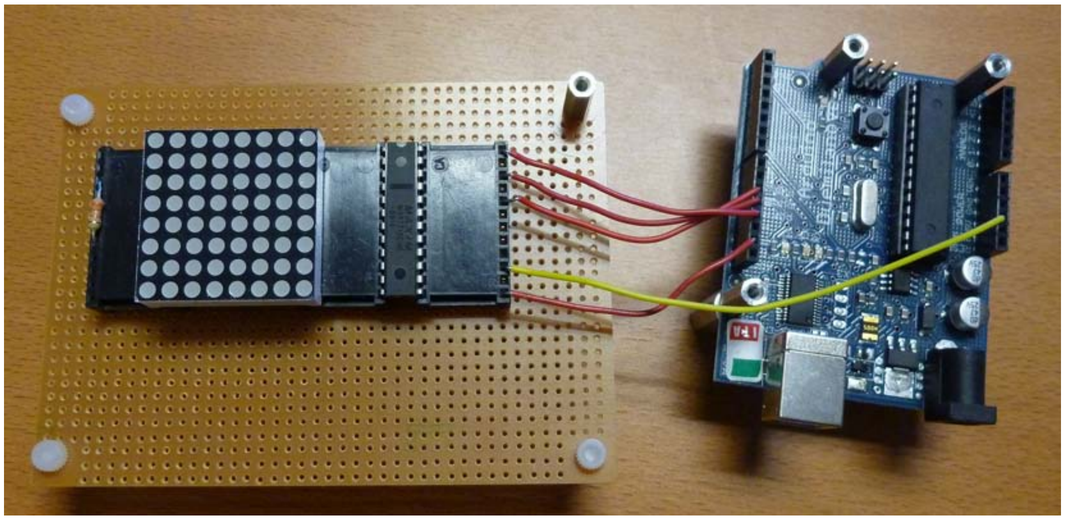


Figure 2.24: A MAX7219 chip connected to an Arduino and to an 8x8 matrix LED display.

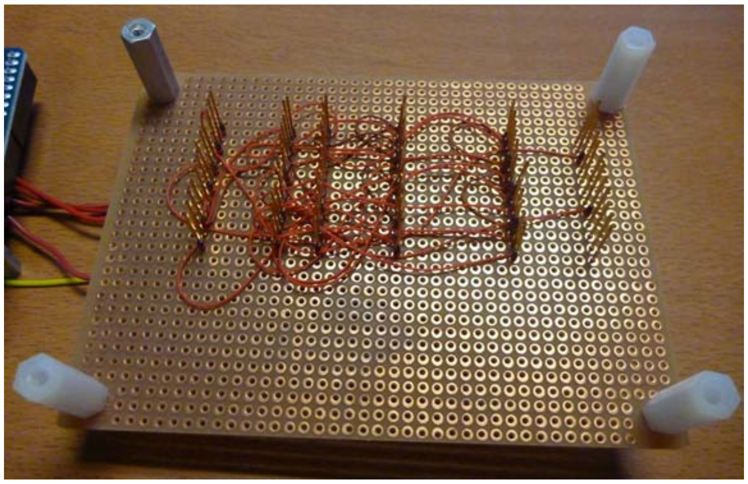


Figure 2.25: The wiring of the MAX7219 chip and the LED matrix. These connections are made using a technique called wire-wrap. In this connection technique thin 30AWG wires are wrapped tightly around metal posts using a special tool. The connections are more secure than with a solderless breadboard, but take a little more time to make.

matrix (connected as in Figure 2.22) is shown here (I've augmented the comments somewhat).

```
// Example program from ledControl distribution.
#include "LedControl.h"   //We always have to include the library

/* Now we need an LedControl object to work with.
***** Change these pins numbers to work with your hardware *****
* pin 12 is connected to the DataIn
* pin 11 is connected to the CLK
* pin 10 is connected to LOAD
* We have only a single MAX72XX.
*/
LedControl lc=LedControl(12,11,10,1);

unsigned long delaytime=100; // delay between updates of the display

void setup() {
  lc.shutdown(0,false); // Wake up the MAX72XX
  lc.setIntensity(0,8); // Set the brightness to a medium values
  lc.clearDisplay(0);   //  and clear the display
}

/*
 * This method will display the characters for the
 * word "Arduino" one after the other on the matrix.
 * (you need at least 5x7 leds to see the whole chars)
 */
void writeArduinoOnMatrix() {
  /* here is the data for the characters */
  byte a[5]={B01111110,B10001000,B10001000,B10001000,B01111110};
  byte r[5]={B00111110,B00010000,B00100000,B00100000,B00010000};
  byte d[5]={B00011100,B00100010,B00100010,B00010010,B11111110};
  byte u[5]={B00111100,B00000010,B00000010,B00000100,B00111110};
  byte i[5]={B00000000,B00100010,B10111110,B00000010,B00000000};
  byte n[5]={B00111110,B00010000,B00100000,B00100000,B00011110};
  byte o[5]={B00011100,B00100010,B00100010,B00100010,B00011100};

  /* now display them one by one with a small delay */
  /* I've stacked multiple statements per line to save space */
  lc.setRow(0,0,a[0]); lc.setRow(0,1,a[1]); lc.setRow(0,2,a[2]);
  lc.setRow(0,3,a[3]); lc.setRow(0,4,a[4]);
  delay(delaytime);
  lc.setRow(0,0,r[0]); lc.setRow(0,1,r[1]); lc.setRow(0,2,r[2]);
  lc.setRow(0,3,r[3]); lc.setRow(0,4,r[4]);
  delay(delaytime);
  lc.setRow(0,0,d[0]); lc.setRow(0,1,d[1]); lc.setRow(0,2,d[2]);
  lc.setRow(0,3,d[3]); lc.setRow(0,4,d[4]);
  delay(delaytime);
  lc.setRow(0,0,u[0]); lc.setRow(0,1,u[1]); lc.setRow(0,2,u[2]);
  lc.setRow(0,3,u[3]); lc.setRow(0,4,u[4]);
  delay(delaytime);
  lc.setRow(0,0,i[0]); lc.setRow(0,1,i[1]); lc.setRow(0,2,i[2]);
  lc.setRow(0,3,i[3]); lc.setRow(0,4,i[4]);
  delay(delaytime);
  lc.setRow(0,0,n[0]); lc.setRow(0,1,n[1]); lc.setRow(0,2,n[2]);
  lc.setRow(0,3,n[3]); lc.setRow(0,4,n[4]);
  delay(delaytime);
  lc.setRow(0,0,o[0]); lc.setRow(0,1,o[1]); lc.setRow(0,2,o[2]);
  lc.setRow(0,3,o[3]); lc.setRow(0,4,o[4]);
  delay(delaytime);
  lc.setRow(0,0,0); lc.setRow(0,1,0); lc.setRow(0,2,0);
  lc.setRow(0,3,0); lc.setRow(0,4,0);
  delay(delaytime);
}

/*
 * This function lights up some LEDs in a row.
 * The pattern will be repeated on every row.
 * The pattern will blink along with the row-number.
 * row number 4 (index==3) will blink 4 times etc.
 */
void rows() {
  for(int row=0;row<8;row++) {
     delay(delaytime);
     lc.setRow(0,row,B10100000);
     delay(delaytime);
     lc.setRow(0,row,(byte)0);
     for(int i=0;i<row;i++) {
        delay(delaytime);
        lc.setRow(0,row,B10100000);
        delay(delaytime);
        lc.setRow(0,row,(byte)0);
     }
   }
}

/*
 * This function lights up some LEDs in a column.
 * The pattern will be repeated on every column.
 * The pattern will blink along with the column-number.
 * column number 4 (index==3) will blink 4 times etc.
 */
void columns() {
  for(int col=0;col<8;col++) {
     delay(delaytime);
     lc.setColumn(0,col,B10100000);
     delay(delaytime);
     lc.setColumn(0,col,(byte)0);
     for(int i=0;i<col;i++) {
        delay(delaytime);
        lc.setColumn(0,col,B10100000);
        delay(delaytime);
        lc.setColumn(0,col,(byte)0);
     }
  }
}

/*
 * This function will light up every Led on the matrix.
 * The led will blink along with the row-number.
 * row number 4 (index==3) will blink 4 times etc.
 */
void single() {
  for(int row=0;row<8;row++) {
     for(int col=0;col<8;col++) {
        delay(delaytime);
        lc.setLed(0,row,col,true);
        delay(delaytime);
        for(int i=0;i<col;i++) {
           lc.setLed(0,row,col,false);
           delay(delaytime);
           lc.setLed(0,row,col,true);
           delay(delaytime);
        }
     }
   }
}
```

```
void loop() {
  writeArduinoOnMatrix();
  rows();
  columns();
  single();
}
```

The ledControl library is fully documented on the website `playground.arduino.cc//Main/LedControl`. The main features are:

- The Max chips are communicated to through an ledControl object. The ledControl creation method has four arguments:
  `ledControl <name> = ledControl(Data, Clock, Latch, Num);`
  - The pin number connected to the Data pin on the Max chip
  - The pin number connected to the Clock pin on the Max chip
  - The pin number connected to the Latch pin on the Max chip
  - The number of Max chips connected together. The Max chips can be chained in a serial chain by connecting the Clock and Latch wires in parallel to all chips, and then connecting the Dout (Data-Out) pin of one chip to the Din (Data-In) pin of the next chip in series. The ledControl library can handle eight Max chips in series for each ledControl object. See Figure 2.26 for an example of chaining two Max chips in series.
- By default the Max chip is in shutdown mode where it's not driving the inputs. Take it out of shutdown with
  `<name>.shutdown(num, false);`
  where num is the ID of the Max chip you're talking to (numbered starting at 0 for the first chip).
- You can set the intensity (brightness) of the LEDs using the
  `<name>.setIntensity(num, level);`
  where num is the ID of the Max chip you're talking to, and level is a number 0 though 15 where 15 is the brightest intensity.
- You can clear the display to all-off with
  `<name>.clearDisplay(num);`
- Set the on/off status of an individual LED with
  `<name>.setLed(num, row, col, state);`
  where num is the ID of the chip, row and col are the address of the LED you're changing, and state is true for on and false for off.
- Set the on/off status of all LEDs in a row of the matrix using
  `<name>.setRow(num, row, byte);`
  where num is the chip ID, row is the address of the row, and byte is the data for that row. The 1-bits correspond to turning the LED on, and 0-bits for off.
- Set the on/off status of all LEDs in a column of the matrix using
  `<name>.setColumn(num, col, byte);`
  This works the same way as `setRow()`, but be aware that it requires a more complex set of commands to the chip than `setRow()` so it takes longer. That might not matter, but of speed does matter, organize things to be able to use `setRow()` instead.

- If you're using the Max chip to control a multi-digit seven-segment display, then you can use
  `<name>/setChar(num, digit, char, dp);`
  to make a digit take on the shape of a character. As usual, num tells which chip you're talking to. Digit is the address of the digit you're changing, and char is the character to put on that digit. Use dp to turn on/off the decimal point (the eighth segment in the seven-segment display). The char can be one of the following: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, h, l, p, -, ., _, and <space> (denoted as ' ' in the argument list). Those are all the characters that can be distinguished easily on a seven-segment display.

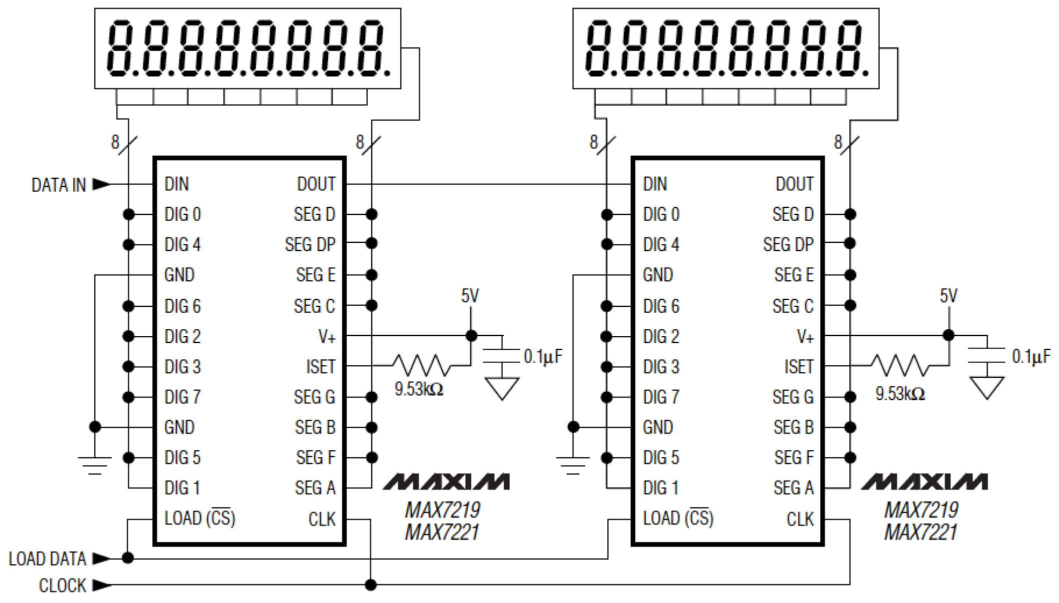


Figure 2.26: This shows two Max 7219/7221 chips installed in series. The LOAD DATA and CLOCK signals from the Arduino go to all the chips. The DATA-IN on the far left comes from the Arduino. For the series connected chips the DOUT from one chips goes to the DIN of the next. Note that each chip needs its own external current-setting resistor on the ISET pin. The 0.1uF capacitor is optional and decouples the 5v supply from ground. The chips are shown here controlling 8-digit displays, but they could just as easily each be controlling an 8x8 LED array.

2.4.4 The Texas Instruments TLC5940 LED driver chip

All of the chips described so far will drive a set of LEDs on the output, and are controlled by a simple SPI communication through the Arduino's shiftOut() function. The STP05DP08 and Max 7219/7221 have constant-current drivers on the outputs so that the output current for the connected LEDs is set by a single current-setting resistor. The Max chip controls the brightness of the LEDs using an internal command that changes the pulse width modulation at the output, but for all the LEDs at one time. You can similarly control the brightness of all the LEDs connected to the STP05DP08 chip by using the Arduino's PWM on the OE (Output Enable) pin.

However, these brightness settings control all the LEDs connected to the chip at once. If you'd like to individually control the brightness (using PWM) of each LED attached to the LED driver chip, you need a different chip. You need a TLC5940. This chip drives only 16 LEDs (not the 64 that the Max chips drives), but it has individual PWM on every pin. In fact, there are 4096 levels of brightness for each LED driven by the TLC5940. That is, 4096 different pulse width modulation options for each pin.

The downside of this chip is that the communication protocol is not as simple as SPI, so you can't just use shiftOut() to send data to the chip. In fact, it's so complex that it takes a carefully designed timer-driven interrupt handler to get it right. Luckily, this too has a nice library of commands written by a helpful open-source community member (Alex Leone). The library is available on Google Code at `code.google.com/p/tlc5940arduino/`.

Basic usage for the TLC5940 library is as follows. There are more advanced functions that are documented on the Google Code repository:

- The library pre-defines which pins to use for the required connections from the Arduino to the TLC5940:
  - Pin 13 of the Arduino to pin 25 of the TLC5940 (SCLK)
  - Pin 11 of the Arduino to pin 26 of the TLC5940 (SIN)
  - Pin 10 of the Arduino to pin 23 of the TLC5940 (BLANK)
  - Pin 9 of the Arduino to pin 24 of the TLC5940 (XLAT)
  - Pin 3 of the Arduino to pin 18 of the TLC5940 (GSCLK)
- The other connections required by the TLC5940 are:
  - +5v to pins 21 and 19 of the TLC5940 (VDD and DCPRG)
  - GND to pins 22 and 27 of the TLC5940 (GND and VPRG)
  - A resistor for setting current limit from pin 20 of the TLC5940 to GND. Typically this is between 2k (20mA) and 4k (10mA).
- The library assumes that you're only using one TLC4950, and defines NUM_TLCS to 1. If you daisy chain more chips, you need to update this value in the source code in `tlc_config.h` in the library folder.
- The library predefines an object for the chip called Tlc.
- Use `Tlc.init()` to initialize the library - typically in `setup()`.
- Use `Tlc.clear()` to set all the grayscale values of the LEDs to 0. This does not actually send the values, it just sets them for the next time that they'll be sent.
- `Tlc.set(channel, value)` sets the output value (brightness) for an LED. The channel is the pin number between 0 and 15 for one chip. If you daisy chain extra chips the channel number keeps increasing. The second chip would drive LEDs 16-31, for example. The value is the grayscale value for that LED in the range of 0 (off) to 4095 (fully on). Like Tlc.clear(), this doesn't actually send the data to the chip, it just accumulates updates.
- `Tlc.update()` sends the accumulated LED brightness values to the chip. An example from the library is given below. It shows how to wire up the chip, and a simple program that makes the "Knight Rider" (also known as the Cylon eye) effect on the LEDs connected to the TLC5940. It makes use of the fading values so that the moving

LED is fully on (PWM value 4095), and the trailing LED to that one is faded to a much lower value (PWM value of 1000).

```
// This is the BasicUse example from the TLC5940 library by
// Alex Leone. I've reformatted some of the comments to fit
// this course notes format. (Erik Brunvand)
/*
  Basic Pin setup:
  ------------                                 ---u----
  ARDUINO   13|-> SCLK (pin 25)        OUT1 |1       28| OUT channel 0
            12|                        OUT2 |2       27|-> GND (VPRG)
            11|-> SIN (pin 26)         OUT3 |3       26|-> SIN (pin 11)
            10|-> BLANK(pin 23)        OUT4 |4       25|-> SCLK (pin 13)
             9|-> XLAT (pin 24)           .  |5       24|-> XLAT (pin 9)
             8|                           .  |6       23|-> BLANK (pin 10)
             7|                           .  |7       22|-> GND
             6|                           .  |8       21|-> VCC (+5V)
             5|                           .  |9       20|-> 2K Res. -> GND
             4|                           .  |10      19|-> +5V (DCPRG)
             3|-> GSCLK (pin 18)          .  |11      18|-> GSCLK (pin 3)
             2|                           .  |12      17|-> SOUT
             1|                           .  |13      16|-> XERR
             0|                        OUT14|14       15| OUT channel 15
  ------------                                 --------

  - Put the longer leg (anode) of the LEDs in the +5V and the
    shorter leg (cathode) in OUT(0-15).
  - +5V from Arduino -> TLC pin 21 and 19    (VCC and DCPRG)
  - GND from Arduino -> TLC pin 22 and 27    (GND and VPRG)
  - digital 3        -> TLC pin 18           (GSCLK)
  - digital 9        -> TLC pin 24           (XLAT)
  - digital 10       -> TLC pin 23           (BLANK)
  - digital 11       -> TLC pin 26           (SIN)
  - digital 13       -> TLC pin 25           (SCLK)
  - The 2K resistor between TLC pin 20 and GND will let ~20mA
    through each LED. To be precise, it's I = 39.06 / R (in ohms).
    This doesn't depend on the LED driving voltage.
    A 3k resistor = ~13mA, and 4k is ~10mA
  - (Optional): put a pull-up resistor (~10k) between +5V and BLANK
    so that all the LEDs will turn off when the Arduino is reset.

    If you are daisy-chaining more than one TLC, connect the SOUT of
    the first TLC to the SIN of the next. All the other pins should
    just be connected together:
       BLANK on Arduino -> BLANK of TLC1 -> BLANK of TLC2 -> ...
       XLAT on Arduino  -> XLAT of TLC1  -> XLAT of TLC2  -> ...
    The one exception is that each TLC needs its own resistor
    between pin 20 and GND.
  This library uses the PWM output ability of digital pins 3, 9, 10,
  and 11. Do not use analogWrite(...) on these pins.

  This sketch does the Knight Rider strobe across a line of LEDs.
  Alex Leone <acleone ~AT~ gmail.com>, 2009-02-03
*/

#include "Tlc5940.h"  // remember to include the library!

void setup() {
 /* Call Tlc.init() to setup the tlc.
  * You can optionally pass an initial PWM value (0 - 4095) for all
  * channels.
  */
  Tlc.init();
}
```

```
/* This loop will create a Knight Rider-like effect if you have LEDs
 * plugged into all the TLC outputs. NUM_TLCS is defined in
 * "tlc_config.h" in the library folder. After editing tlc_config.h
 * for your setup, delete the Tlc5940.o file to save the changes.
 */
void loop() {
  int direction = 1;
  for (int channel=0; channel<NUM_TLCS*16; channel+=direction) {

    /* Tlc.clear() sets all the grayscale values to zero, but does not
     * send them to the TLCs. To actually send the data, call
     * Tlc.update()
     */
    Tlc.clear();

    /* Tlc.set(channel (0-15), value (0-4095)) sets the grayscale value
     * for one channel (15 is OUT15 on the first TLC, if multiple TLCs
     * are daisy-chained, then channel = 16 would be OUT0 of the second
     * TLC, etc.). The value goes from off (0) to always on (4095).
     *
     * Like Tlc.clear(), this function only sets up the data,
     * Tlc.update() will send the data.
     */
     if (channel == 0) { direction = 1; }
        else { Tlc.set(channel - 1, 1000); } // end if-else

     Tlc.set(channel, 4095);

     if (channel != NUM_TLCS*16 - 1) { Tlc.set(channel + 1, 1000); }
        else { direction = -1; } // end if-else
    /* Tlc.update() sends the data to the TLCs. This is when the LEDs
     * will actually change.
     */
     Tlc.update();
     delay(75);    // wait to see the effect.
  }// end for();
}// end loop()
```

2.5 Questions about LEDs

The questions that you should ask when connecting LEDs to a microcontroller like the Arduino are:

- *What is the forward voltage (V_f) of the LED that you're trying to use?*

  You can get this value from the vendor of the LEDs, or you can figure it out using a resistor, power supply, and voltmeter. To figure it out on your own, connect the LED through a largish resistor (470Ω for example) from +5v to ground. Measure the voltage drop across the lit LED. This is the forward voltage.

- *How much current should you put through the LED?*

  Your spec for the LED should tell you what the max current capability of your LED is. If it doesn't, assume that 20mA is the maximum. You can always back off a little bit from the LED's max without having much impact on the brightness.

- *What current-limiting resistor should you use?*

You can compute this for a single LED using the formula $R = (V_{source} - V_f)/I_{desired}$ where V_{source} is the power supply voltage, V_f is the LED's forward voltage, and $I_{desired}$ is the amount of current you would like to put through the LED.

If you're using an LED driver chip that has constant current outputs, you need to compute the current-setting resistor according to the data sheet of that chip.

- *How bright would you like your LED?*

  Use `digitalWrite(pin, HIGH/LOW)` to set the LED full on or full off. Use `analogWrite(pin, value)` to set the LED to an intermediate brightness with value being between 0 and 255. This only works on Arduino pins 3, 5, 6, 9, 10, and 11 on the Duemilinove and Uno boards (see `Arduino.cc` for details of other Arduino boards).

- *If you'd like to drive more than one LED from a single microcontroller pin, what series/parallel organization should you use, and what resistors?*

  The easiest approach here is to use an LED calculator from the web like `ledcalculator.net/` and `led.linear1.org/led.wiz`.

- *If you're planning on using more LEDs than one Arduino (or other microcontroller) can support, what external chip should you use?*

  - If you need a small number of extra LEDs, the STP08DP05 is a good choice. It drives 8 LEDs per chip, has constant-current outputs and has a simple shift-register interface that lets you daisy-chain extra chips for even more LEDs. Use `shiftOut()` to send data to the chip(s).
  - If you need more LEDs, or are using seven-segment digit displays the MAX 7219/7221 is a good choice. It also has constant-current outputs and can drive up to 64 LEDs, or 8 digits of seven-segment digit display. It's also daisy-chainable. I recommend using the ledControl library. It's available at `playground.arduino.cc/Main/LedControl`.
  - If you need each LED to have its own brightness control, the TLC5940 is a good choice. The chip drives only 16 LEDs per chip, but has constant-current outputs, 4096 levels of brightness per LED, and you can daisy-chain multiple chips. I recommend using the Tlc5940 library. It's available at `code.google.com/p/tlc5940arduino/`.

Chapter 3
Speed! Sensors

LEDs are a wonderful way to add pizzazz to a project, but they're primarily outputs. It turns out that you can actually use them as inputs by using them backwards as light sensors, but that bit of esoterica aside, they're really output devices that light up. If you want to have your physical computing system react to the environment in which it is installed, you need environmental sensors. An environmental sensor can be as simple as a switch that can be read by the program to see if it is in the on or off position, of medium complexity like a whole range of sensors that change their resistance based on a physical condition like light or temperature, or as complex as something like GPS sensing that receives multiple satellite signals to pin down very precise latitude and longitude. We'll primarily look at the first two styles of sensors.

3.1 Switches

Switches are great environmental sensors: they're simple, they're easy to interface, and they're easy to understand. A simple switch is either open or closed, and that property can result in an electrical signal that is easily interpreted by your program as on or off (or true or false if you prefer).

Some terminology for switches:

- Open vs. Closed
  - A switch is a mechanical device that has conductors that can either be in contact and thus conduct electricity (a closed switch), or be separated and thus not conduct electricity (an open switch).
- Normally Open vs. Normally Closed
  - A switch can be in either state in its quiescent position, and then some action must take place to change to the other state.
- Toggle vs. Momentary action
  - The action that changes the switch state can be static in the sense that once the action is removed the switch stays in that state, or momentary in the sense that when the action is removed the switch reverts to its beginning stage. Think of a

toggle switch vs. a pushbutton with a spring that returns when you release the button.

- Poles and Throws
  - Switches are often defined in terms of these parameters. Poles are the number of separate circuits that are controlled by the same physical switch. Throws are the number of positions that the switch can take.
  - A single-throw switch has one pair of contacts that can be connected or not. A double-throw switch has a common contact that can be connected to one of two other contacts. A triple-throw switch can connect to any of three contacts, etc.
  - A single-pole switch has only one set of connections. A double-pole switch has two completely separate circuits that are switched using the same physical switching mechanism.
  - Some common switch types are shown as schematic symbols in Figure 3.1.

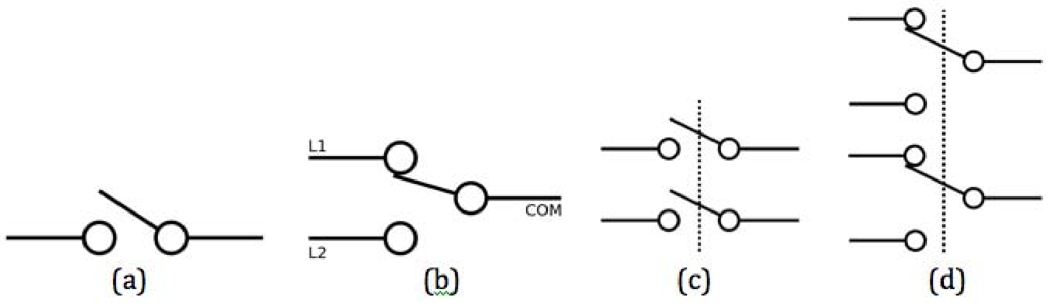


Figure 3.1: Schematic diagrams for four common switch types. They are: (a) SPST or Single pole Single Throw - the simplest switch, (b) SPDT or Single Pole Double Throw where the COM contact can connect to either of the L1 or L2 contacts, (c) DPST or Double Pole Single Throw where there are two separate SPST switches in the same package (the dotted line indicates that the same action causes both switches to change their contact), and (d) DPDT or Double Pole Double Throw.

Switches come in all shapes, size, and configurations. A quick look at any home improvement store, electronics store, or web site will reveal a huge variety of switches. You can also make your own switches out of any conducting and insulating materials. For example, you could make a switch from two pieces of aluminum foil with wires connected to the foil and where contact is made by pressing the foil together. You could make a switch from a hinged piece of wood with thumb tacks placed so that when the mechanism shuts the thumb tacks make contact. You could make a switch from a wind chime where one connection is the metal chime and the other is a metal chimer that hits the chime so that the switch closes when the wind blows. The variations are endless.

The question is, how does a microcontroller like the Arduino connect to a switch and sense whether it's open or closed? Recall that Arduino digital pins can be configured to either `OUTPUT` or `INPUT`. We've used only `OUTPUT` up until now because we've been driving LEDs. Switches are a case where we can use the pins in `INPUT` mode. In this mode, the `digitalRead(pin)` function of Arduino looks at the voltage on a digital pin and returns a Boolean 0 (`LOW`) if the voltage is low, and Boolean 1 (`HIGH`) if the voltage is high. For this purpose, a low voltage is somewhere between 0v and approximately 2v, and a high voltage is somewhere between approximately 3v and 5v.

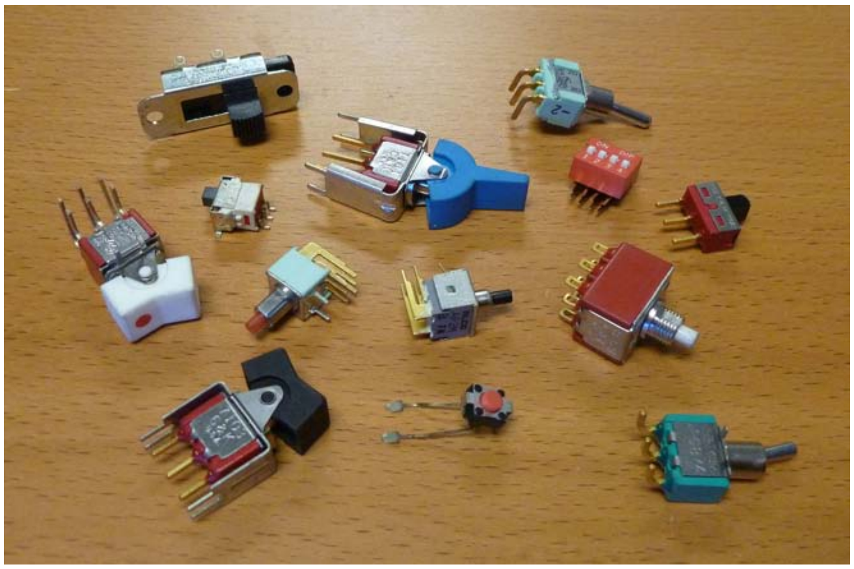


Figure 3.2: A variety of switches. Many of these are SPDT switches with three pins. Some are DPDT with six pins.

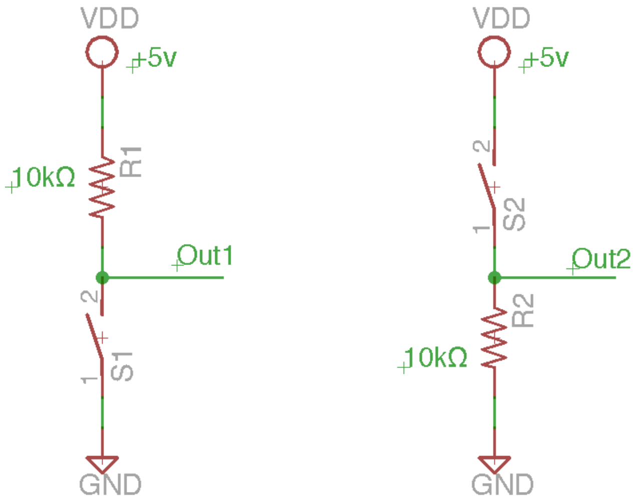


Figure 3.3: Simple SPST switch connections with current-limiting resistors. The configuration on the left has Out1 pulled up to +5v through the resistor (a pullup resistor) when the switch is open, and pulls Out1 to 0v when the switch is closed. The circuit on the right has Out2 pulled to 0v when the switch is open (a pulldown resistor), and pulls Out2 up to +5v when the switch is closed. Out1 and Out2 can be connected directly to an Ardunio digital INPUT pin.

So, to use switches, we need to make a simple circuit that will be around 0v in one position of the switch, and around +5v for the other position of the switch. We also need to be careful about how much current is flowing in that circuit. If you connect a switch directly between the power connection (+5v) and GND, and you close that switch, the low-resistance conductor used in the switch will allow huge currents to flow. Remember Ohm's law: the current is directly proportional to voltage, and inversely proportional to resistance ($V = IR$). So, with very low resistance, you get very high current for a given voltage. This is bad for at least two reasons: one is that high currents burn a lot of power, and the other is that high currents can (because of that power) physically burn up your circuit.

What's needed is our old friend the current-limiting resistor. In this case we need almost no current at all, we just want a voltage difference, so we can use a fairly large resistor that will dissipate almost no power. Typically a 10kΩ resistor is used as shown in Figure 3.3. The figure shows two different versions of the simple SPST switch connection: one where the signal sent to the Arduino is high until the switch is closed, and one where the signal is low until the switch is closed.

Big Idea #12

Like an LED, a switch that goes between power and ground always needs a current-limiting resistor! A typical value is 10kΩ which works well for most situations. In practice, this is a non-critical resistance value: anything between 5kΩ and 100kΩ will likely work.

Some simple code that demonstrates the use of a SPST pushbutton switch in an Arduino program is shown below. The switch is connected using a circuit such as the one shown in Figure 3.3, and is connected to digital pin 2 on the Arduino.

```
// This example code is in the public domain. It's taken from the
//  button  example in the Arduino IDE. It senses the value on the
// external switch and turns the built-in LED on pin 13 on and
// off based on that switch.
// http://www.arduino.cc/en/Tutorial/Button

// constants won't change. They're used here to
// set pin numbers:
const int buttonPin = 2;     // the number of the pushbutton pin
const int ledPin =  13;      // the number of the LED pin

// variables will change:
int buttonState = 0;         // var for reading the pushbutton status

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

void loop(){

  buttonState = digitalRead(buttonPin); // read the state of the switch

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

3.1.1 Arduino Built-in Pullup Resistors

According to Big Idea #12, every switch that makes contact between power (VDD) and ground (GND) must have a current-limiting resistor. This means that you're always looking around for a resistor whenever you connect a switch to one of the digital input pins of the Arduino.

However, if you're using the simple switch schematic on the left hand side of Figure 3.3 you can make use of a nifty feature of the Arduino input pins: they all have built-in pullup resistors that you can enable when the pins are being used as inputs. These built-in resistors are pullups meaning that they connect the pin to Vdd (+5v in this case) through the resistor. The built-in pullup on the ATmega microcontroller that is used on the Arduino Duimilenove and Uno boards has a value of 20kΩ so it's a fine value to use with switches. To enable the internal pullup on a digital pin, declare that pin to be `INPUT_PULLUP` instead of simply `INPUT`. For the example code listed previously, the only line that needs to change is in `setup()`:

```
pinMode(buttonPin, INPUT_PULLUP); // Enable internal pullup resistor
```

With this declaration, you can connect a SPST switch directly to, in this case, pin 2 and ground without the extra resistor. The 20kΩ pullup resistor is enabled inside the Arduino's microprocessor. Other microprocessor chips have similar features.

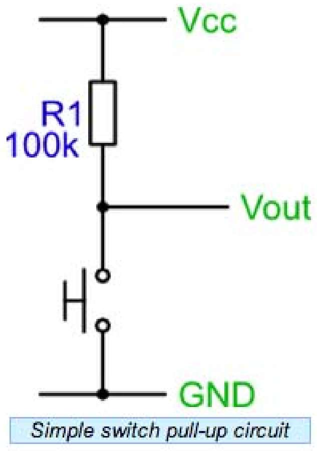
3.1.2 Switch Bounce and Debouncing

Switches are mechanical devices. Inside the body of most switches you'll find small pieces of metal that are moved to become in contact with each other (closed switch), or to not be in contact (open switch). Because these are physical pieces of metal, with mass, and with springiness, they have the unfortunate property that when they are physically brought into contact with each other they can bounce while making contact. This means exactly what it sounds like: the metal bounces up and down while making contact and thus makes and breaks contact a number of times before settling down to a solid connection (or disconnection).

This bouncing happens very quickly from a human's point of view. Typical bounce times range from 0.1ms to 6ms which is pretty fast for a human, but an amazingly long time for a computer. Consider that our Arduinos are running at a relatively slow 16MHz (16,000,000 ticks per second). This is pretty slow from a modern computer point of view where an iPhone 5 has a processor running at 1GHz (1000 MHz). Nevertheless, even at 16MHz, each tick of the Arduino's clock is 16.5ns, and the assembly instructions that are running on Arduino are issued one per clock. So, if a switch bounces for 0.1ms, which is 100us, or 100,000ns, that means that it's bouncing (and thus not a stable high or low signal) for as long as it takes to execute 1600 Arduino assembly language instructions. That's plenty of time for a bouncing switch to be seen as changing many times when the user really only pressed it once!

There are many ways to address the problem of switch bounce. For example you might build a mercury wetted switch where the contact is made with a ball of mercury that makes the desired electrical contact. These are becoming much less common as the hazards of mercury are well known.

An easy solution is simply to wait after the initial indication that the switch has been pressed. If the program that is reading the switch waits after the initially sensed change, and waits longer than the longest anticipated bounce time, this can keep the program from



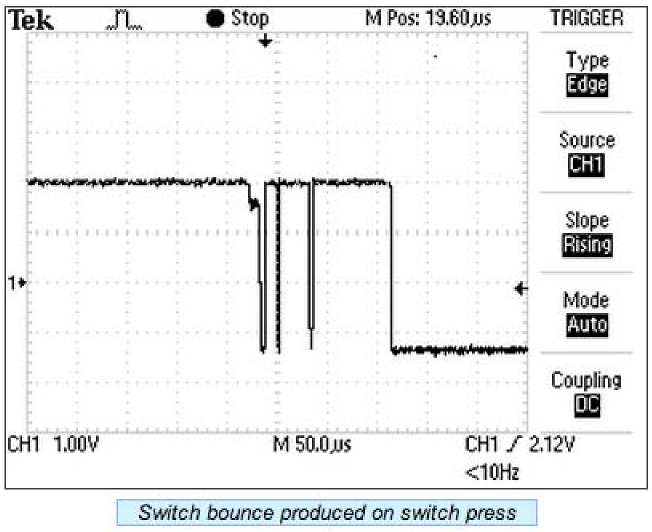


Figure 3.4: A simple SPST switch and the resulting oscilloscope trace showing the effect of switch bounce. The switch will pull V_{out} low when pressed, but there are three distinct bounces before the switch settles into its final position. A microcontroller looking at this signal using `digitalRead()` could easily read this as multiple switch-presses. From Dr. Andrew Greensted `www.labbookpages.co.uk/electronics/debounce.html`

seeing the bounces as extra button presses. The downside of this simple approach is that it delays the reaction to the switch. Code from the Arduino example set for debouncing is as follows.

```
/*
 * This debounce code from the Arduino Examples set. I've added a few
 * additional comments. This code iplements simple debouncing by
 * checking for a change on the button signal (e.g. the button has
 * been pressed or realeased). When the button signal has changed
 * changed state, this code waits for debounceDelay ms before
 * assuming that that state is permanent. Note that it restarts
 * the debounce timer after each change in the signal's state, so
 * this waits for debounceDelay ms after the last signal change that
 * was seen.
 */

// constants won't change. They're used here to set pin numbers:
const int buttonPin = 2;     // 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;  // debounce time; increase if output flickers

void setup() {
  pinMode(buttonPin, INPUT); // button uses an external pullup resistor
  pinMode(ledPin, OUTPUT);   // Drive an LED when the button is pressed
}

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) {
     lastDebounceTime = millis(); // reset the debouncing timer
  }

  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
     buttonState = reading;
  }
  // set the LED using the state of the button
  digitalWrite(ledPin, buttonState);
  // save the reading. Next time through the loop,
  // it'll be the lastButtonState:
  lastButtonState = reading;
}
```

This debouncing code is simple, and reasonably effective. It's not a perfect solution because of the extra delay involved, but it's a pretty short delay in human terms so it's almost always good enough.

Another technique or debouncing that uses extra hardware, but is much faster at sensing the initial flipping of the switch, uses a set-reset flip flop and a single-pole, double throw (SPDT) switch. This circuit, shown in Figure 3.5, is a little tricky to understand. The set-reset flip flop is implemented with the cross-coupled NAND gates (the circuits with the round fronts and the bubbles on the outputs). A NAND gate is a logical gate that will drive its output to logic-1 (+5v) whenever either of the inputs is low, and drive its output to logic-0 (0v) only when both inputs are high.

By cross-coupling the NAND gates in this configuration they form a flip-flop: a circuit that can store one bit of information. Without going into too much detail, the key to this design is that when the common connection of the SPDT switch is in between the output contacts, both of the signals coming from the switch will be pulled high by the pullup resistors. Only when the switch has settled into one of its two main (non-moving) positions will one of the outputs be pulled low. When this happens, the set-reset (SR) flip-flop will be either set or reset depending on which wire is pulled low. If the switch bounces, it simply sets (for example) a bunch of times in a row, but after the first set the output of the flip-flop (Out) will not change. It's only when the flip-flop is reset that the output will change to a low signal. And in that case, a series of reset signals will cause it to reset, and multiple reset signals (a bouncing reset) will not have any visible impact on the Out signal.

This circuit completely solves the bouncing problem, but at the expense of using a SPDT switch, two pullup resistors, and two NAND gates. NAND gates come four-to-a-package when implemented as a chip, but that's still a lot of extra circuitry. If you're willing to use a SPDT switch, and use two digital inputs on the Arduino instead of one, you can simulate this flip-flop approach in software, and leverage the built-in pullup resistors in the Arduino's digital inputs. This uses two input pins for a single non-bouncing input,

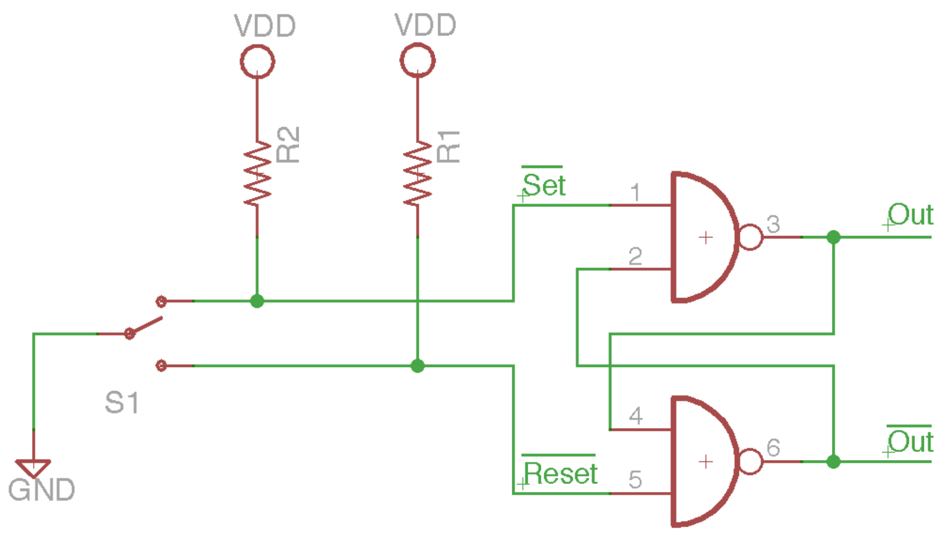


Figure 3.5: A debouncing circuit using a set-reset flip-flop made from NAND gates, and a single-throw, double pole (SPDT) switch with pullup resistors. The SPDT switch has its common signal connected to ground, and the two outputs (one from each throw) pulled up through pullup resistors. These form the Set and Reset signals to the flip-flop. Connect the Out signal to one of the digital inputs f the Arduino and you'll have a non-bouncing signal that you can sample with `digitalRead()`.

but uses only a switch and a ground connection external to the Arduino. If you need a non-bouncing signal and have pins to spare, this is a good solution. In this case you would connect the Set and Reset signals in Figure 3.5 directly to two of the digital inputs of the Arduino. Some example code is seen below:

```
/* This code emulates the cross-coupled NAND style of debouncing
 * a SPDT switch using a flip-flop.
 */
const int switch_set = 2;   // The Set input from the switch
const int switch_reset = 3; // The reset input
const int ledPin = 13;         // The built-in LED pin
boolean state = LOW;        // the state of the switch

void setup() {
  pinMode(switch_set, INPUT_PULLUP);   // Enable pullups on
  pinMode(switch_reset, INPUT_PULLUP); // the switch inputs
  pinMode(ledPin, OUTPUT);             // drive the LED
}

void loop() {
  // read the values coming in from the switch. Note that these
  // are active-low values because they're being pulled to
  // ground by the switch. So, the action (such as "set") happens
  // when the incoming value is low. Because of the physical
  // construction of the SPDT switch, you'll never get both
  // switch_set and switch_reset being low at the same time.
  if (digitalRead(switch_set)   == 0) state = HIGH;
  if (digitalRead(switch_reset) == 0) state = LOW;

  // Now use the "state" value as the debounced value coming
  digitalWrite(ledPin, state); // light up LED if state is HIGH
}
```

3.1.3 Aside: Serial Communications with the Host Machine

In Section 1.5.2 there was a quick overview of some useful functions that are pre-defined by the Arduino IDE. One of them is the serial monitor. This is a way that your Arduino can communicate through the serial connection (the UART) to the host machine that you're connected to. This serial monitor can be used for a wide variety of things, but one of the most useful is for calibration and debugging of Arduino programs. The serial monitor lets your program print things to the screen. This is easy if you're writing a C++ program in Linux, but not as easy if your program is running on an embedded controller like the Arduino that has no operating system, and more importantly, no display.

To use the serial monitor you first initialize the communication in the setup() function using the Serial.begin(baud) function. The baud argument sets the baud rate (serial bit rate) for the communication. Because this uses standard EIA232 serial protocols, your choice of baud rate must be one of 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200. You must also make sure that the serial monitor that you start from the Arduino IDE is set to the same baud rate as your Arduino program is using.

Once your serial connection is initialized in the `setup()` function, you can print to the monitor using the `Serial.print(arg)` and `Serial.println(arg)` commands. The only difference between them is that the second prints a newline after the argument is printed to the monitor.

Once the code is uploaded and running on the Arduino hardware, you can start the serial monitor on the host using the Arduino IDE. You can either use the menus with Tools → SerialMonitor or you can use the Serial Monitor icon in the upper right of the IDE's main screen.

The following is a simple program that uses a button, an internal pullup for the button, and the serial monitor instead of an LED to indicate the button's state.

```
 /*
 * Read the state of a button and print it to the Serial Monitor
 */

int pushButton = 2;  // button is connected to digital pin 2

void setup() {
  // initialize serial communication at 9600 bits per second
  Serial.begin(9600);
  // make the pushbutton's pin an input using a built-in pullup
  pinMode(pushButton, INPUT_PULLUP);
  // print a header to the serial monitor
  Serial.println( Testing a button using the Serial Monitor );
}

void loop() {
  // read the input pin
  int buttonState = digitalRead(pushButton);

  // print out the state of the button
  Serial.print( Button state is:  );
  Serial.println(buttonState);       // buttonState will be 0 or 1

  delay(100); // delay between reads
}
```

This program continuously prints the state of the button on the serial monitor. The delay means that the serial monitor will get an update (a new value will be printed) every 100msec (10 times a second). That will still scroll things up the serial monitor pretty quickly. The following example will print something only when the program sees the state of the button change. You could use this to see, for example, if your program is seeing any bouncing with the button. If so, you'll see multiple button press messages when you press the button once (or flip the switch if you're using a toggle switch). Note that there's no delay in the loop here so the program will check the button as fast as it can, and should show the effect of a bouncing switch.

```
 /*
 * Print a message on the Serial Monitor whenever you see the state
 * of the switch change.
 */

int swPin = 2;          // switch is connected to pin 2
int val;                // variable for reading the pin value
int swState;            // variable to hold the last switch state

void setup() {
  pinMode(swPin, INPUT_PULLUP); // use internal pullup on switch
  Serial.begin(9600);           // Init serial communication - 9600bps
  swState = digitalRead(swPin); // read the initial value
}

void loop(){
  val = digitalRead(swPin); // store switch value in val

  if (val != swState) {     // the switch state has changed
     if (val == LOW) {      // check if the button is pressed
        Serial.println("The button was just pressed");
     } else {               // the button is not pressed
       Serial.println("Button just released");
     } // end if(val == LOW)
  } // end if (val!= swState)

  swState = val;  // save the new switch state
}
```

3.1.4 Another type of Switch: Passive Infrared (PIR) Motion Detector

Toggle and pushbutton switches, and homemade contact switches are great sources of human input to your projects. Another form of switch that doesn't require physical contact is a passive infrared (PIR) motion detector. These sensors are used, for example, by alarm systems to sense whether anything is moving around in a room. They're a great way to cause something to happen when a person or animal has entered the vicinity of the project.

A PIR sensor is a reasonably complex sensor. If you look on the back of one you'll see a number of integrated components. What the sensor is doing is taking two readings of infrared (IR) light and comparing them. If the values ever differ, it assumes that something warm has moved in its field of view and it raises the voltage on the output pin. If nothing changes after some amount of time, it lowers that signal. The Fresnel lens allows a whole hemisphere map onto those two regions of interest.

From your microprocessor program's point of view this looks just like a switch. You can read the value on the P0 signal (Figure 28) using `digitalRead()` and use a HIGH reading to indicate that something is moving. Your sensor may have a different pin connection than the one shown here. It may even have an open drain output that requires a pullup resistor just like a toggle switch. There may also be ways to adjust the sensitivity and timeout

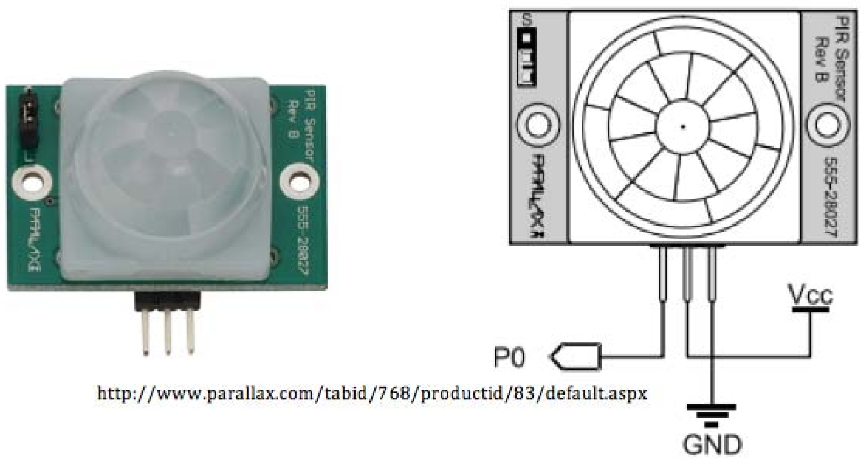


Figure 3.6: A passive infrared (PIR) sensor. The milky white plastic piece on the front is a Fresnel lens that focuses infrared radiation (heat) onto the sensor. That lens modifies a physical sensor that looks only in one direction and allows it to see into an entire hemisphere. The three pins at the bottom of the right-hand image are the connection pins: signal, power (3v to 6v DC), and ground. The P0 pin in this figure will be low until motion is sensed. That pin goes high when motion has been observed by the sensor.

(the time it takes to reset after movement has been detected) of your sensor. Read the information that came with your sensor carefully.

A simple program to test a PIR sensor using the serial monitor and the Arduino's built-in LED as indicators is shown here. This simple PIR test program is from Limor Fried (Lady Ada) and can be found with more details about PIR sensors at `www.ladyada.net/wiki/tutorials/learn/sensors/pir.html`.

```
/*
 * PIR sensor tester
 */

int ledPin = 13;          // choose the pin for the LED
int inputPin = 2;         // choose the input pin (for PIR sensor)
int pirState = LOW;       // we start, assuming no motion detected
int val = 0;              // variable for reading the pin status

void setup() {
  pinMode(ledPin, OUTPUT);        // declare LED as output
  pinMode(inputPin, INPUT);       // declare sensor as input
  Serial.begin(9600);             // init serial connection for output
}

void loop(){
  val = digitalRead(inputPin);  // read input value
  if (val == HIGH) {            // check if the input is HIGH
    digitalWrite(ledPin, HIGH); // turn LED ON

    if (pirState == LOW) {// we have just turned on
      Serial.println("Motion detected!");
      // We only want to print on the output change, not state
      pirState = HIGH;
    } // end if (pirstate == LOW)
  } else { // (val != HIGH)
    digitalWrite(ledPin, LOW); // turn LED OFF

    if (pirState == HIGH){ // we have just turned off
      Serial.println("Motion ended!");
      // We only want to print on the output change, not state
      pirState = LOW;
    } // end if(pirState == HIGH}

 } // end else (val != HIGH)
}// end loop()
```

3.2 Resistive Sensors

Switches are great sensors. They're easy to use, easy to sense, and easy to install both physically and electrically. But, they have only two states: on and off. There is another class of sensors that can return information over a whole range of values. The easiest to use of these sensors are the so-called resistive sensors. These sensors act like resistors, but with the feature that based on some environmental condition they will change their resistivity. This change in resistivity can be sensed by your program and used to change program behavior based on a range of conditions.

Because these sensors react to conditions by changing their resistance, in order to use them you must be able to sense a change in resistance. The easiest way to do this is with a voltage divider circuit (see Big Idea #6 in Section 1.2). To review, a voltage divider is a circuit that has two resistors in series. Kirchhoff's current and voltage laws tell us that both resistors will see the same current, and the entire voltage from the power supply to ground will be dropped across the series-connected device. Furthermore, Ohm's law tells us that the magnitude of the voltage drop will be proportional to the resistance of the components.

If you build a voltage divider with two resistors, but one of the resistors is a variable resistor, then the voltage "divided" in the middle of the circuit will change proportionally to the changing resistance. In Figure 3.7 there are two variations on this theme: one where the OUT signal goes to Vdd (+5v in our case) as the variable resistance goes towards 0Ω, and one where OUT goes to 0v in that case. All that's needed now are to find sensors that behave like variable resistors, and to figure out how to sense the continuously variable voltage that results from a voltage divider under changing resistance situations.

3.2.1 Potentiometers

A potentiometer is a variable resistor where the resistance is changed by turning a knob. These types of components are everywhere: volume knobs and light dimmers are two extremely common examples. The sliding controls you see on music mixing boards are an example of a linear style potentiometer. A potentiometer (called a pot for short) is a piece of resistive material that is connected just like a regular resistor. The distinguishing feature is that there's third terminal that is connected to a wiper that can move up and down, or around, that material and make contact partway through the resistor. The resistance between the endpoints is fixed. The resistance seen at the third terminal varies by how far you've turned the knob, or how far you've moved the slider.

The behavior of a potentiometer allows a particularly simply technique for connecting it to a microprocessor like Arduino. As seen in Figure 3.9, the pot can be connected with

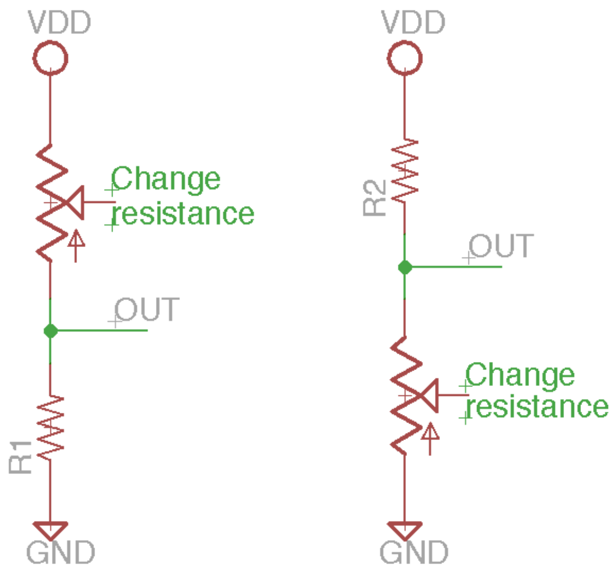


Figure 3.7: Two variations on a voltage divider with a variable resistance element. In the circuit on the left, if the resistance of the variable resistor goes to 0Ω, OUT goes to VDD. If the resistance is extremely high, OUT becomes close to 0v. The circuit on the right is the opposite.

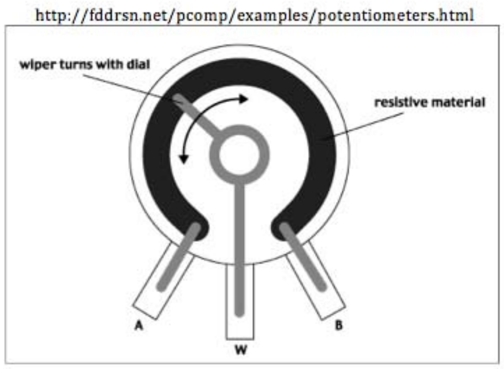


Figure 3.8: A cutaway drawing of a potentiometer. The resistance between terminals A and B is always fixed because the signal goes through all of the resistive material. As you turn the knob, the wiper goes further to the left or right. As it does, the relative resistance between A and W changes, as does the resistance between W and B. If the wiper is all the way counterclockwise, then the resistance between A and W is very low (essentially 0Ω) and the resistance between W and B is the max amount of resistance for the pot. If the wiper is turned all the way clockwise, the result is opposite.

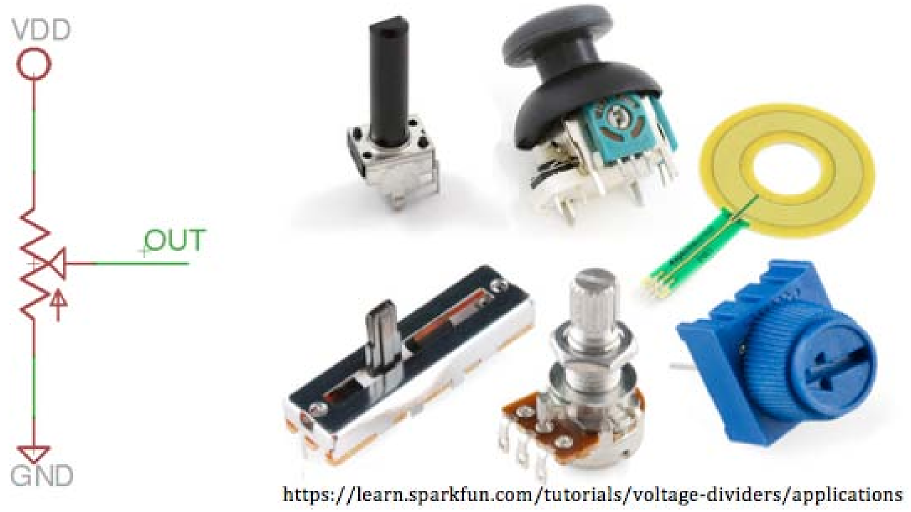

VDD
OUT
GND
https://learn.sparkfun.com/tutorials/voltage-dividers/applications

Figure 3.9: A standard potentiometer connection with the endpoints connected to VDD and GND, and the output being read from the wiper. Potentiometers come in a huge variety of shapes, sizes, and resistance values. Shown in the right of this figure are just a few different examples.

Vdd (+5v) at one end, and GND at the other. Because the resistance is fixed at the end terminals of the pot, the only concern is whether there is enough total resistance in the pot to avoid a meltdown (current-limiting). Any pot with a total resistance of 10kΩ or more should be fine. Now if you turn the knob (or move the slider) you'll be changing the point where you're dividing the resistance, and thus see a voltage that varies between Vdd and GND on that OUT signal.

The remaining question is how to connect the OUT signal (wiper) of the pot to the Arduino in a way that your program can sense the entire range of voltages that are generated by the pot. This is where the analog inputs of the Arduino are used. The analog inputs are connections to an internal circuit on the Arduino's microprocessor called an Analog to Digital Converter (ADC). An ADC takes a continuously varying voltage across some range and converts it into a digital number, again with some range.

In our case the ADC has 10 bits of resolution. That is, it can take the range of voltages that it's designed for (0v to +5v in our case) and sense intermediate voltages in that range in 1024 increments. That is, it can sense 1024 steps in voltage between 0v and 5v. Another way to think about this is that it can sense approximately 4.9mV (0.0049v) changes in the analog signal. The practical effect is that you can put a voltage that varies between 0v and 5v into an analog input, and get back an integer between 0 and 1023 that tells you what that voltage is in 4.9mV steps. The Arduino function to use is `analogRead(pin)` where the analog pin is A0, A1, A2, A3, A4 or A5. Plain integers 0 through 5 also work as identifiers for the analog pin being read.

Some simple Arduino code that reads an analog sensor (like a potentiometer) and uses it to adjust the brightness of an LED is shown below. Note that `analogRead()` reads the value of the analog pins. `analogWrite()` sends a PWM value to a digital output pin. They sound similar, but they work on completely different sets of pins on the Arduino.

```
/*
 * Read the analog value coming from a potentiometer (the sensor)
 * and use that value to fade an LED using analogWrite().
 */

int sensorPin = A2;  // analog pin used to connect the potentiometer
int ledPin = 9;      // digital pin for an external LED (a PWM pin)
int val;             // variable to read the value from the analog pin

void setup() {
  pinMode(ledPin, OUTPUT); // define ledPin as output
}

void loop() {
  val = analogRead(sensorPin);      // reads the sensor (value 0-1023)
  val = map(val, 180,620, 0, 255); // scale it between 0 and 255
  val = constrain(val, 0, 255);    // keep it in the right range
  analogWrite(ledPin, val);        // fade the LED
  delay(15);                       // wait a bit
}
```

For an example of a physical connection of a potentiometer to an Arduino see Figure 3.10. In this figure the endpoints of the pot are connected to +5v and GND, and the center tap (the wiper) is connected to analog pin A2. This physical connection is the same as the schematic in the left part of Figure 3.9.

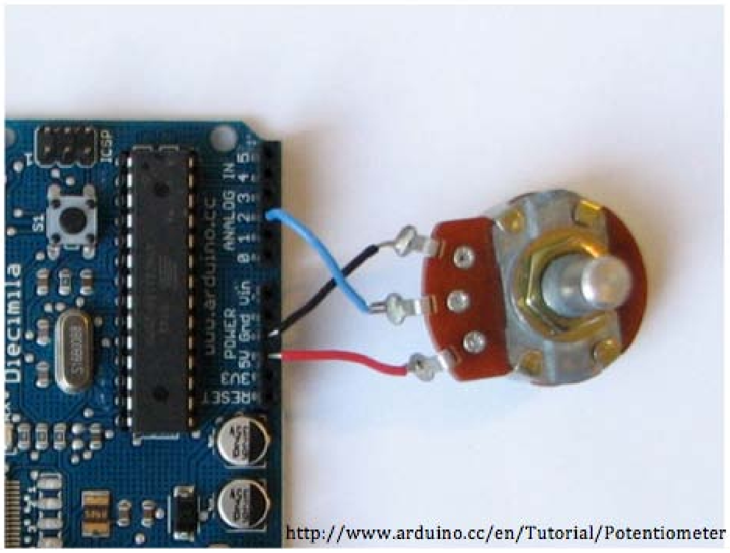


Figure 3.10: A potentiometer connected to an analog input of the Arduino. In this case the endpoint of the pot are connected to +5v (red wire) and GND (black wire), and the wiper of the pot is connected to analog pin A2 (blue wire).

3.2.2 Other Resistive Sensors

A potentiometer is in some sense the most fundamental resistive sensor. It is also the most physical: it is sensing how much the knob is turned or the slider is moved. There are other resistive sensors that react to environmental conditions without physical contact or movement. For example, a Cadmium Sulfide (CdS) light sensor is a component that changes its resistance depending on how much light falls on the sensor. These sensors often have a flat-topped look with a squiggly line in the surface of the sensor. That squiggly line is the CdS light sensitive material. It is this material that changes its resistance when light on it. It has a very high resistance in the dark, and has a much lower resistance when light is shining on it. Typical values might be 200kΩ in the dark and 10kΩ in bright light, but these cells vary tremendously so it's a good idea to test them using a resistance meter.

These sensors can take the place of the variable resistors in Figure 3.7. You can use them in either configuration as seen in Figure 3.11. Connect the OUT signal from the voltage divider to one of the analog inputs of the Arduino and read the value using `analogRead(pin)`. Because the CdS cell has high resistance in the dark and lower resistance in the light, the OUT value for the left hand circuit in Figure 3.11 will be low in the dark and high in the light. The right hand circuit will be opposite: OUT will be high in the dark and lower in the light. Some CdS light sensors in various sizes are shown in Figure 3.12

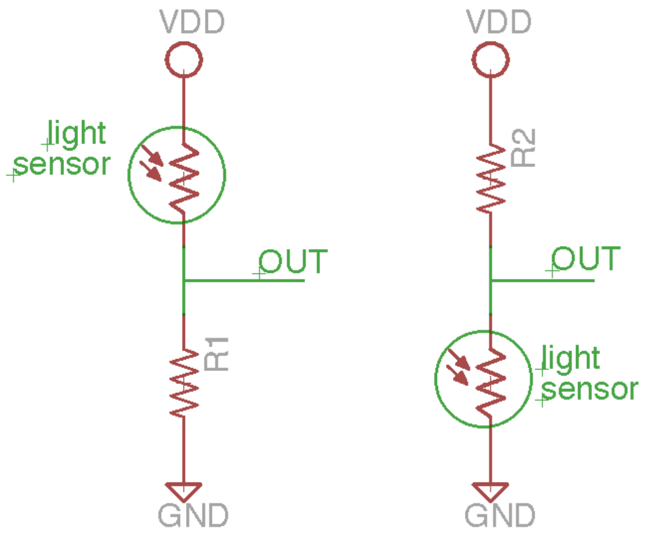


Figure 3.11: CdS light sensors used in a voltage divider with a current-limiting resistor. The value of the fixed resistor should be sized so that if the CdS sensor's resistance gets close to 0Ω, the current from VDD to GND is limited to a safe, low value. In practice a 10kΩ resistor is a good place to start.

There are a huge range of resistive sensors that behave just like CdS light sensors. That is, they react to some environmental condition and change their resistance accordingly. They can all be connected in the same way as the CdS light sensors in Figure 3.11. The only issue to be careful of is to know or check the range of possible resistances that the sensor can take on. If the sensor can take on a value close to 0Ω then the fixed resistor must be large enough to limit the current to a safe level. Like a switch (which can take on a value very close to 0Ω when the switch is closed) a 10kΩ resistor is a good starting point. If the sensor can't go below 10kΩ on its own, then a separate fixed resistor may not be necessary.

Environmental sensors that react by changing their resistance include:

Temperature: known as thermistors, these sensors typically lower their resistance as temperature rises.

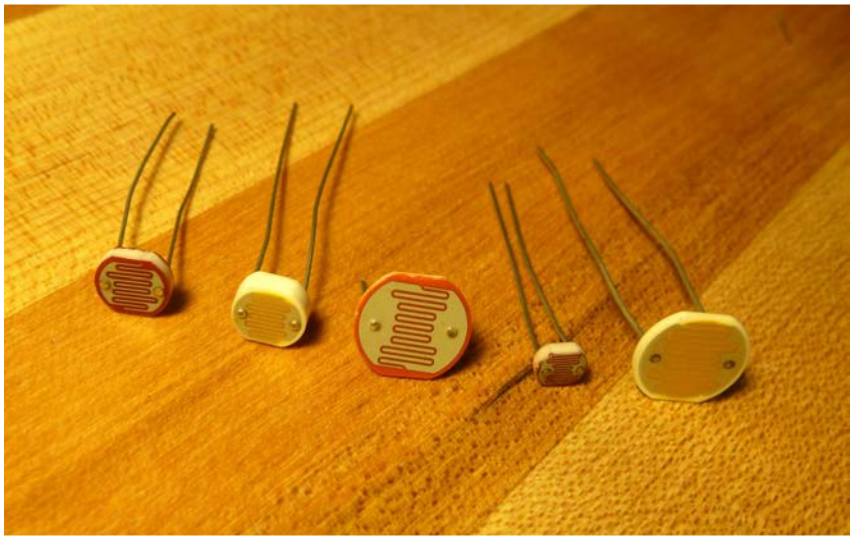


Figure 3.12: Some CdS light sensors in various sizes. It's difficult to determine the exact range of resistances these sensors will take on. You'll need to measure them with an ohmmeter or by using a calibration program (see Section 3.4)

Flex: This ribbon-like sensors change their resistance when bent.

Pressure: These sensors react to force applied directly to the sensor.

Gas: These sensors change their resistance based on concentrations of various gasses in the atmosphere. Speciality sensors for Alcohol, $C0_2$, CO, Methane, Hydrogen, and other gasses are common.

3.3 Analog Sensors

Resistive sensors are easily used in a voltage divider configuration. By putting them in a voltage divider you are generating an analog voltage that changes as the sensor changes. There is another class of sensor that generates an analog output signal directly without having to be connected as a voltage divider. These analog-output sensors can be connected directly to an analog input of the Arduino and be read using the same analogRead(pin) function as used with a voltage divider.

A great example of an analog sensor is the Sharp IR Distance sensor. This sensor, shown in Figure 3.13, sends an IR signal out of one side, and senses the return on the other. It uses this information to sense the closest object (that reflects IR) to the sensor. The sensors come in a range of sizes that detect objects in different distance ranges. The sensor shown in Figure 3.13, for example, senses objects in the range of 20cm to 150cm (approx 8in to 60in). Other Sharp sensors have different ranges including very short distances from the sensor (10cm or less). The interface is three wires: power (+5v), ground, and output. The output in this case is an analog voltage that ranges from approx 2.7v at 20cm to approx 0.4v at 150cm. This output signal can be connected directly to an analog input pin on the Arduino where an `analogRead(pin)` will result in an integer approximately in the range of 550 (2.7v) to 80 (0.4v). Unfortunately, a linear change in distance sensed does not result in a linear change in output voltage. The curve from the Sharp data sheet is shown in Figure 3.13. Some mapping code in software, perhaps something as simple as a lookup table, will be required to really decode distance.

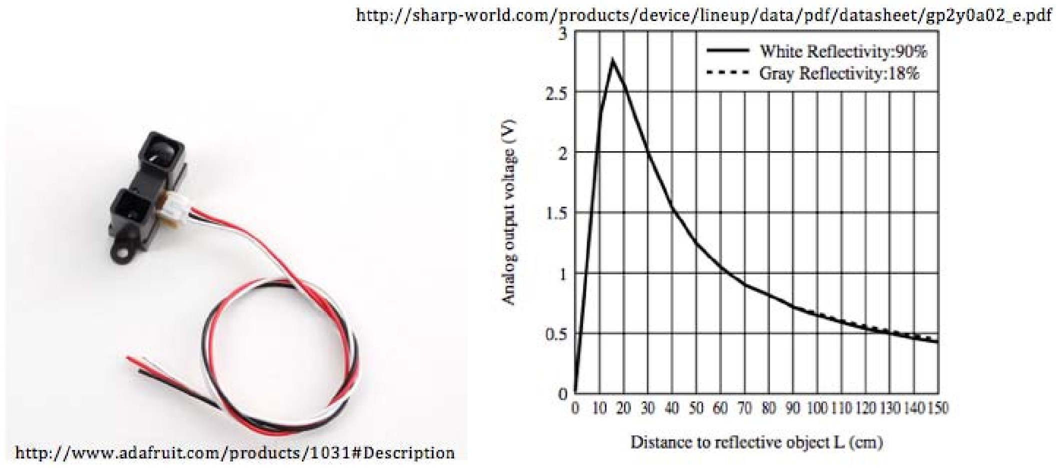


Figure 3.13: A Sharp IR distance sensor. The sensor's interface is three wires. The red is Vdd (+5v), black is ground, and the white wire is the analog output that varies depending on the sensed distance to the closest object. The output voltage varies with this distance according to the curve on the right.

Sharp distance sensors are great devices, especially for short distances. However, if you'd like to sense longer distances, a sonar-based sensor like those made by Maxbotics are a good choice. These operate by sending a sonar ping and sensing the returned signal with a small microphone. They are interfaced in exactly the same way as the Sharp sensor: power ground and analog signal. An example of this type of sensor is seen in Figure 3.14. This sensor operates over a wider range of power supply voltage (2.5v to 5v), over a longer distance (15cm (6in) to 645cm (254in)), and with a more linear analog behavior (Vdd/512v per inch). They are a little more expensive than the Sharp IR sensors, and not effective at short ranges under 15cm.

As with the resistive sensors, there are a wide variety of sensors that return an analog voltage that relates to the property being sensed. Some examples include:

Acceleration: You can get 1-axis, 2-axis, and 3-asix accelerometers that return an analog voltage that varies with the current acceleration in that axis' direction.

Gyroscopes: Gyros tell you what their current orientation are in 2- or 3-dimensions. This information can be returned as an analog voltage.

Color: This is similar to a light sensor, but returns different analog voltage levels that encode each of the Red, Green, and Blue components of the light falling on the sensor.

3.4 Sensor Calibration

Many sensors have a wide variation in their behavior. A CdS light sensor, for example, might vary in its range of light and dark resistance by more than 50% even within the same

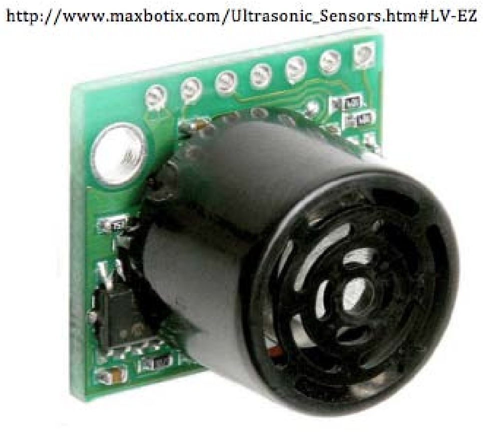


Figure 3.14: : Maxbotix sonar sensor. This sensor returns an analog voltage that varies as Vdd/512v per inch.

size and batch of sensors. Even if you know exactly what range of resistances you will see with a particular sensor, the behavior will be very different depending on the light levels in the environment in which the sensor is installed. A perfectly working system in one room might not react at all in a different room with a different light profile.

Other sensors such as the IR distance sensors have nominal voltage/distance curves, but will have noticeably different behavior at different temperatures, or with small power supply voltage variations. Because of these variations in behavior, it's always a good idea to calibrate your sensors. Calibration in this context simply means to run your program in a mode where the range of sensor values is reported so that you can modify your system to react to the range of values that you'll actually see.

The easiest way to calibrate is simply to use the serial monitor to print the range of values that you're seeing from your sensor. A simple "print statement" calibration program is seen below.

```
/*
 * This code is based on the readAnalogValue code in the Arduino
 * example set. It reads the analog value coming in on an analog
 * input and prints it to the serial monitor, along with an estimate
 * of the actual voltage that the ADC is reading.
 */

int sensorPin = A0; // which pin are you reading sensor data on?

void setup() {
  Serial.begin(9600); // initialize the serial connection
}

void loop() {
  int sensorValue = analogRead(sensorPin); // read analog pin
  float voltage = sensorValue * (5.0 / 1023.0); // convert to a voltage

  // print the values
  Serial.print( Analog value and voltage are:  );
  Serial.print(sensorValue);
  Serial.print(  ADC value,  );
  Serial.print(voltage);
  Serial.println( v );
  delay(100);            // delay for a bit
}
```

This form of manual calibration is simple, but effective. It's easy to build this in to your program, perhaps as a separate mode that runs when a switch is flipped. It could also be a separate calibration program that you run when you are concerned that conditions might have changed. Based on the values you see printed on the serial monitor, you can update `map()` functions or other pieces of interpolation in your program.

Another technique for calibration is to automatically calibrate by entering a program mode that collects data from the sensor and that records the high and low values seen. After that phase those high and low values can be used to interpolate the remaining data. The Arduino IDE includes an example of this sort of self-calibration. That code is repeated here:

```
/*
  Calibration
 Demonstrates one technique for calibrating sensor input.  The
 sensor readings during the first five seconds of the sketch
 execution define the minimum and maximum of expected values
 attached to the sensor pin.
 The sensor minimum and maximum initial values may seem backwards.
 Initially, you set the minimum high and listen for anything
 lower, saving it as the new minimum. Likewise, you set the
 maximum low and listen for anything higher as the new maximum.
 The circuit:
 * Analog sensor (potentiometer will do) attached to analog input 0
 * LED attached from digital pin 9 to ground
 created 29 Oct 2008
 By David A Mellis
 modified 30 Aug 2011
 By Tom Igoe
 http://arduino.cc/en/Tutorial/Calibration
 This example code is in the public domain.
 */

// These constants won't change
const int sensorPin = A0;    // pin that the sensor is attached to
const int ledPin = 9;        // pin that the LED is attached to

// variables:
int sensorValue = 0;         // the sensor value
int sensorMin = 1023;        // minimum sensor value
int sensorMax = 0;           // maximum sensor value

void setup() {
  // turn on LED to signal the start of the calibration period:
  pinMode(13, OUTPUT);
  digitalWrite(13, HIGH);

  // calibrate during the first five seconds
  while (millis() < 5000) {
     sensorValue = analogRead(sensorPin);
     // record the maximum sensor value
     if (sensorValue > sensorMax) {
        sensorMax = sensorValue;
     }
     // record the minimum sensor value
     if (sensorValue < sensorMin) {
        sensorMin = sensorValue;
     }
  }
  // signal the end of the calibration period
  digitalWrite(13, LOW);
}

void loop() {
  // read the sensor:
  sensorValue = analogRead(sensorPin);
  // apply the calibration to the sensor reading
  sensorValue = map(sensorValue, sensorMin, sensorMax, 0, 255);

  // in case sensor value is outside the range seen during calibration
  sensorValue = constrain(sensorValue, 0, 255);
  // fade the LED using the calibrated value:
  analogWrite(ledPin, sensorValue);
}
```

3.5 Questions about Sensors

Some questions that you should ask when considering sensors for use with a controller like the Arduino are:

- *How many digital and/or analog pins are available to use?*

  Switches, PIR motion sensors, and other switch-style sensors use digital pins. Resistive and analog sensors use analog pins. There are external chip solutions for expanding the number of both types of pins, but without resorting to those external chips, there are a limited number of each of these pins (14 digital pins and 6 analog pins).

- *If you're using a switch, what type of switch do you need?*

  The most common switches are single pole, single throw (SPST) and single pole, double pole (SPDT). Make sure you know what the connections are. Use a voltmeter configured as a continuity tester if you're not sure. Almost all voltmeters have a continuity setting. In this setting you attach the two leads of the tester to the terminals that you're interested in. If the meter buzzes, the terminals are connected. If it makes no sound, they are not connected.

- *Will your sensor need debouncing?*

  Many switch-type sensors will bounce. There may be situations where you don't care about occasional extra bounces being read as separate events. But if you do care, and there is a possibility of bouncing, you need to consider a debouncing strategy.

- *Do you need a manufactured switch, or can you make your own?*

  Switch functionality can be made of almost any conductive material. There are many creative ways to make something that acts like a switch.

- *If using a digital input with a switch of some sort, how much current will be drawn when the switch is closed?*

  Any time there might be a connection between power and ground, you should always ask whether a current-limiting resistor is needed, and if so, what value of resistor is needed? Standard switches make a very low-resistance connection when closed so a 10kΩ or larger resistor is typically used. Other switch types, or homemade switch devices might have different closed-resistances.

- *If using a resistive sensor in a voltage divider, what should the fixed resistance be?*

  This is yet another current-limiting question. You want to make sure that at the limits of the variable resistor that not too much current will flow from power to ground. The tradeoff is that if you size the fixed resistor to be too large, then the range of voltages that you'll see from the voltage divider will be restricted, so you won't get as much resolution. If you know the range of resistances that your sensor will produce you can size the fixed resistor so that the entire voltage divider has 5kΩ to 50kΩ in the lowest resistance situation.

- *Potentiometers are great sensors and are easy to use. Is there a way to use a potentiometer to sense what you need to sense?*

  There are lots of ways to connect things to pots to sense their position. Remember the classic arcade driving games with steering wheels? Those were simply connected to pots to see how far the user turned the wheel. A linear pot (slider) is a great way to sense movement in on dimension.

- *If using an analog sensor, what range of analog voltages will you see at the analog inputs?*

  Knowing the range of possible voltages, either through the data sheet or through calibration, can enable you to size your `map()` function to get optimal range out of your sensor.

- *Can you build a calibration mode into your application?*

  Calibration is very important when using sensors. If there's any way to build a calibration mode into your application it will make your life much easier. Don't assume that the values you see the first time you run your system will be the same over the life of the system, or when you take your system to alternate locations.

- *How frequently do you need to check your sensor's value?*

  Your Arduino is running at 16MHz (16,000,000 cycles/sec.). That means that it can execute a new assembly language instruction on every clock tick. Of course, a C++ instruction might take multiple assembly instructions to execute, but that's still a lot of instructions in human terms per second. If you are reading a sensor that operates on a human time scale, you may not need to check it as frequently as you think in order to get good data.

  That being said, using a `delay(ms)` instruction is a busy-wait. The processor is not doing anything useful other than counting down until the delay is finished. If there are other things that your application can be doing, you should use the millisecond counting technique to decide when to check the sensor again. A simple example of this follows (from the Arduino blinkWithoutDelay program). More advanced solutions would use timer interrupts, or interrupts based on external events, to trigger activities. Those techniques will not be covered in this course. Ideas about using interrupts can be found on the Arduino web site, and other places on the web.

```
const int ledPin =  13;     // the number of the LED pin
int ledState = LOW;         // ledState used to set the LED
long previousMillis = 0;    // will store last time LED was updated

// the follow variables is a long because the time, measured in
// miliseconds, will quickly become a bigger number than can be stored
// in an int.
long interval = 1000;      // interval at which to blink (milliseconds)

void setup() {
  pinMode(ledPin, OUTPUT); // set the digital pin as output:
}

void loop() {
/*
 * Here is where you'd put code that needs to be running all the
 * time.
 */

  // check to see if it's time to blink the LED; that is, if the
  // difference between the current time and last time you blinked
  // the LED is bigger than the interval at which you want to
  // blink the LED.
  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis > interval) {
     // save the last time you blinked the LED
     previousMillis = currentMillis;
     // if the LED is off turn it on and vice-versa:
     if (ledState == LOW) ledState = HIGH;
       else ledState = LOW;
     // set the LED with the ledState of the variable:
     digitalWrite(ledPin, ledState);
  }
}
```

- *How much power does your sensor consume?*

  All the sensors described in this course are reasonably low-power sensors that shouldn't tax the total power available from your Arduino's voltage regulator. But, there are certainly sensors out there that use more power. A miss-sized resistor can also consume extra power. If you do have a high-power sensor, you may need to use a separate power supply. Remember to connect the grounds of all separate power supplies together.

Chapter 4
Action! Motors

We've looked at lights (LEDs), and inputs (sensors). The next big piece of physical computing, in many ways the essence of physical computing, is movement. The easiest way to make things move is with a motor of some sort. There are other ways of causing movement, but motors are inexpensive, and easy to use. They do take a bit of planning though, especially in terms of power. Making anything significant move will take a lot more power than an Arduino board can provide. The general strategy is to have the controller (like Arduino) control the movement, but have the power for the motors come from a separate power supply. In fact, the power supply that is powering the motor can often also be used to power the Arduino.

4.1 Hobby Servos

Hobby servos are designed to easily move small objects through a controlled range of motion. They are an example of a motor connected to a servo-mechanism which means that there is some sort of feedback involved in the system that can be used to control the position of the motor. In this case the servo's motor is sent through some gears to reduce the range of motion (and increase the torque) and ends up going through a potentiometer that is used as the feedback. The servo's output shaft is controlled by sending a PWM signal to the motor. That signal, combined with the position sensing from the potentiometer, controls the amount of rotation of the servo's shaft. That means that by adjusting the PWM signal the servo shaft's position can be controlled reasonably precisely in a range of around 180 degrees.

Hobby servos are designed primarily for radio-controlled (RC) models. RC airplanes and cars have radio transmitters and receivers to transfer information, and that information is, for the most part, what position the servos should be in. A typical servo expects to see a control pulse every 20ms. The width of that pulse determines how far the motor turns. Common values for the pulse width are 1ms for 0 degrees rotation (typically anti-clockwise as far as the servo shaft will rotate), a 1.5ms pulse for 90 degree rotation from that 0 point, and 2ms for 180 degrees from the starting point.

Using the Arduino delay functions it would be reasonably easy to write a program that drives the control signal of a servo according to that formula. However, servos are so common that the Arduino IDE comes with a Servo library that works well and saves you the trouble of figuring out when to raise and lower the control signal. The Servo library

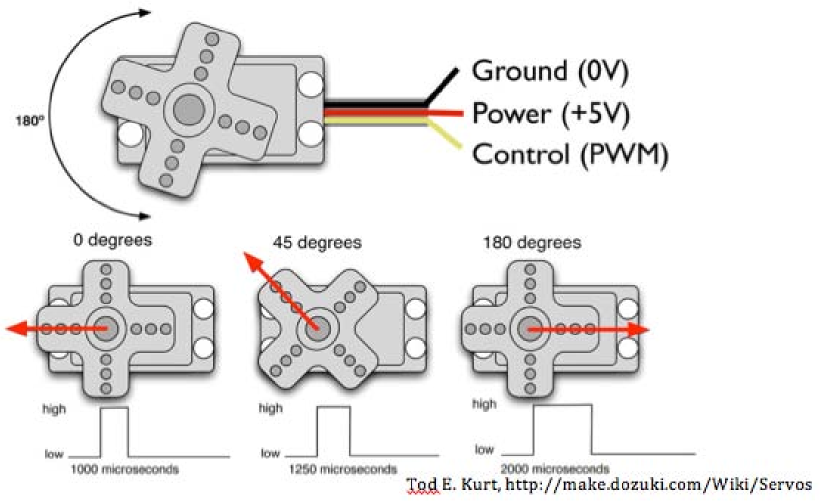


Figure 4.1: A standard servo has a three-wire interface. Power is always in the middle. Ground is on one side and will be the darker of the two outside wires (typically brown or black). The control wire will be on the other side and will be lighter in color, usually yellow, orange, or white. The position of the servo depends on the width of the PWM pulse sent to the control wire. Typically the width of the pulse should range from 1ms (1000us) and 2ms (2000us).

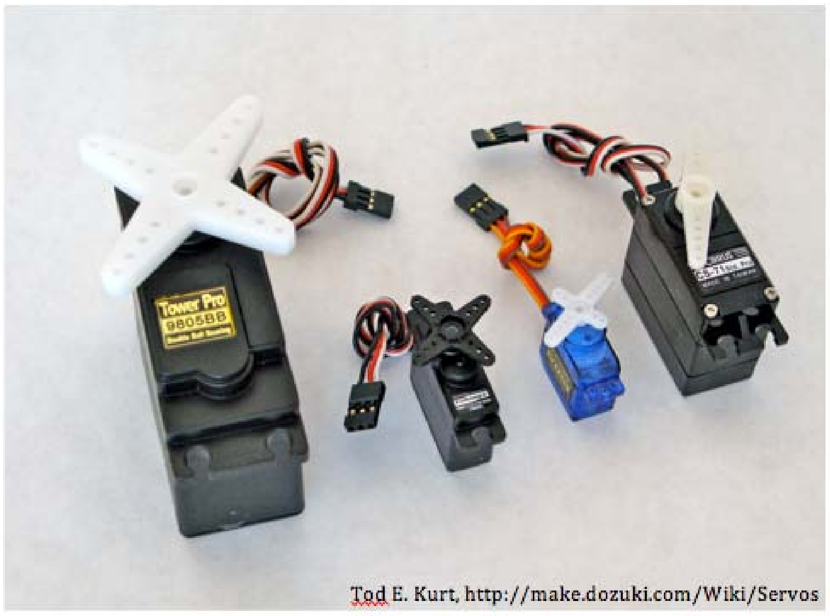


Figure 4.2: Servos come in a variety of sizes. Larger servos have more torque, but consume more power.

works by having you create a Servo object for each servo that you would like to control. The API is:

- `#include <Servo.h>`
  Include the Servo library in your program.
- `Servo <name1>, <name2>;`
  Create servo objects for each servo in your system
- `<name1>.attach(pin);`
  Attaches a servo to an Arduino digital output pin
- `<name1>.write(angle);`
  Move the servo to angle degrees. The angle argument must be in the range 0 to 179.
- `<name1>.writeMicroseconds(us);`
  Use this method if you'd like to adjust the microseconds of the PWM pulse directly. Useful ranges are 1000 to 2000 for standard servos, but you may have an oddball servo that needs different pulse widths.
- There are also methods for reading back the current angle of the `servo <name1>.read()`, what pin it's currently attached to `<name1>.attached()`, and detaching the servo from that pin `<name1>.detach()`, but they are less frequently used.

A nice example program for a servo is found in the Arduino example set:

```
// Controlling a servo position using a potentiometer
// (variable resistor)
// by Michal Rinott http://people.interaction-ivrea.it/m.rinott
//
// Modified slightly by Erik Brunvand
// This code assumes that you have a potentiometer attached with
// the end terminals connected to +5v and 0v, and the center wiper
// terminal attached to analog pin A0. The Servo has its power
// connected to +5v, ground to 0v, and control to digital pin 9
// The delay(15) is important - you have to give the servo time
// to get to its new position after you update. Servos are physical
// objects and take time to move to their new position.
// The Servo library is documented at
// http://arduino.cc/en/Reference/Servo

#include <Servo.h>

Servo myservo;    // create servo object to control a servo

int potPin = A0;  // analog pin used to connect the potentiometer
int val;          // variable to read the value from the analog pin

void setup(){
  myservo.attach(9);  // attach the servo object to pin 9 object
}

void loop(){
  val = analogRead(potPin);          // read from the pot (0-1023)
  val = map(val, 0, 1023, 0, 179); // scale for the servo (0-179)
  val = constrain(val, 0, 179);     // make sure to stay in range
  myservo.write(val);          // set servo position to the scaled value
  delay(15);                         // wait for the servo to get there
}
```

Servos are wonderful devices - they're inexpensive, easy to control with the Servo library, and provide reasonably precise position control in the range of 0-179 degrees. There are a variety of ways to connect them to things in your system that allow things to rotate, or (with an appropriate linkage) move back and forth.

Servos also use a bit more power than any of the other devices that we've seen in this course. A couple small servos should be fine when connected to the +5v output of the Arduino, but remember that there's a limit to how much current that output can provide (500mA total when connected to USB, and ~800mA when connected to a 9v power adapter). If you're using more than a couple small servos, or larger servos, you'll want to connect them to a separate power supply. You can easily use a separate +5v power supply (like an AC adapter, or wall wart) just for the servos. Most servos will operate with a DC voltage from around 4.8v to 6.0v. Connect your servo power and ground to that separate power supply, and your Arduino to USB or to its own power supply. Remember Big Idea #7 and connect the grounds of the servo power and the Arduino power together.

The Arduino's Servo library can control up to 12 servos, and you should note that it also disables the PWM (`analogWrite()`) functionality on pins 9 and 10 whether there is a servo connected to those pins or not. If you need to control more than 12 servos, or if you would like to control multiple servos from fewer pins, there are many multiple servo-driver solutions out there. Some are based on the TLC5940 chip that we saw in Section 2.4.4. That chip has 16 outputs where each output has its own separate PWM frequency. It's natural to use those for dimming LEDs, but it works almost as well for controlling multiple servos. The TLC5940 library has a servo-control option.

A standard servo moves through 180 degrees of motion, but you can also buy continuous rotation servos that spin rather than stopping at a fixed spot. These servos are controllable using the same PWM signals that a standard servo uses, and can thus be controlled using the Servo library and `<name>.write(degrees)`. For a continuous rotation servo the 90 degree point is where the servo stops. A degree rating in between 0 and 89 will spin the motor anti-clockwise with the motor speeding up as you go from 89 to 0. A degree specification of 91 to 179 will spin the motor clockwise with the motor speeding up as you move from 91 to 179.

4.2 DC Motors

Inside of a servo is a small DC motor. That works well for the specific use in a servo, but sometimes you want a motor that spins (perhaps faster or with more torque than a continuous rotation servo), or a larger motor that can drive a larger load. In other words, sometimes you want to use a DC motor directly. DC motors are actually extremely simple devices - simpler in many respects than servos. A DC motor will spin whenever a sufficiently positive DC voltage is applied to the two inputs of the motor. If you reverse the voltage (reversing Vdd and GND), the motor will spin backwards. If a motor is rated for a range of voltages, the motor will spin faster and have more torque if you increase the voltage. A tiny motor like the one in Figure 4.3 will spin very fast (some tiny motors spin up to 10,000RPM) but with very low torque. These motors consume very little power and can be powered from the +5v pin of the Arduino (assuming your motor is rated for 5v).

A larger motor will not only be physically larger, it will require much more power (more current), and be much stronger. DC motors in sizes from something similar to a D-battery, to motors the size of a garbage can, or even larger, are staples of all sorts of machinery. A selection of motors slightly larger than the one in Figure 4.3 is shown in Figure 4.4. These

Figure 4.3: A tiny DC motor. A motor like this (approximately an inch long) typically spins very fast with a small voltage. This motor is rated at 5v.

motors run at voltages ranging from 9v to 24v. Some have gear reduction heads that slow the rotation speed and increase torque. All of these motors consume much more current than an Arduino can supply, so a separate power supply is a must. The external supply is also required if the motor supply is higher than the Arduino can tolerate (such as the 24v motors).

What is needed in the case of these larger motors is some way of turning the motor on and off from a microcontroller like the Arduino, but have the power to the motor supplied by a separate supply. We could easily turn the motor on and off by putting a switch in series with the power supply connection to the motor. If we want a program to control the motor, we need an Arduino controlled switch that turns the voltage on and off to the motor. The perfect component for this is the **transistor**.

A transistor is a component that does many interesting things. One of the simplest things that it can do is act as a switch. A basic transistor is a three-terminal device that has a schematic symbol as seen in Figure 4.6. The bipolar devices have three terminals named Base, Collector, and Emitter. The MOS devices have three terminals named Gate, Source and Drain. For our purposes these are identical devices with slightly different names. There are two main types of transistors that we're interested in, and two techniques for constructing them from silicon. The two types are NPN/NFET which "turn on" (close the switch) when the Base/Gate is at a high voltage, and PNP/PFET that close the switch when the Base/Gate is low.

The operation (Bipolar terminology) is that if a high voltage is applied to the Base terminal, the transistor closes the switch and current can flow between Collector and Emitter. If the Base is at a low voltage, the switch is open and no current can flow between Collector and Emitter. This makes the transistor act like a switch that is controlled by the Base voltage. The thing that makes this a great switch for our motor application is that a very small signal on the Base can switch a very large signal between Collector and Emitter. Bipolar devices like this can switch quite high voltages and quite high currents. A 2N2222 transistor is a standard example of a very common NPN device. It can switch up to 40v between Collector and Emitter, and can conduct up to 800mA of current.

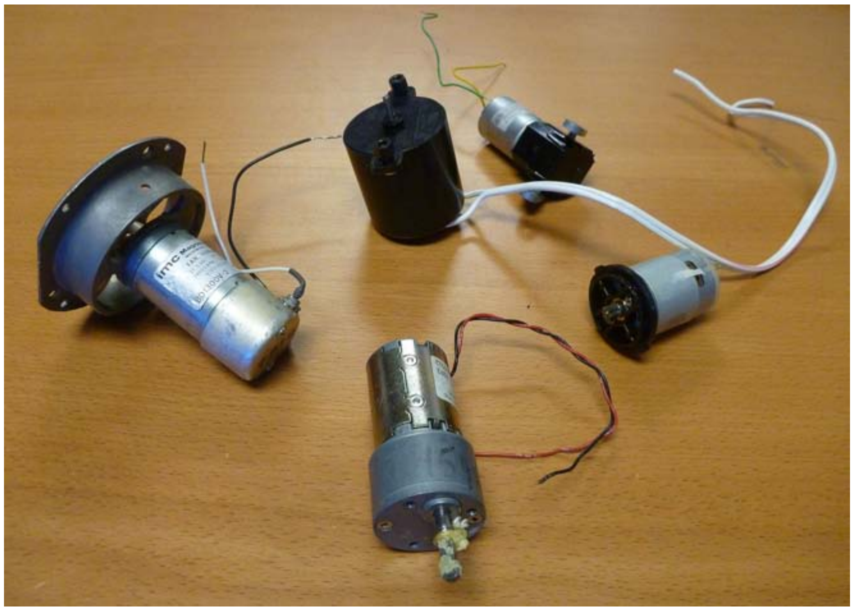
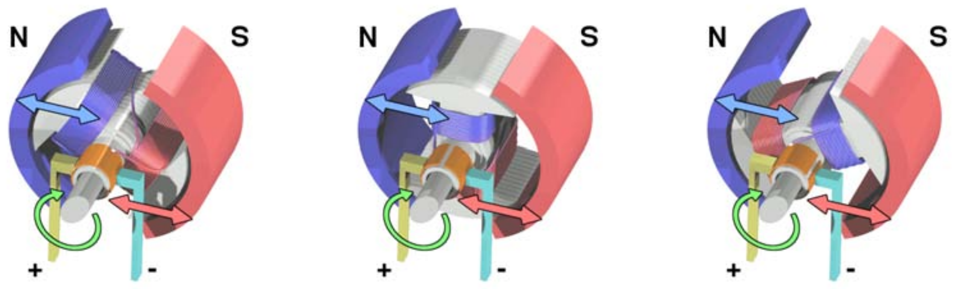


Figure 4.4: Slightly larger motors. These motors are around 3-4inches long, and operate at voltages ranging from 9v to 24v. The motor in the front center has a gear head. This attachment (the dull gray part) is a gear reduction unit that slows the rotation down to around 20 RPM. The gear reduction also increases the torque dramatically. The motor on the left has a fan attached.



Images by Wapcaplet, wikimedia commons

Figure 4.5: Diagrams of the behavior of a simple 2-pole brushed DC motor. When the coil is powered a magnetic field is generated. This causes the left side of the armature to be repelled from the N side of the fixed magnet and drawn towards the S magnet. By the third step, the fields are about to be reversed through the brushes connecting on the shaft, and the rotation will thus continue.

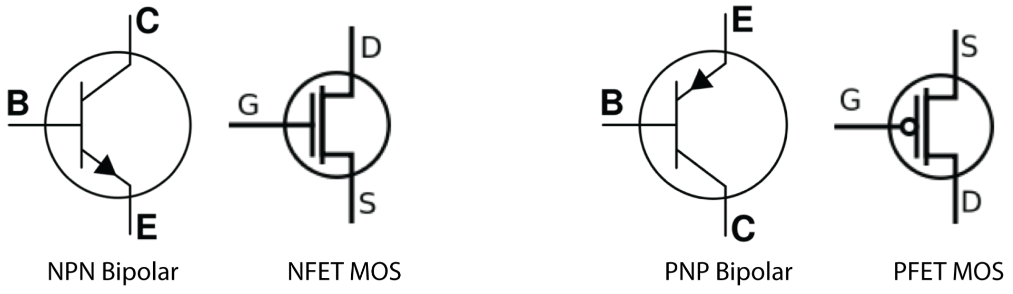


Figure 4.6: Schematic symbols for the major types of transistors used in physical computing. The three terminals o the devices have different names, but behave the same for our purposes (using the transistors as switches). The bipolar devices have a Base that controls the switching between Collector and Emitter. The MOS devices have a Gate that controls the switching between Source and Drain. The NPN Bipolar and the NFET MOS devices will turn on (conduct) when the Base or Gate inputs are high. The PNP Bipolar and PFET MOS devices will conduct when the Base or Gate inputs are low.

If you need to switch more current or higher voltages, a TIP120 is another common NPN device. Technically this is what is known as a Darlington pair device, that is, it has two NPN transistors in series to boost the capacity. The TIP120 can switch up to 60v at 5A of current. If you need to pass more than about 1A of current you should put a heat sink on the transistor. The heat sink attaches to the metal tab on the package (see Figure 4.7).

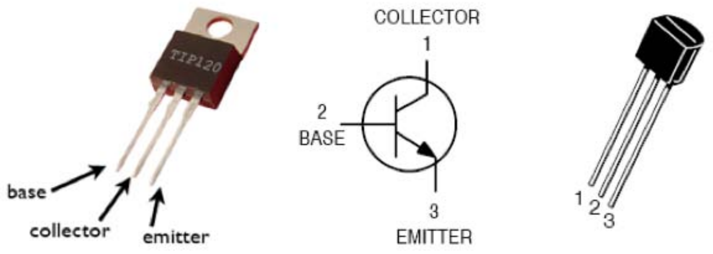


Figure 4.7: Images of two common NPN bipolar transistors, the TIP120 on the left and 2N2222 on the right.

One extra thing to keep in mind with NPN Bipolar transistors is that when you raise a voltage on the Base terminal, some current will flow into that terminal. It's common to put a small resistor in series with the Base terminal to limit that current. One could figure exactly what this resistor could be, but as a rule of thumb most people use around 1000-1500Ω.

As an example of using a transistor like the TIP120 to control a DC motor is shown in Figure 4.8. The transistor is acting as a switch controlled by the Arduino's digital output pin. The 1000Ω resistor is a current-limiting resistor that keeps too much current from flowing from the Arduino pin into the transistor's Base. This is a nice general schematic that can switch all sorts of devices that require more voltage or current than the Arduino can supply directly. Remember that if you use multiple power supplies you should always connect all the ground signals together.

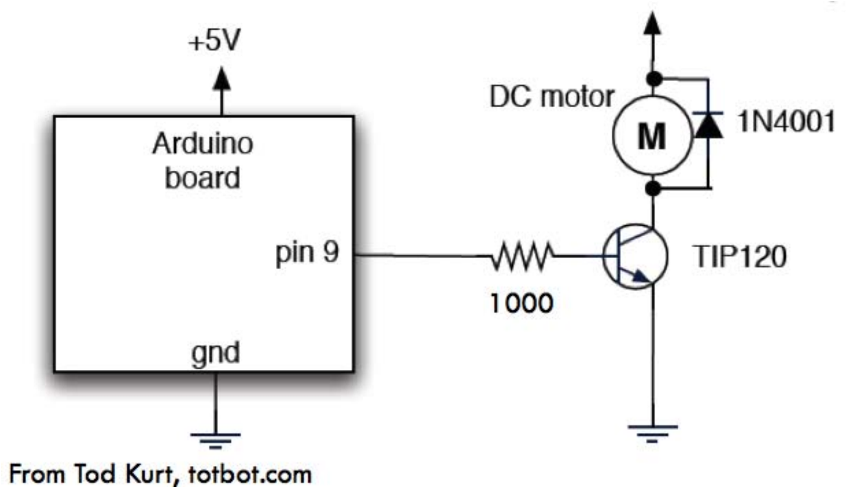


Figure 4.8: An example of using a TIP120 transistor to switch a DC motor on and off. Pin 9 of the Arduino is driven to +5v or 0v using a `digitalWrite(9);` function. When the pin is at 5v the TIP120 conducts from Collector to Emitter and the motor be activated. When the pin is at 0v the TIP120 does not conduct and the motor turns off. The 1N4001 is a diode. It is used as a snubber diode to protect the circuit against reverse current as the motor is spinning down. With a TIP120 transistor, the Motor Power Supply can be up to 60v DC.

Big Idea #13

If you need to turn a device on and off that uses a higher voltage or more current than the Arduino can supply, you should use a transistor as a switch. The signal going to the Base of the transistor is easily driven by the Arduino, and the device power can be switched through the transistor's Collector and Emitter terminals.

A DC motor is an example of an "inductive load." This means that it acts somewhat like an inductor when current flows though the device. Without going into details, this means that when current stops flowing in the device it reacts by resisting that stoppage of current though a burst of reverse current. In the motor's case this happens because a motor that is still turning after the power is removed acts like a generator. In any case, this possibility of reverse current can cause damage to the circuit. The diode in Figure 4.8 is a "flyback diode" or "snubber diode" that protects against this reverse current by letting it flow through the diode instead of the transistor. The size of the diode is not critical and the 1N4001 is an easily obtained common diode for this application that can conduct up o one amp. If you're switching an inductive load with a transistor you should always use a snubber diode.

4.2.1 Motor Speed Control

DC motors have very simple control: when you apply a voltage to the terminals of the motor, the motor turns. They also have the interesting property that if you send pulses to motor, the motor will essentially integrate those pulses and act as though it is receiving a lower voltage. This can be used a speed control for the motor.

This is yet another use for pulse width modulation (PWM). The percentage of time that the pulses are high is directly proportional to the speed of the motor. This PWM speed control works if the motor is connected directly to the Arduino, and it also works if it is connected through a transistor as in Figure 4.8 (as most motors will be). The switching transistor is more than fast enough to transfer a PWM signal on the Base into PWM switching between Collector and Emitter. To use PWM as speed control use `analogWrite(pin);` on the pin that is driving the Base of the switching transistor.

4.2.2 Motor Direction Control

Another interesting feature of DC motors is that if you reverse the voltage on the control wires, the motor will switch direction. This is a feature that is used in all sorts of mechanical equipment. The trick is how to organize the circuit so that you can switch the voltages without grabbing the wires and physically changing them. The circuit that accomplishes this trick is the H-bridge. The H-bridge is a clever connection of transistors that allows the connection to a motor to be switched to be in either direction.

A schematic for an H-bridge is shown in Figure 4.9. It consists of four switches organized so that closing pairs of switches can cause the direction that current flows in the motor to reverse (Figure 4.10. If the current reverses, so does the direction of the motor. The switches in an H-bridge can be physical switches, or electronic switches such as transistors. For motors of the size typically used in physical computing systems transistors are the most common switches used. For larger motors that might need hundreds of volts, physical switches such as relays or contactors (heavy duty industrial relays) are often used in H-bridge circuits so that extremely high voltages and current can be switched.

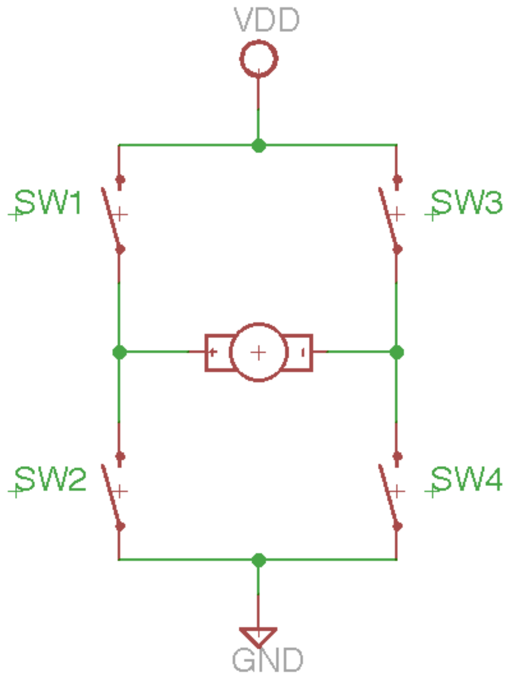


Figure 4.9: Schematic of an H-bridge circuit. There are four switches in an H-bridge. The purpose is to change the direction that the current flows in a DC motor.

You could make an H-bridge yourself with four switches, or with four transistors. It's not uncommon for people to wire up four TIP120 transistors to make a high-current H-bridge. Actually, it's more likely that they would use two TIP120s and two TIP125s. The

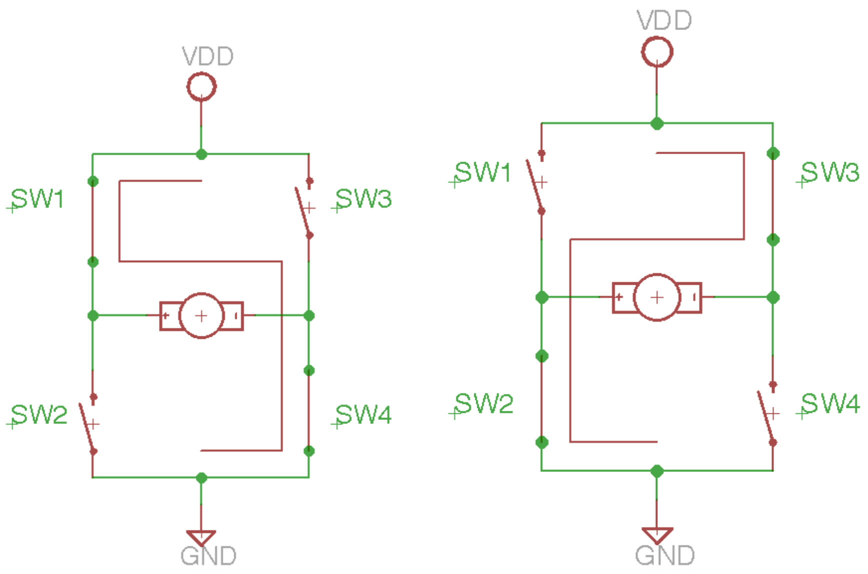


Figure 4.10: The operation of an H-bridge circuit. In the left half of the figure switches S1 and S4 are closed and current flows from left to right in the motor M. In the right hand circuit switches S2 and S3 are closed and current flows from right to left through the motor. Switching the current direction switches the motor direction.

TIP120 NPN transistors turn on when you apply a high Base voltage, and the TIP125's turn on when you supply a low Base voltage so it's easier to build an H-bridge with its "one on and one off" style with these complementary types of transistors. But, even more convenient than building an H-bridge with transistors is simply using an H-bridge chip. Because there are so many places where DC motors are used, there are a lot of pre-designed H-bridge chips that you could choose from. Two common chips that are great for physical computing because they're small, convenient, and perfectly sizes for the relatively small motors that physical computing systems tend to use (i.e. an amp or two) are the L293D and SN754410.

These chips are called "quad half H-bridge" chips because they have four clusters of circuits on the chip, and each of the clusters implements half of an H-briidge (the left side of the circuit in Figure 4.9 for example). An overview of these chips is shown in Figure 4.11. Eavh of the triangle-shaped components in the figure are half of an H-bridge as seen in the right side of the figure. Figure 4.12 shows how these "half H-bridges" can be used to control motors. If you're just controlling a single-direction motor you can use a single half-H-bridge just to provide high-current switching as seen in the right side of the figure. If you would like to reverse the direction of your motor you can use two of them configured as a full H-bridge as seen in the left of the figure.

The L293D and SN754410 are great chips to use for motor control. They come in a variety of packages and are easy to use with solderless breadboards. Some things to keep in mind when using these chips:

- There is a separate Vdd pin for the motors, and for the chip itself. The chip itself needs +5v. The motor supply can be any voltage in the range of +5v to 36v.
- Remember that if you're using multiple power supplies (such as one from Arduino

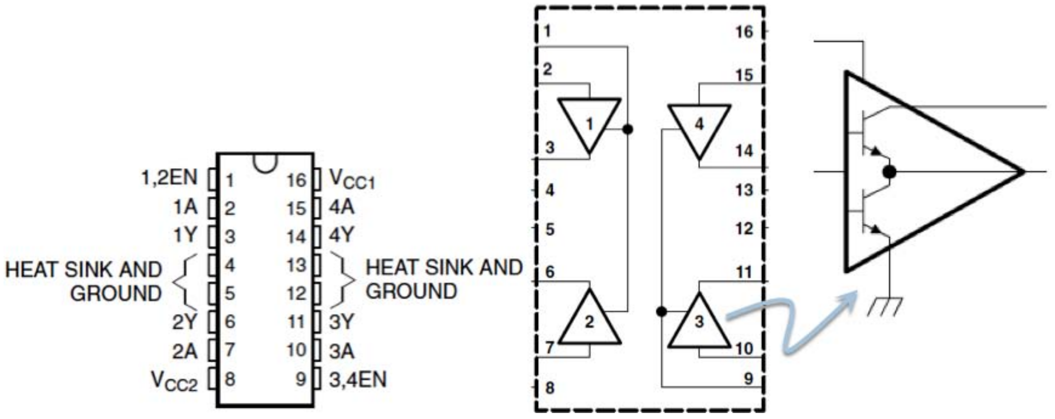


Figure 4.11: Chips that implement H-bridge circuits include L293D or SN754410. These diagrams are taken from an L293D datasheet. These chips are known as quad half H-bridge chips because there are four circuits in the chip, each of which is half of an H-bridge as shown in the right side of this figure. Two of these half H-bridge circuits can be used to make a full H-bridge and be used for controlling the direction of a DC motor. The enable inputs (pin 1 and pin 9 on the chip) can be used with PWM for motor speed control.

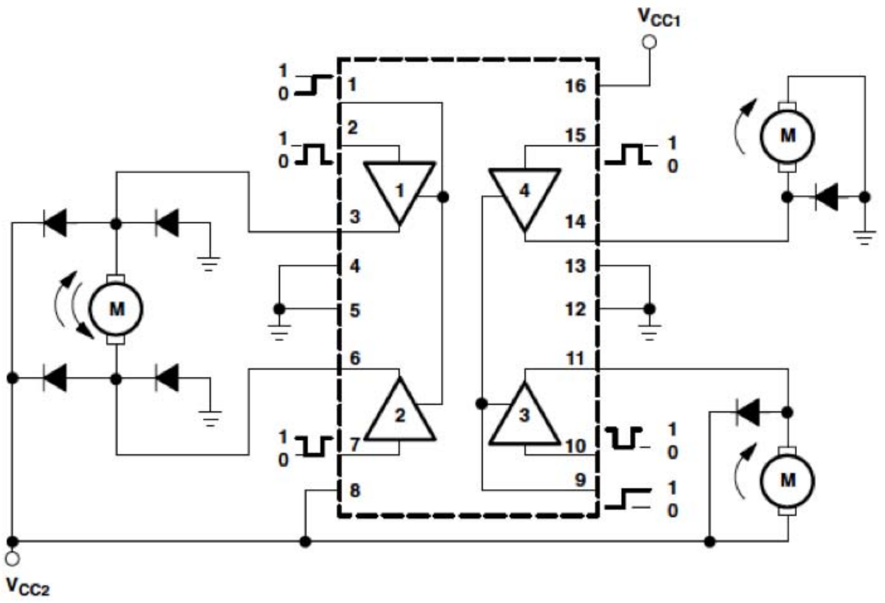


Figure 4.12: This figure is taken from the data sheet for an L293D chip. It shows the chip being used to control a bidirectional motor on the left, and two single-direction motors on the right. In the bidirectional motor case, the Arduino would control the direction using pins 2 and 7 on the H-bridge chip. The unidirectional motors are controlled through pins 15 and 10 of the chip. These chip inputs may be driven direction from Arduino pins: no series resistance is needed because the H-bridge chip is designed to not sink current through the pins. Note that in all cases snubber diodes have been used to protect the circuit. In this circuit, Vcc2 at the bottom left is the motor's power supply and can be up to 36v. Vcc1 in the upper right is the chip's power and is +5v. This can come from the Arduino because although the chip switches high power devices, it doesn't itself consume must power.

and one for the motor) you should tie the ground signals together.

- If your motor is at a relatively high voltage you may want to put a heat sink on the driver chip. Note the ground pins in Figure 4.11 (pins 4, 5, 12, and 13). They are also marked as "Heat Sink." This means that they dissipate heat from the chip as well as being ground connections. You can attach a metal heat sink to these pins, either by buying a clip-on heat sink for chips like this, or by soldering a piece of metal to the pins. That heat sink will radiate the generated heat and allow the chip to operate at higher motor voltages.

- You can use PWM on the Enable signals to control the motor's speed. In Figure 4.12 the bi-directional motor's speed could be controlled with PWM on Pin 1 of the driver chip.

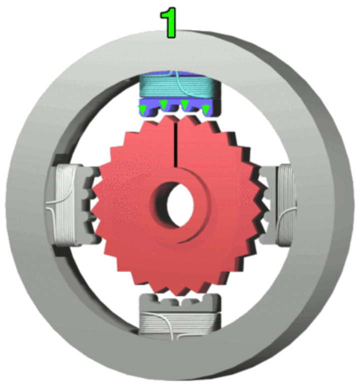
4.3 Stepper Motors

Servo motors are great because they are positionable across a reasonably precise range of 0 to 180 degrees, and will hold that position once put there. But, they aren't very powerful, and they are limited to a 180 degree range. DC motors are great because they spin as long as you power them, they can be very powerful, and it's easy to control the speed with PWM. But, they aren't precisely positionable, and they don't stay put. A stepper motor is, in some sense, the combination of these two types of motors: it is a powerful DC motor that is also precisely positionable and will hold its position when stopped. They are used in a huge variety of consumer applications that require precise movement. The sensor/imager of a scanner, and the paper drive in a printer are driven from steppers, for example. The axes of a numerically controlled milling machine or X-Y plotter are also driven by stepper motors.

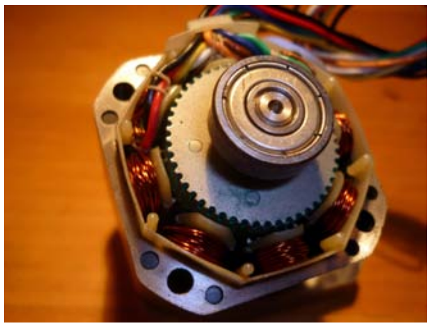
The only downside of a stepper motor is that they're a little more complex to control than a DC motor or a servo. Stepper motors have multiple armatures that are wired so that if you energize them in sequence, the shaft is pulled around in small discrete steps (see Figure 4.13). Typical stepper motors have steps that range from 7.5 to 0.9 degrees per step. This corresponds to 40 to 400 steps per revolution. This gives them nice fine-control of position by choosing how many steps to move.

One easy way to tell if the motor that you have is a "regular" DC motor or a stepper is by the number of wires used to control the stepper. A regular DC motor from the previous section has two wires. Stepper motors have four, six, or eight wires. The multiple wires control different coils inside the motor and are energized in very specific sequences to make the motor advance.

There are two main varieties of stepper motors: unipolar and bipolar (see Figures 4.14 and 4.15). A unipolar stepper has multiple coils that are each driven in sequence, but that are always driven in the same "direction." That is, in Figure 4.14 the A1-A2 coil is always driven with + on A1 and GND on A2. In fact, A2 is always GND: the second coil is driven with + on A3 and GND on A2. A unipolar stepper will have six or eight wires depending on whether the GND wires of the first two coils are connected (as shown in Figure 4.14), or separated. A bipolar stepper has only four wires (as seen in the figure). The circuit that drives a bipolar stepper must be able to reverse the current in those coils for different phases of stepper operation. If that sounds like an H-bridge might be the right circuit to use, you would be correct.

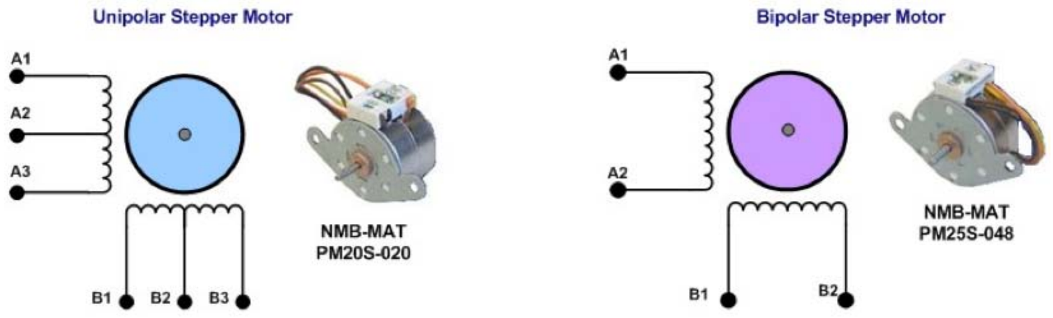


Wapcaplet, wikimedia commons



Vlastn fotografie, wikimedia commons

Figure 4.13: On the left is a cutaway diagram of a stepper motor. The zig-zags on the armatures attract the zig-zags on the shaft to be aligned exactly with the points. The other armatures are offset slightly so they pull the shaft around a small number of degrees if you energize the armatures in sequence. The photo of a stepper motor on the right shows the zig-zags in real life.



http://www.ermicro.com/blog/

Figure 4.14: The two main varieties of stepper motors: unipolar and bipolar. A unipolar stepper requires that each of the coils be energized in a particular sequence. They typically have 6 wires, but can have 8 wires if the center taps are split into two wires. That is, there might be four separate coils each with two wires. Bipolar steppers have only 4 wires and the circuit that drives them must be able to reverse the current through each coil during different phases of operation.

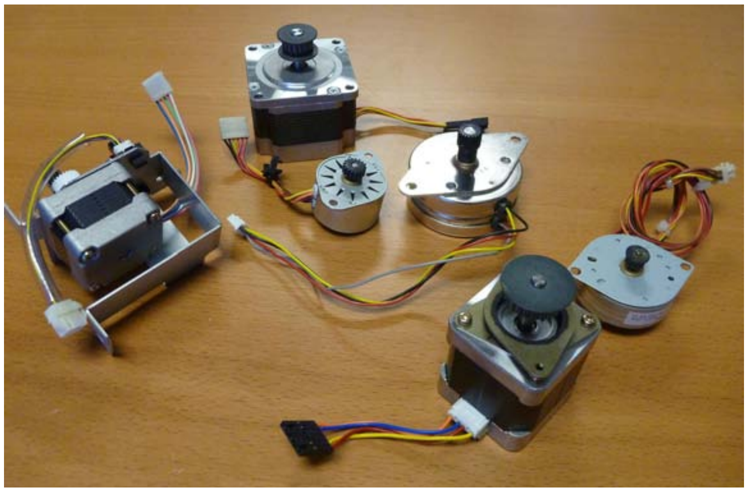
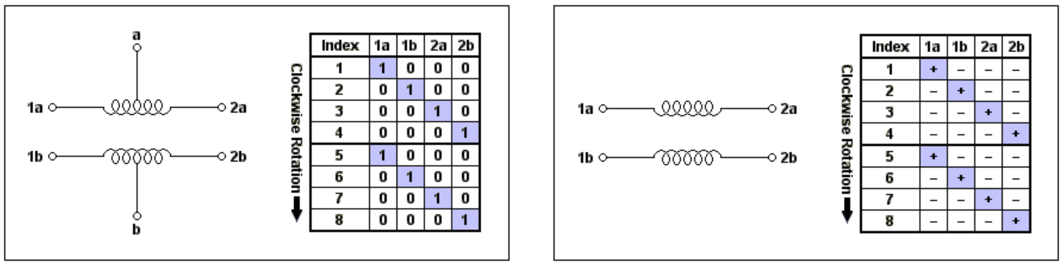


Figure 4.15: Some stepper motors in different sizes and types. Any steppers with four wires are bipolar steppers. Steppers with six wires are unipolar. These steppers include new motors and motors pulled from old equipment such as scanners. There is no standard color coding for control wires. A little exploring with an ohmmeter will help you figure out which ones are connected through a coil, and which are connected to different coils (i.e. not connected to each other).



Index	1a	1b	2a	2b
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	1	0	0	0
6	0	1	0	0
7	0	0	1	0
8	0	0	0	1

Index	1a	1b	2a	2b
1	+	–	–	–
2	–	+	–	–
3	–	–	+	–
4	–	–	–	+
5	+	–	–	–
6	–	+	–	–
7	–	–	+	–
8	–	–	–	+

http://members.home.nl/bzijlstra/hardware/stepper/stepper.htm

Figure 4.16: These are examples of coil activation sequences for stepper motors: unipolar on the left and bipolar on the right. The bipolar sequence includes bi-directional activation. In practice there are many other sequences that can be used and that have different properties.

Some example sequences for driving steppers are seen in Figure 4.16. If you connected the control wires of a unipolar stepper to an Arduino, for example, and then drove them in this sequence (with the center tap wires of the unipolar stepper tied to GND), the stepper would rotate one step for each step in the sequence. Actually, that's not true. You would burn out your Arduino because each phase of the step sequence would draw more current than the Arduino could supply. So, instead you would drive each of the four coil activation wires (A1, A3, B1, and B3 in Figure 4.14 on the left) through transistors that could provide more current to the motor. You could use a connection such as the one in Figure 4.8 to drive each of the control wires. The Arduino web site shows another connection through a "Darlington array" as seen in Figure 4.17. A Darlington array is just a chip that has six separate Darlington-type transistors on it, so it's exactly the equivalent of using discrete transistors, but contained in a handy chip.

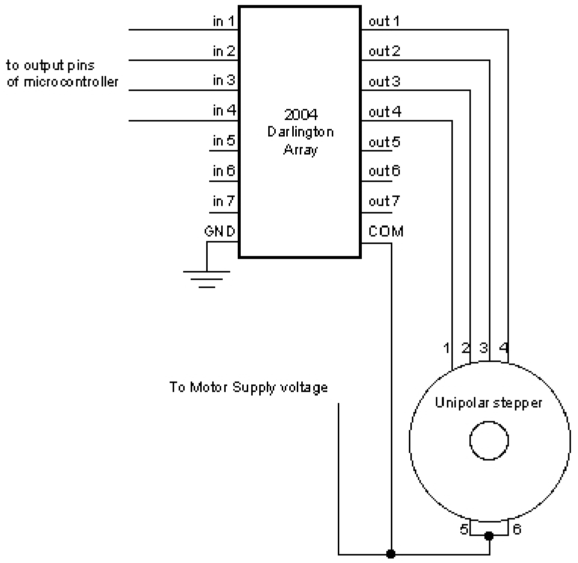


Figure 4.17: A circuit that drives each of the four control wires of a unipolar (6 wire) stepper using Darlington transistors. The two center-tap wires are connected to the motor's power supply because the Darlington transistors are configured to pull the 1, 2, 3, and 4 wires low (sink current).

A bipolar stepper takes a slightly more complex circuit because the four wires of this type of stepper need to be used as two coils that can each be driven in both directions. So, you need some sort of circuit that can reverse the direction of the current like an H-bridge. The H-bridge chips discussed in the previous section (the L293D and SN754410) work well for this application. A connection of a bipolar stepper to an H-bridge chip is shown in Figure 4.18. A more detailed image of this connection is seen in Figure 4.19.

You could easily write code to send the appropriate sequence of signals on the digital pins of the Arduino to make the stepper move. However, this is such a commonly used type of motor that there is built-in Stepper library in the Arduino IDE. This library lets you instantiate a Stepper object for each stepper that you have connected according to Figures 4.17 or 4.18. It turns out that if you have the steppers connected in this way, the same sequence works for both unipolar and bipolar steppers. The current switching is handled through the H-bridge connection for the bipolar motor.

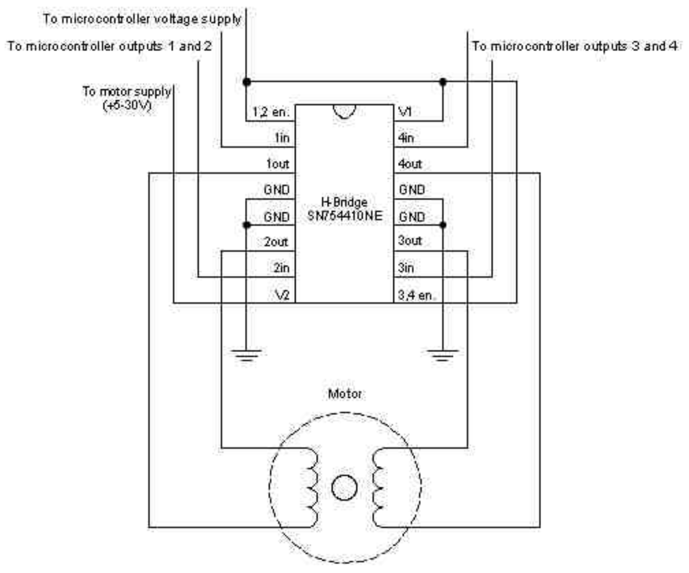


Figure 4.18: The connection of a bipolar stepper with four wires to an H-bridge driver chip. Note that the motor's power supply is separate from the chip's power supply (shown as connected to the micro controller voltage supply in this figure).

To use the Stepper library:

- Include the Stepper library with `#include <Stepper.h>`
- Instantiate a Stepper object for each motor that you have connected. Note that each motor requires four Arduino pins. The object instantiation function takes the number of steps in one revolution and the four pins you're connected to as arguments `Stepper <name1>(steps, pin, pin, pin, pin);`
- Now you can use the following methods on that Stepper object

  `setSpeed(rpm)` Sets the speed of the stepper in RPM. This adjusts the delay between steps that the lib ray uses when driving the stepper

  `step(steps)` Make the stepper turn for a number of steps. A positive number of steps turns the motor one way, and a negative number makes it turn the other way.

An example Stepper program that comes with the Arduino IDE is called "MotorKnob." This program reads the analog value of a potentiometer (the knob), and turns the stepper motor to match the rotation of the knob. That is, when you turn the knob, the stepper will turn the same amount.

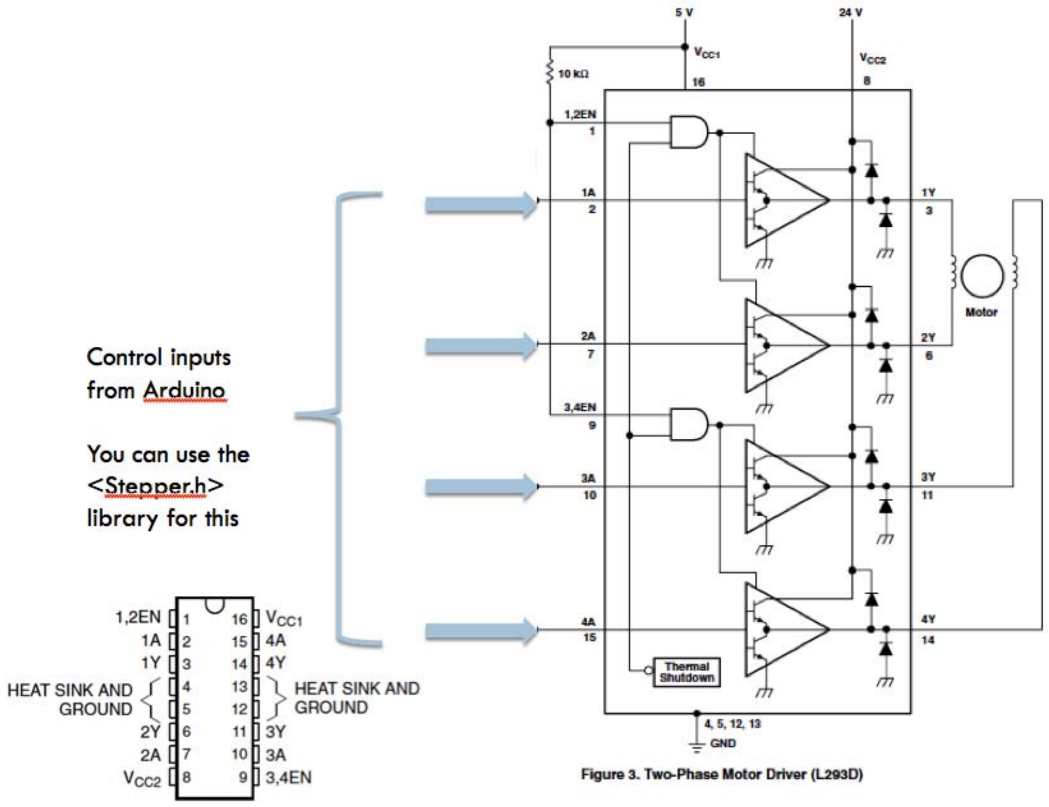


Figure 4.19: A more detailed view of the H-bridge connection to a bipolar stepper motor. Note that there are snubber diodes contained within the L293D driver chip (that's what the "D" in the name means). However, many people include extra snubber diodes on the outside of the chip also just to be safe.

```
/*
 * MotorKnob
 *
 * A stepper motor follows the turns of a potentiometer
 * (or other sensor) on analog input 0.
 *
 * http://www.arduino.cc/en/Reference/Stepper
 * This example code is in the public domain.
 */

#include <Stepper.h>

// change this to the number of steps on your motor
#define STEPS 100

// create an instance of the stepper class, specifying
// the number of steps of the motor and the pins it's
// attached to
Stepper stepper(STEPS, 8, 9, 10, 11);

// the previous reading from the analog input
int previous = 0;

void setup() {
  // set the speed of the motor to 30 RPMs
  stepper.setSpeed(30);
}

void loop() {
  // get the sensor value
  int val = analogRead(0);

  // move a number of steps equal to the change in the
  // sensor reading
  stepper.step(val - previous);

  // remember the previous value of the sensor
  previous = val;
}
```

The Stepper library makes using unipolar and bipolar steppers very easy. However, it doesn't make use of the most modern stepper driving circuits, and it uses four wires per stepper motor. The Arduino web site shows how to reduce this to two wires per stepper with the addition of a few more parts, but it's still the case that there are more full-featured stepper drivers out there. The main extra feature of other driver chips is the ability to do "micro stepping." This is a technique where multiple coils are driven at the same time in a more complex pattern that allows the stepper to move a half step or less on each step. This can greatly increase the resolution of the stepper. The other thing that the more modern stepper drivers have is the equivalent of constant current outputs for motors. These outputs are called "chopping drivers" because they limit the total amount of current delivered to the pins independent of the voltage by chopping up the higher voltage current into smaller pieces.

The reason these types of chopper drivers are interesting is that stepper motors are primarily rated by how much current you can put through the coils. The current determines the strength of the magnetic fields that move the motor so it's an important parameter. Sometimes stepper motors are rated by voltage and resistance of the coils, but the really important number is the current rating. If you have only the resistance of the coils and the recommended voltage you can use Ohm's law to figure out how much current that rated voltage would result in. It's the current limit that the chopping drive enforces, independent of voltage.

This raises the possibility of using a higher than rated voltage, and letting the chopping drive limit the current to a safe level. This allows the motor to move more quickly because of the higher voltage, but still be safe because the current is limited. For example, you might have a stepper motor that's rated at 6v DC with 7.9Ω in each of the coils. Ohm's law tells us that this rated voltage and resistance will result in 0.76A of current (760mA). So, you could set your chopping drive on a stepper driver to deliver 760mA of current, but then bump the voltage up to 24v. The higher voltage will make the motor "snappier" when it moves, but it will still be safe because the current will be limited to the correct amperage. Almost all industrial use of steppers makes use of this type of high-voltage chopping driver.

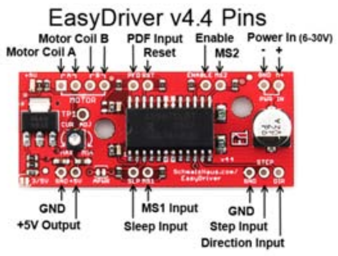
There are a number of purpose-built stepper driver chips and boards that use this technique. The EasyDriver and Pololu A4988 are two examples and are shown in Figure 4.20. These boards can each control one bipolar stepper motor using chopping current drive on the outputs. They will accept a motor power supply up to 30v. The interface to the microcontroller is through two wires: step and dir. The microcontroller makes a pulse on the step signal (raises the signal to +5v and then lowers it o 0v), and the driver will make the motor take one step for each time that signal rises. The dir pin controls the direction of that step.

These drivers also allow the motor to be driven in micro-step mode where each step is a full, half, 1/4, or 1/8th of the stepper motors normal full step. This is controlled by some mode pins on the stepper drivers. The pins of these stepper driver boards are on the same pitch as the holes in a solderless breadboard so they can be used with that connection technique. A picture of the wiring of one of these boards to an Arduino and a motor is shown in Figure 4.21. The power connection shown to the right of the motor is the motor's power supply. In this picture it is assumed that the Arduino is powered separately. The Step and Dir outputs from the Arduino are on digital pins 8 and 9. Note that the GND of the Arduino is connected to the GND of the motor driver board - always a good idea!

A feature of this board that is not being used in the figure is that the board also includes a +5v regulator. So, even if you're powering the motor with 30v, you could still take a +5v signal from that board for powering the Arduino. This is a very handy feature for installing things after programming and debugging is finished. You would only need one power supply for the entire system.

Note that these motor drivers are *only* for bipolar motors. They will not work with unipolar motors. Luckily bipolar motors are very common. Even more luckily, you can almost always take a unipolar motor and wire it for use as a bipolar motor. Look at the wiring in Figure 4.14. If you take the unipolar motor on the left and ignore (don't connect) the two center-tap wires (A2 and B2) you can use the remaining four wires as a bipolar stepper.

These stepper drivers are in many ways the preferred ways to driver stepper motors. They're a little more expensive than just using an H-bridge chip, but they allow higher voltages and chopping current drive, and micro-stepping. They also only use two wires per motor so you can use more of them with a single Arduino. A code snippet showing how to use the step/dir interface is shown below. There are also libraries available that deal with these step/dir style drivers. The `AccelStepper` library is one of the fancier ones that includes a feature to slowly accelerate your stepper to full speed and then slow down when it reaches the end of its rotation. I won't discuss all its features here, but you can find the details at `www.airspayce.com/mikem/arduino/AccelStepper/`.



https://www.sparkfun.com/products/10267

http://www.pololu.com/catalog/product/1183

Figure 4.20: Two stepper driver boards with chopping drivers. These boards each drive one bipolar stepper motor. They are driven by a pair of control wires from the microcontroller: step and dir. The dir pin controls motor direction, and a transition on the step pin causes one step for each rising edge on that pin.

```
// the number of steps to take is in the "steps" variable
// and the desired direction is in the dir variable

digitalWrite(dirPin, dir);              // set the desired direction

for (int i=0; i < steps; i++) {         // for each desired step
    digitalWrite(STEP_PIN, HIGH);  // move the step pin high
    delayMicroseconds(usDelay);    // wait a bit for the motor to move (1us is
        plenty)
    digitalWrite(STEP_PIN, LOW);   // now set step low for the next time
    delayMicroseconds(usDelay);    // and wait again
}
```

4.4 Questions About Motors

Some questions that you should ask when considering motors for use with a controller like the Arduino are:

- *What will you be moving with your motor?*

  If you're moving small things, and they don't need to move very much, you should use a servo. They're inexpensive, easy to control, and relatively precise. A couple of small servos can even be driven directly from your Arduino with no extra power supply. You will want to add a separate supply if you have many of them though.

  If you're making something move by driving things like wheels, or reeling something in and out, you probably want a DC motor. They're much stronger than servos, and definitely need a separate power supply. Control them through a transistor like the 2n2222 or TIP120. Remember that you can control the speed of the motor with PWM.

  Steppers are the right choice for precise rotational control and ample torque. They're the fanciest motors, and the most expensive, but the most versatile.

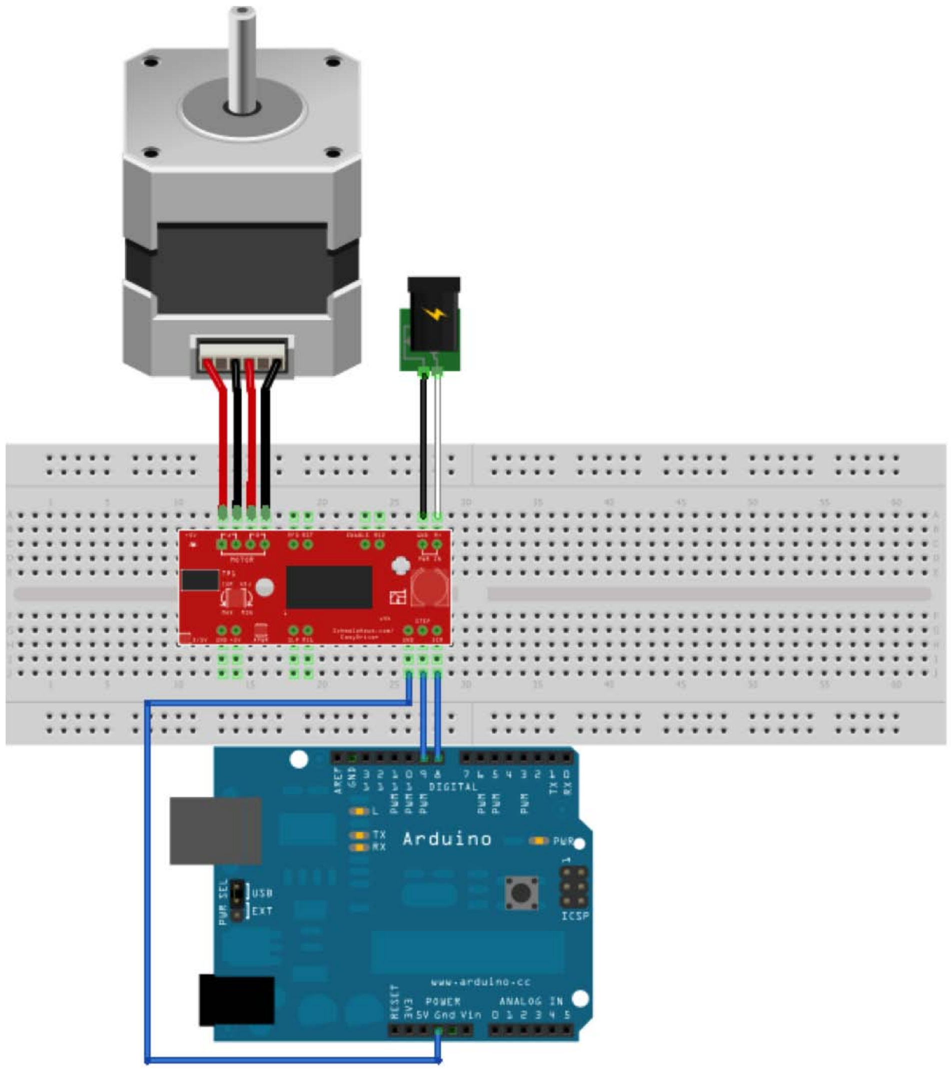


Figure 4.21: Wiring an external step/dir stepper driver to an Arduino and a motor.

- *How many servos are you driving?*

  The Arduino's servo library can control up to 12 servos. If you need more than that you need to look for an external servo-driver board. They're available at many of the sources listed in the sources chapter of these notes.

- *How much current will your stepper motor draw?*

  You should know this number so that you can set the chopping driver on your stepper driver board appropriately. There's a small potentiometer on these boards that you use to set the current limit. The question is, what is the current limit for your motor? Some motors will tell you in the specs when you buy them. Other motors will (strangely, I think) tell you indirectly by giving you a rated voltage and resistance for the coil. You should use that and Ohm's law to figure out what the current limit is. You can drive the motor at higher than the rated voltage if you have a chopping current drive set appropriately.

- *Are the ground signals from all your power supplies connected together?*

  Remember that whenever you have separate power supplies you must connect the grounds together so that they all have a common 0v reference.

- *Which wires are which on your servo?*

  Servos always have a three-wire interface. This is a standard connection so that servos can be swapped out of things like radio-controlled airplanes without rewiring. The power is always in the middle, and the GND and control are on the outside of the three-wire bundle. The ground will be the darker of the two (usually black or brown). The control wire will often be white, yellow, or orange.

- *Which wires are which on your DC motor?*

  It doesn't matter. What matters is if the motor is spinning the direction you want it to when you apply a voltage to the wires. If it's not, reverse the wires. DC motors are designed to spin both directions by reversing the current.

- *Which wires are which on your stepper motor?*

  This is a little tricky. Stepper motors seldom come with data sheets unless you're lucky, and there's no standard for color or for physical arrangement. First you need to figure out whether you have a unipolar (6 to 8 wires) or a bipolar (4 wires) motor. Then you should use an ohmmeter to check the resistance between pairs of wires. Look at Figure 4.14 as a guide.

  For a bipolar stepper the wires that are part of the same coil will have some measurable resistance between them, and wires in different coils will not be connected. If you get the orientation of two wires turned around your motor will "stutter" when you drive it. Reverse one pair of wires in your circuit and that should fix it.

  For a unipolar motor first figure out which wires are in the same coils. Then look at resistance within the coil. The center tap wire should have roughly half the resistance to the other wires in that coil because it comes out of the middle.

Chapter 5

Supply Sources

This is a definitely non-exhaustive list of some of the equipment mentioned in this course, a rough guide to prices, and some example suppliers.

Arduino: The main site for the Arduino is `www.arduino.cc`This is the official site for all open-source information, and for software downloads.

A great secondary source for Arduino is `www.freeduino.org`which is a non-official community-supported site.

Arduino comes in many hardware variations. The current generic Arduino is the Arduino Uno. It retails for around $30. Some suppliers include:

- Sparkfun: `www.sparkfun.com`
- Radio Shack: `www.radioshack.com`
- Adafruit: `www.adafruit.com`
- Hacktronics: `www.hacktronics.com`

These are also great sources for all sorts of things related to embedded systems, physical computing, making, and hardware hacking.

Switches and sensors: The previously mentioned sources also have loads of switches and sensors of all sorts. Many of the sensors that we didn't discuss in this workshop are also resistive sensors and can be used in exactly the same way as we used pots and light sensors.

Other sources for small electronic parts like this include:

- Jameco: `www.jameco.com`
- Digikey: `www.digikey.com`
- Mouser: `www.mouser.com`

Of these suppliers, Jameco is more for the hobbyist, and Digikey and Mouser are more for the professional. They all sometimes have great prices, especially in quantity, but you sometimes really have to know what you're looking for. As an example an individual light sensor at Sparkfun or Jameco runs between $1.25 and $2.00. But, with a little more searching, Jameco has a bag of 50 assorted light sensors for $15. Look for Jameco's bags of assorted switches, assorted LEDs, etc. They can be a great way to quickly stock a set of kits.

Servos: These are the type of servos used in radio controlled model airplanes and model cars, and for small robotics projects. So, any hobby shop will have them. Radio Shack, Sparkfun, etc. also carry them, but you might find a better deal at a hobby shop or a robotics hobbyist shop. Some examples:

- Hobby Partz: `www.hobbypartz.com`
- Trossen Robotics: `www.trossenrobotics.com`
- Pololu: `www.pololu.com`

Electronics surplus resellers: Sometimes you can find great deals on surplus parts at a surplus reseller. Here you really do need to know exactly what you're getting, and be willing to take some chances, but you can find some real bargains if you look around. Some examples:

- Electronics Goldmine: `www.goldmine-elec.com`
- Marlin P. Jones: `www.mpja.com/`
- Alltronics: `www.alltronics.com/`
- Electronics Surplus: `www.electronicsurplus.com/`

Lady Ada (Limor Fried) at Adafruit also has some great suggestions: `www.ladyada.net/library/procure/hobbyist.html`

Chapter 6

Kinetic Art using Physical Computing

Kinetic art contains moving parts or depends on motion, sound, or light for its effect. The kinetic aspect is often regulated using microcontrollers connected to motors, actuators, transducers, and sensors that enable the sculpture to move and react to its environment. But, distinct from other types of computer art, the computer itself is usually not visible in the artwork. It is a behind the scenes controller.

An embedded system is a special-purpose computer system (microcontroller) designed to perform one or a few dedicated functions, often reacting to environmental sensors. It is embedded into a complete device including hardware and mechanical parts rather than being a separate computer system.

Kinetic art using embedded control is a marriage of art and technology. Artistic sensibility and creativity are required for concept and planning, and computer science and engineering skills are required to realize the artistic vision.

Using embedded computer control to build things such as kinetic art is, in some ways, the essence of physical computing. Wikipedia's definition of physical computing is

> Physical computing, in the broadest sense, means building interactive physical systems by the use of software and hardware that can sense and respond to the analog world.

In this course we've looked at some fundamental building blocks that would enable a programmer to start using physical computing elements in his or her work. Hopefully this brief introduction to the world of electronics as it pertains to physical computing will help point you in the right direction. Once you're off in the right direction, the choice of how you employ that direction in your own work is up to you!