Introduction to the Arduino Microcontroller

Learning Objectives:

At the end of this lab you should be able to:

* Describe the basic functioning of the “standard” Arduino microcontroller board
* Describe the capabilities of the Spartronics Experimenter Shield (any version), and how it relates to the Arduino
* Configure the Arduino IDE to communicate with the Arduino hardware
* Use the Arduino IDE to load, compile, download and execute (provided samples and user-written) programs
* Describe the basic structure of an Arduino program
* Use discrete components on a solderless breadboard with an Arduino
* Locate the resources available on the Arduino project and Atmel website

Components:

Qty. Item

1 Arduino Duemilanove or Uno board with ATmega328 microcontroller

1 Spartronics Experimenter Shield (SES) v1 or v2) or EduShield v3.x

1 USB (A-B) cable

1 hobby servo motor

1 9-12 VDC power supply (wall wart)

1 solderless breadboard

1 1kΩ resistor

1 3mm or 5mm red, green, or yellow LED

### Introduction - What is, and why, the Arduino?

A microcontroller often serves as the “brain” of a mechatronic system. Like a mini, self-contained computer, it can be programmed to interact with connected hardware and/or a user, much like a PC connected to a small network of hardware. As the computer industry has evolved, so has the technology associated with microcontrollers. Every year microcontrollers become much faster, have more memory, and extend their input and output feature sets, all the while becoming even cheaper and easier to use.

Even the most basic microcontroller units (MCUs) can perform simple math operations, monitor (read) analog/digital inputs, and control (write) analog/digital outputs. Most modern controllers have analog-to-digital converters (ADCs), high-speed timers, counters and comparators, interrupt capabilities, PWM outputs for producing square-waves, and a variety of serial communication ports. All of these can be used to interact with the board's immediate environment for a very wide variety of purposes.

The development board used in this course is called the *Arduino*. The name Arduino is (admittedly somewhat ambiguously) used to denote the hardware board, the software development environment, its library of easy-to-use software functions, and/or the layout standard of the original version's connection headers, depending on the context. As you will learn in the next few weeks, it is not necessary to use the Arduino IDE or libraries to use the Arduino hardware if your requirements dictate otherwise.

The name “Arduino” is a copyright held by the original team based in Italy that originally built the hardware, the IDE (integrated development environment) and the software libraries. It was designed to be a platform for *physical computing* usable primarily by non-programmers (artists, hobbyists, teachers, students, etc.). Another goal of the project was that all of the hardware designs (except those for the chips themselves) as well as the software tools are open-source – free to copy and distribute without any licensing restrictions. By making an easy-to-use, easy-to-make platform, they hoped that it would enable a wide variety of interactive and creative uses by an even greater variety of users. Its popularity and use has spread much wider than originally anticipated. It's estimated that roughly 400,000 Arduinos and Arduino-compatible boards have been made and/or sold by the end of 2011, and could reach one million by the end of 2012. There are also dozens of variations of the Arduino serving specific niche purposes such as robot/motor control, miniaturized applications, UAVs, sensor networks, etc.

Another attraction of the Arduino platform is that there is almost no configuration necessary to get started. Once you have downloaded and installed the software on your computer, you can load, compile, download[[1]](#footnote-2), and run a sample program in one or two minutes – as you'll soon see.

Arduino programs are called *sketches*, reflecting the vocabulary of the artist community. You will still often hear them called *programs* (including here) and not *sketches*, but at least now when you hear “sketches”, you'll know what is being talked about.

Besides its ease-of-use, the Arduino was one of the first complete development systems to cost around $30. The current, standard Arduino is still priced around $30, but small, barebones Arduino-compatible boards can be purchased for $11[[2]](#footnote-3). In fact, the hardware platform is simple enough that even cheaper, barebones versions can be handmade using a variety of construction methods[[3]](#footnote-4). Chinese clones of reasonably current standard Arduinos may be found on eBay for less than $20.

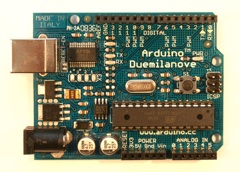
Another advantage is that the Arduino development environment can be run on either Windows, Linux and MacOS for no cost other than for the hardware. All of the software is freely downloadable in one bundle from [www.arduino.cc](http://www.arduino.cc/), the website that is ground-zero for all-things-Arduino. This website should earn a prominent bookmark in your web browser.

Before switching to the Arduino, we used the Atmel STK500 development platform in ME 106. The STK500 allows you to use many of the hundreds of 8-bit MCUs that Atmel currently makes, one of which is used in the Arduino. The STK500 accommodates chips with 8 to 40 pins, and has optional attachment boards supporting chips up to 128 pins. In addition to being substantially more difficult to configure and use than the Arduino, the STK500 requires RS-232 ports to connect to a PC – which are almost obsolete nowadays, having been replaced by USB. While it is a still a fine board for intermediate-to-advanced users, it is less than ideal for uses outside of Electrical and Computer Engineering.

The Arduino boards that we are using are from the group of “standard” Arduinos, in particular, the Arduino Duemilanove (Italian for “2009”) and the current Arduino Uno, the former shown in Figure 1. What characterizes this class of Arduino boards is the layout of the four rows of female headers across the top and bottom, and the single 2x3 header along the right edge. Any Arduino following this standard layout can connect to a large number of expansion PCBs (printed circuit boards) that plug on top of the Arduino called *shields*. Most Arduino-family boards follow this classic layout, but some do not, mostly for size or cost reasons.

The microcontroller unit (MCU) currently used on the most common standard Arduino is the Atmel in a 28-pin PDIP (plastic dual-inline package) package. All I/O pins may be used as digital inputs or outputs, but only six may be used specifically as analog inputs. Additionally, six of the digital I/O pins may be used as PWM (pulse-width modulated) as pseudo-analog outputs. All the technical information about the ATmega328P may be found in its [datasheet](http://www.atmel.com/Images/doc8271.pdf). You will have to refer to this datasheet often during the semester, so you may also want to bookmark it (or download it, since it's 21 MBytes).

The analog output pins are located on the bottom-right six-pin female connector labeled:   
**ANALOG IN 0-5**. The digital-only pins are located on the top labeled **DIGITAL 0-13**.



ATmega328P MCU

Reset button

Analog pins header

Digital pins header

Power-Ground header

Pin 13 LED

USB connector

Barrel jack

Figure . Arduino Duemilanove (source [www.arduino.cc](http://www.arduino.cc/), 2010).

The board is most often powered from the USB connector on the upper-left side during software development. The USB connection is the programming link over which programs compiled on your computer are transferred to the MCU. This connection also may be used to exchange data between the board and your computer.

For greater portability, the board can be untethered from the computer and separately powered via the 2.1 mm barrel jack in the lower left corner with either a battery pack or a 6-20 VDC (7-12 VDC recommended) AC-to-DC wall adapter. You may also connect a similar voltage source to the Vin pin in the power-ground header. The final, but less used, alternative is to attach a regulated 5V power supply directly to the 5V POWER header pin near the center-bottom of the board.

The Arduino board comes with a single LED, often called the *Pin 13 LED* because it is electrically connected to Digital Pin 13. This LED is the board's only built-in indicator accessible to programs. As you will soon see, you may bring more of your own input and output devices to the party and connect them yourself. A virgin Arduino board comes pre-programmed with a sample sketch called *Blink* that flashes the Pin 13 LED once per second. This feature is handy because you can easily check the general health of your new board simply by plugging it into USB, and the flashing LED demonstrates that the board is mostly working.

On the board is also a single reset button that causes the program to restart when pressed, and two more LEDs that flash as data is passed back-and-forth over the USB connection.

The remaining 6-pin female header near the bottom left of the board is the multi-connection power header, labeled **POWER**. From this header you can find two taps to the ground reference (GND), access to the RESET signal, a tap to a small 3.3V source produced by the USB interface chip, the 5 V source that is used to power the MCU, and the raw input voltage (Vin) from whatever power supply is being used to power the board. When using a solderless breadboard or shield, power is taken from this header's 5V and GND pins.

### General precautions

As with any piece of non-consumer-grade electronics equipment, you have to be somewhat careful to avoid frying electronic components. Some types of components, mostly semiconductor ICs (integrated circuits) ***can*** be delicate and easily damaged by electrical shorts and static discharges. In general, before touching any circuit board with integrated circuits (ICs), make sure that you have dissipated any static charge that you may have accumulated on your body. The best way to do this is by using an ESD wrist strap that has been connected to a good earth ground and by placing your circuit board on a grounded ESD mat. If you don’t have this or similar ESD equipment (we don't), it's generally sufficient to touch a grounded metal surface like the metal frame of the lab bench before you handle the circuit board. Static charge is especially easy to accumulate during the dry winter months, simply by walking across the floor.

Handle all PCBs by their edges, not by grasping the faces of the printed circuit board or its components. Finally, always wash your hands after handling electronics because lead and other heavy metals in the components and solder connecting them are hazardous in sufficient quantities.

### Getting to know the Arduino website

As previously mentioned, the main Arduino website is [www.arduino.c](http://www.arduino.com/)c. From here you can [download the software bundle](http://arduino.cc/en/Main/Software) that runs on your PC, find the [reference pages for programming](http://arduino.cc/en/Reference/HomePage) with the Arduino libraries, [learn to use the IDE](http://arduino.cc/en/Reference/HomePage), and search the central repository for community-contributed software modules in the [Playground](http://www.arduino.cc/playground/). The [Learning](http://arduino.cc/en/Tutorial/HomePage) section of the website is probably the first place you should start exploring after [Getting Started](http://arduino.cc/en/Guide/HomePage).

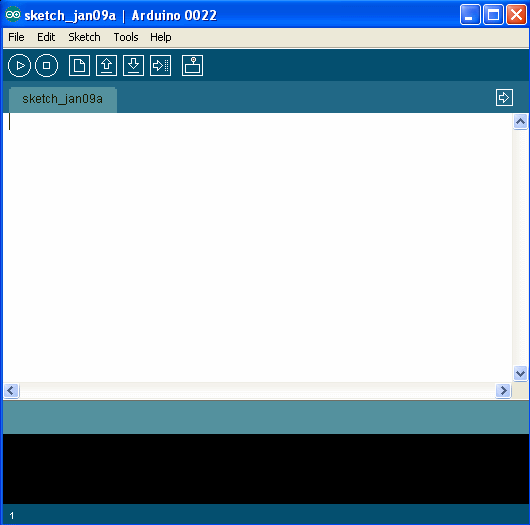
### Downloading and installing the Arduino software

Although you will not need to do this on the lab computers, you will have to do this on your own computer if you either choose to use it in the lab or with your own Arduino at home. Depending on which version of the Arduino board you are using, the installation procedures vary slightly, but are very easy, ***if you follow the installation instructions****.*  Note that different drivers are required for the most recent Arduino Uno versus the Arduino Duemilanove, so if you're planning on using both on one computer, both driver installation procedures will have to be performed[[4]](#footnote-5).

### Configuring the Arduino IDE

At this point you should connect the Arduino to the computer with the USB cable. After initial set-up, the computer should issue its ascending-beep to acknowledge that the board has been connected. After launching the Arduino IDE, you should see a window like that shown in Figure 2 below.

Figure 2. Screenshot of Arduino IDE



Now you have to configure the IDE so that it can communicate with the Arduino board. Please note that ***every time that you connect an Arduino over USB, you may have to go through the same configuration procedure*** (at least with pre-Uno boards). Additionally, if you cause the Arduino to hard-reset (for instance, because you created a temporary short), that will also reset the USB connection. Hard resets are functionally equivalent to disconnecting and reconnecting the cable or board. Fortunately, pressing the reset button on the Arduino board (or shields) will *not* reset the USB connection.

There are only two settings that need to be configured, and one of them rarely changes. The first setting is the *board type*. If you are using an Uno board, under **Tools->Board** select ***Arduino Uno.*** This option is probably already the default, and this setting is retained from the last time the Arduino IDE was run. If you are using a Duemillanove board, under **Tools->Board** select ***Arduino Duemilanove or Nano w/ATmega328***. Not setting the board type will cause program downloads to the board to fail.

The other option, under **Tools->Serial Port**, specifies the COM port that the USB driver should use to communicate with the Arduino. This one is annoying because every time that your USB configuration changes, your Arduino board may move to another COM port. There are three ways to figure out which COM port to use:

1) try to download to each of them,

2) For Windows only – grope around in ***Start->Control Panel->System->Hardware->Device Manager->Ports***, looking for an entry labeled *USB Serial Port[[5]](#footnote-6),*

3) remember which USB connections aren't connected to the Arduino (but to mice, etc.) and use the one that changes

Trying a COM port using the trial-and-error method takes approximately fifteen seconds per attempt, so it is up to you how you want to spend your time determining which COM port is the right one to use. If the Arduino is the only USB device on your system then it's easy because you will only ever have one option to choose from. If *no* choices are offered in the Serial Port list, unplug and replug your board and one should show up. If one does not, then you either have a board that is toast, a bad cable, or hardware that is connected to your board that is preventing the Arduino from functioning correctly, like a short circuit or something else that's drawing an excessive amount of current even when the board is off.

Take a moment to read just the first section of [Environment](http://arduino.cc/en/Guide/Environment) from the main Arduino website through the explanation of the IDE's icons. After doing so, it should be pretty obvious that this IDE is a very simplified development environment. If you have ever used Microsoft Visual C/C++, Eclipse, NetBeans, Ch or any other IDE, this one is really, really simple. Besides just six icons and five menubar choices, there just isn't much to it.

The only IDE button that may need some explanation is the last one, the *Serial Monitor*. Clicking on this button opens a separate window that is usually used to display messages printed from Arduino programs, similar to using printf() in C programs, or cout in C++. This feature is useful for prompting users to perform some action, for printing debug information from your sketches, or for sending keyboard data to the Arduino as input.

### Running your first Arduino program – Blink

One of the nice features of the Arduino development environment is that a large number of examples come bundled with it, under ***File->Examples***. Some of the feedback that we've gotten in past semesters (when using the STK500) is that students wanted more examples – well, here they are! These examples demonstrate many of the functions contained in the Arduino software libraries, and how to communicate with some of the more popular (or early) shields. We're going to show you how to get started with the Arduino software and hardware with the most useful and well-known example, *Blink*.

If you are not running the IDE, go ahead and launch it now. After the IDE starts, it always begins with a window containing a new, empty sketch named *sketch\_<currentdate>.* The assumption is that you are going to start writing your own program in this window, but this is not always the case. As you will see, any sketches that you open are opened in another window rather than in the current window, leaving you the original window to close and get out of the way.

Select ***Files->Examples->Basics->Blink***, and then close the original*sketch\_<currentdate>* window. This *Blink* example is the sketch that is programmed into new Arduinos before sale. However, because someone else may have used the Arduino before you, you're going to reload this sketch into this Arduino yourself.

There are two ways to compile a sketch, one with, and one without loading the program into the Arduino.

To compile a sketch *without* loading it on the Arduino

Click the right- arrow ***Verify*** icon,

or select ***Sketch->Verify/Compile***, (keyboard shortcut **CTRL-R).** This will compile your code *without* uploading it to the Arduino. You might want to use this to make sure that your program is at least syntactically correct before continuing with more code changes.



To compile a sketch and load it on the Arduino in one step

Clicking the right-arrow-to-header ***Upload*** icon,



or select ***File->Upload to I/O Board*** (keyboard shortcut **CTRL-U).** Both methods will compile, upload and run the sketch in one step. Since this is most often what you want, as is the case now, do this now.

You should see the following two areas in the bottom part of the IDE window, a blue-gray status message section, and a detailed message/log section:

In succession, you should see the following series of messages:

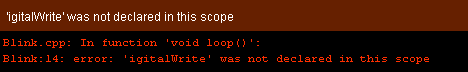


1. During the compilation process, the status message “Compiling...”
2. After successful compilation, the logged message “*Binary sketch size: ...*”
3. During upload, the status message “*Uploading to I/O Board...*” followed by a few-seconds-long flashing of the TX/RX LEDs on the Arduino (next to the Pin 13 LED). If either your COM port or board selection is incorrect, the upload will time-out, and you will likely see an error from the “avrdude” program in the logged messages window.
4. After uploading, the status message “*Done uploading*.”

If all goes well you should see the Pin 13 LED flashing with a period of two seconds.

If a sketch does not compile and upload successfully, you will get errors in both message windows. Explore this behavior by deleting the “d” character in the first call to digitalWrite() within the function “void loop()”, then click the ***Upload*** icon again. You should see the following:

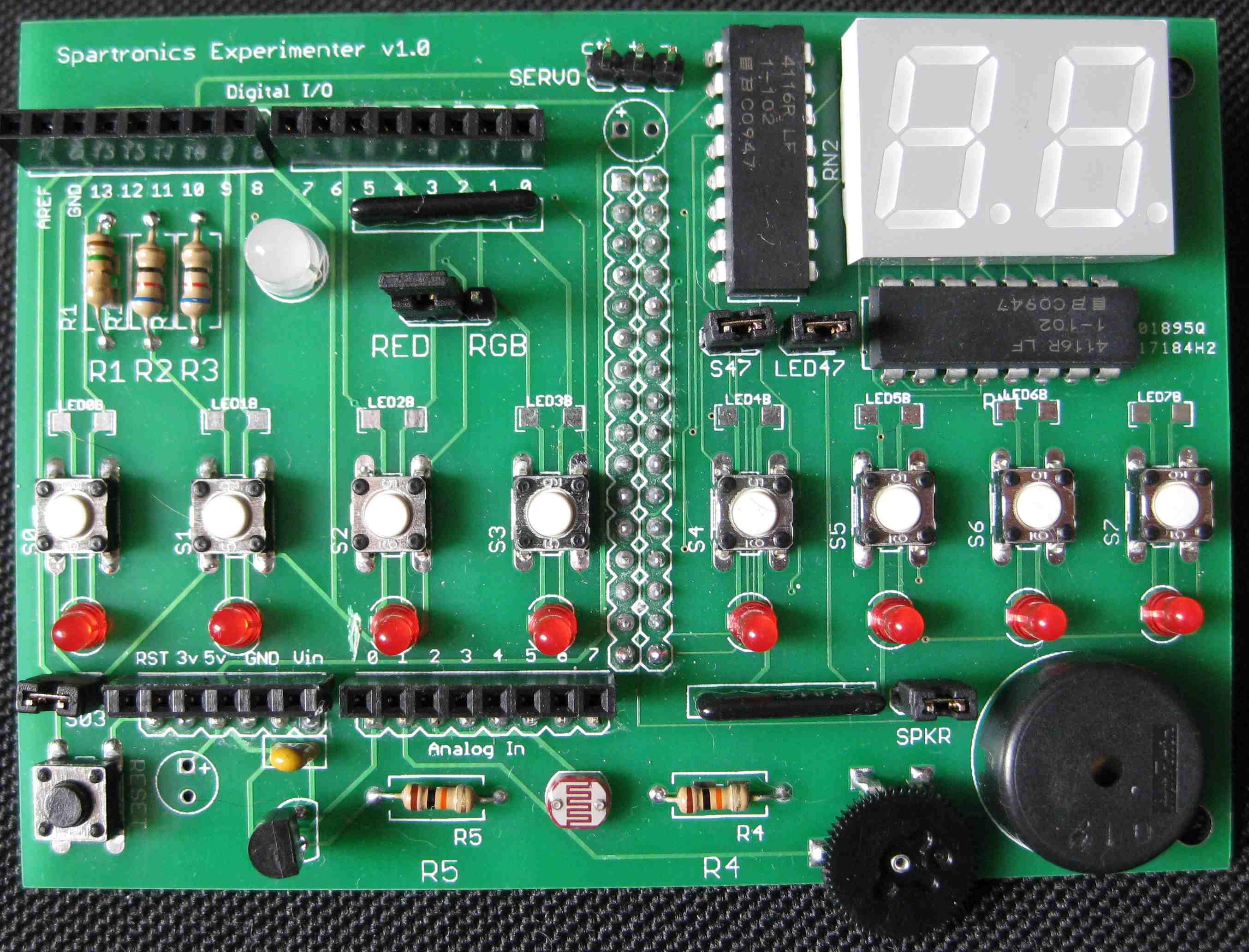
Looking at the Arduino, you should see that it is still running the *Blink* sketch. The point here is that if you do not pay close attention to the messages in the IDE, you may believe that you have made a valid change to your program, and that the new version has compiled, uploaded, and launched successfully. This is an easy mistake to make if you are not paying attention, *so pay attention*! Otherwise at some point you may introduce an error to your program, and not realize that it was not the most recent change but an even earlier change that hosed your code. Do not let yourself fall into this trap!



Examining the beginning of each line of the error messages, you should see that an error was detected at Line 14 of the *Blink* source file. You may remember from your C programming class that interpreting error messages can sometimes be a little challenging. The key thing to realize is that the reported line numbers are the places at which the compiler has realized that there has been an error by this point, but are not necessarily the locations of the errors themselves. You can only be sure that the error is *on or before* line 14. In this trivial case, the error *is* on the reported line. However, errors like missing semi-colons or unmatched curly braces tend to not be identifiable until many lines later, and in some cases, not even until the end of the source file. For similar reasons, when faced with a series of compilation errors, it is usually best to deal with the *first* error first, as subsequent errors are often due solely to not being able to recover from the earlier error(s).

### More than the Pin 13 LED – the Spartronics Experimenter Shield v1.0

Clearly there is a need to have more than just one LED to learn about the range of things can be done with a microcontroller. Although there were already a small number of ready-made Arduino shields for educational use back in 2009, we did not find any that fit most of our purposes when we introduced the Arduino into ME 30 and ME 106. So we made our own, and that is the Spartronics Experimenter Shield (SES), shown below.



RC servo header

RGB LED

Tactile buttons

Red LEDs

Reset button

Rotary pot

Piezo speaker

Light-dependent resistor

Temperature sensor

Figure 3. Spartronics Experimenter Shield (SES) v1.0. The SES provides a variety of I/O components and sensors on a printed circuit board that mounts on top of a standard Arduino or ‘mega-Arduino’.

The SES was originally designed for an Arduino-compatible board called the [Seeeduino Mega](http://www.seeedstudio.com/depot/seeeduino-mega-p-717.html) – which we are not currently using for either ME30 or ME106. Because all the "Mega" variants of the Arduino have many more I/O connections available to them that are not present on the standard Arduino, the SES can make use of them. Consequently, neither the dual 7-segment LED display in the upper-right corner, nor the rightmost four red LEDs and buttons are available for use with your standard Arduinos. Other than that, everything else on the SES is available to use.

Note that there is a newer v2 version of the SES that is smaller than the original that omits the dual 7-segment LED and the rightmost four buttons and LEDs, but is functionally otherwise exactly the same as the v1 board pictured and referred to here. You may be using one or both versions of these boards this semester, but rest assured that they behave exactly the same when used with a standard Arduino, except that the original is obscenely large compared to the newer version.

First, a quick rundown of the devices on the SES (See Figure 3.). Counter-clockwise from the top-center are the RC servo header, the RGB (red-green-blue) LED, four tactile buttons (meaning you can feel a “click” when pressed), four red LEDs, the reset button, temperature sensor, light-dependent resistor (LDR), rotary potentiometer (*pot*), and piezoelectric speaker.

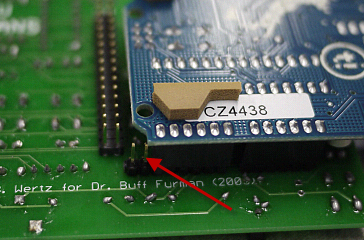
A usability problem we discovered with this first version of the SES is that it can be difficult to see the Pin 13 LED and RX/TX LEDs on the Arduino unless you look at them at the right angle between the boards. This is an annoyance that we've fixed on the v2 boards by placing a hole on the SES that you can look down through at the LEDs on the Arduino board.

*Connecting the Spartronics Experimenter*

The SES attaches to the Arduino just like pretty much every other shield, with downward-facing male headers that plug into the upward-facing female headers on the Arduino. All power and I/O connections are brought up to the shield from the headers below. The female headers on the Arduino have also been replicated on the SES, so you can continue to access signals by sticking 22-gauge wire into the female headers. In all cases, on-board jumpers may be used to disconnect individual devices on the SES should you need to use their respective I/O pins for other purposes.

As with almost all printed circuit boards, it is always recommended to power the Arduino off when plugging/unplugging shields, making new component connections to a board, or adding probe wires if there's any possibility of creating a short. For these reasons, it's safest to always disconnect the board when making any changes to it.

*For the remainder of this lab exercise there is no reason to disconnect the Arduino from the SES once they have been joined, and to limit wear-and-tear on the connectors, we prefer that you* ***do not disconnect them for the remainder of this lab.***



**Figure 4. Extra pins on one side of SES.**

After detaching the Arduino from the USB cable, *carefully* attach the SES to the top of the Arduino. Ensure that both sets of parallel headers are aligned by looking that them from all angles before applying pressure to fasten the two boards together. Note that there are two extra pin headers on one of the six-pin headers that do not connect to the Arduino (See Figure 4). Once they are together, reattach the USB cable and listen for the tell-tale USB connection tones from computer speaker. If you do not hear them, try reattaching the USB cable again. If it still does not work, get help from a TA before continuing.

At this point you should be able to peek under the SES and see the Pin 13 LED flashing once per second, just as it did with the Arduino by itself. This behavior is typical for a microcontroller system, that is, to re-run the same program after a reset or power-on. An MCU can do this because the memory in which the program is downloaded is *non-volatile*, which means that it is not erased when power is removed. Program memory is *flash memory*, the same used in SD/MMC/CF cards and USB memory sticks. Until reprogrammed with something else, the same program will execute again when power is applied.

There is a subtlety here that you may not appreciate. Consider the case where you are working on a new project and you wire up your own circuit to an Arduino and power it on. Because you do not know what is running on it from whomever programmed it last, you run the risk of damaging what you have connected the Arduino to when the previous code runs. For this reason, you should wait until you have some of *your own* code that you (kind of) trust before you power the Arduino when it is connected to your circuit. The other implication is that when designing circuits, you should try to do so in such a way that even in the presence of some unknown code, or your own software errors, the hardware should not self-destruct under reasonable circumstances[[6]](#footnote-7).

### Arduino example – Blinkin' SES-style

*Appendix B* contains tables that describe the relationship between the Arduino's I/O pins and the devices on the SES. You should notice several things:

* The SES does not make use of pin D13 (digital pin 13) for anything
* The four LEDs are connected to pins 11, 9, 6 & 3
* The red, green, and blue parts of the RGB LED share pins with the red LEDs. So when *Blink* runs, it does not affect any devices on the SES, and only continues to flash the LED on the Arduino under it.

We are going to change the *Blink* sketch to blink one of the LEDs on the SES. Ensure that the jumper on the SES board that is labeled **RED-RGB** is connecting the rightmost two pins of the 3-pin header, which selects the ***RED*** configuration. If the jumper is not set correctly, unplug the USB cable to remove power to the Arduino, move the jumper, and reconnect the USB cable.

Modify the Blink code by changing all instances of 13 in the code to 3, then upload the code, and report what happens. While this sketch is still running (or power-off and on if you are more comfortable, but this change is pretty safe to make while powered), change the jumper to the **RGB** position and also report the outcome. This should help make clear what the function of the **RED-RGB** jumper is. Repeat this section, substituting 6 for 3, then 9 for 6 in the code and report what happens in each case.

When you have the time, see *Appendix C* for a discussion of why *Blink* is not a perfect example of how to write even a simple program.

### Arduino example – Multiple LEDs and Buttons

Now we are going to demonstrate how to write your own sketch using the Arduino IDE, and specifically how to solve problems that you may run into all semester if using the shared lab computers.

As you have already seen, when you first start up the IDE, it creates an empty sketch with a unique but fairly useless default name. Select ***File->New*** to create a new sketch. Close the window with *Blink* in it. It should ask you if you want to save the (modified) sketch, but to do so, you’ll have to name it something else, because *Blink* is read-only. [Allowing you to save the modified *Blink*  would clobber the sample that everyone is sharing on this computer!] The IDE will pop up a *Save sketch folder as* dialog which will allow you to save this sketch elsewhere. Change only the ***File name*** field to a unique name like *Blink-Dave23* and click *Save*. From the Windows environment, open up a File Manager window for ***My Documents*** (Desktop ***Start->My Documents***). Double-click the ***Arduino*** folder in the File Manager window, then double-click the folder whose unique name you used for your copy of the *Blink* sketch.

You should notice a few things:

1. ***My Documents\Arduino*** is the default location for the IDE to store sketches.
2. Sketches are placed in their own directories (folders), which are given the same name as the sketch. The source code for the program is in the same folder in a file named ***<yourSketchName>.pde[[7]](#footnote-8)***
3. ***My Documents\Arduino*** is writable by everyone using this computer, so if you do not choose a good, unique name for your sketches, you are very likely to get yours confused with someone else's from another lab section who used the same computer.

Now that you have successfully saved your modified copy of *Blink*, we are done with it, and you can close that window, leaving the new one you created. Back in that remaining Arduino IDE window, select ***File->Save As*** and choose a unique name for this sketch – something like *test\_<whatever>* is just fine. This establishes the name of the sketch for subsequent saves, as well as its location on disk[[8]](#footnote-9).

Enter the program sample in *Appendix D*, either by cut-and-pasting it from this document, or downloading ***blink4.pde*** from the lab website. Compile, upload and execute it, and describe its operation.

Now take a few moments to look at the code. Hopefully there is nothing here that is too scary (but if there is, plan on breaking out that C-language book from your last programming class!) -- an array, a use of *sizeof()*, a couple of simple for loops, and a few calls to Arduino-specific functions that may be new to you. Notice that loop() does two things, it sequences through the list of LEDs and checks the status of Button0 (the leftmost button, also labeled **S0**) on the board.

There are two things to note about sensing the state of the button. First, note that the return value from digitalRead() is checked against the value **LOW** in a handy little *while* loop with nothing inside of the *while* loop body. This is a very common way of waiting for something to happen, and *blocking* (not continuing) until it does. You should read this *while* statement to mean “while the digital input value of the Button0 pin is **LOW**, do nothing”, or conversely “continue when the input is **HIGH**”.

Second, although you might guess that the button would be **LOW** when unpressed, and **HIGH** when pressed, this is clearly not the case here because the sketch is running freely when the button is unpressed. The circuit connected to each button has been designed so that its output is **HIGH** when the button is not pressed, and **LOW** when it is being pressed. For the time being, simply accept that this is the way it is (in fact, it's a fairly typical configuration for buttons), and that when getting to know a button, you have to determine whether it is *active-high*, or *active-low*. In this case, the button is *active-low*, that is, when it is pressed (i.e., *active*), its value is **LOW**.

Go ahead and press Button0 to see what happens. Describe your observations including how long and *where* the flashing sequence is interrupted, and how this behavior matches with the code. Take a few minutes while you are here to be sure that you understand how this code works, because you will be returning to it near the end of the lab.

### Arduino example – AnalogReadSerial

Open and compile/upload the example sketch *AnalogReadSerial* from the *Basics* sub-menu (***File->Examples->Basics->AnalogReadSerial***). Note that there is a call to the function analogRead() within loop(). Although you will learn a lot more about this later, analogRead() reads/senses an analog voltage on an I/O pin and returns a (10-bit) integer related to the value of that voltage. In this case, the pin that it is sensing an analog voltage is A0, which is a synonym for both the value 0, and the name "analog pin 0". If you consult Appendix B, you will see that on the SES, analog pin 0 is occupied by the temperature sensor.

In this sketch, we want to specifically point out the use of *Serial*. *Serial* is an Arduino software component (*object*, if you are familiar with C++ or Java) that represents the serial data connection (using USB) between the PC and the Arduino that we mentioned earlier. Serial.print\*() functions[[9]](#footnote-10) are used to send (any type of) data back to the PC for processing. A common use is to send text messages, and for displaying text there is a built-in message window in the IDE, the *Serial Monitor*.

Click the right-most (*Serial Monitor*) icon in the IDE. This resets the board (starts the sketch over) and opens the *Serial Monitor* window. Here you will see the data (text messages) received from the Arduino over the USB link. This is a feature that you should become very familiar with this semester. To find out everything there is to know about *Serial,* which is not excessive by any means, consult the documention for it in the [Arduino Reference](http://arduino.cc/en/Reference/HomePage).

Record the range of values that you see printed in this window. Now use your fingers to firmly pinch the temperature sensor, *but do not to move or bend it*. The temperature sensor is the plastic component that sticks up slightly from the board located between the reset button in the bottom-left corner of the board (it's near the upper-right corner on v2 boards) -- see Figure 1. Continue to pinch the sensor for at least ten seconds, or until you see the values printed in the serial monitor stop changing. Again, record this higher range of values shown in the Serial Monitor window. The temperature in degrees (Celsius for SES v1 and some v2 boards, Fahrenheit for other v2 boards) is a function of the displayed value, *n,* according to the function in equation 1[[10]](#footnote-11):

 (1)

Compute and document the lowest and highest values that you observed in both degrees Celsius and Fahrenheit.

Change the symbol ***A0*** in the code to ***A1***, and upload this modified sketch. Appendix B specifies that analog pin 1 is connected to the potentiometer on the SES. A potentiometer is a variable resistor, and the one on the SES is a black rotary dial on the bottom edge of the board. Potentiometers are often used to allow users to choose or adjust an analog value along some continuum – for instance to set an output volume level on a speaker. Again use the Serial Monitor to record the minimum and maximum values read by analogRead() as you sweep the dial over its full range. You will need to use these values later.

Change the symbol ***A1*** in the code to ***A2*** and upload it. Analog pin 2 is connected to the photoresistor on the SES. A photoresistor is also a variable resistor, but its resistance changes depending on how much light is falling on its face. The photoresistor is located on the SES v1 in the bottom-center of the board between the two resistors, or the right edge on the SES v2 boards. Once again use the Serial Monitor to record the minimum and maximum values read by analogRead() as you use your hand to throw a shadow over the photoresistor. You can/should try to block-out the maximum amount of light by gently covering it with the tip of your finger. Record the range of values from uncovered to fully-covered. You will also need to use these values in a later exercise.

With the help of the SES you have now seen three examples of simple analog sensors, and how easy it is to capture their values with an Arduino.

### Arduino example – Analog (PWM) output

After setting the **RGB-RED** jumper to **RGB**, go to the examples and open, compile/upload, and run the Arduino *Fading* example under the *Analog* sub-menu.

This example is somewhat more complex because there are two different things going on at the same time here. The first uses a technique that you will soon learn about called *pulse-width modulation* (PWM) that is often used to generate a pseudo-analog (i.e, not *really* analog) output using digital values that are switched on and off quickly. LEDs, like electric motors, *can* run at something other than zero-power or full-power by feeding them a reduced voltage or current, but often they are not driven that way[[11]](#footnote-12). Rather, what is done is that full-voltage is pulsed to it so quickly that the net result is some level between the two extremes of 0V and 5V. The extent to which the component is driven is a function of the fraction of time that the pulse is powered-on. The ratio of the powered-pulse length to the period of the entire pulse is called its *duty cycle*. Sending a train of pulses that are on for 90% of the time results in a net output of 90%, or 4.5V, of the full-scale output voltage (5.0V) over time.

The Arduino library function *analogWrite()* generates a PWM waveform on a given pin. However, rather than specifying a percentage (0-100) as its parameter, it requires a value from (0-255). From the Arduino Reference, what features and limitations are present with AnalogWrite()?

It is probably not obvious that a motor or LED is being supplied a PWM signal, because you cannot percieve it. This is because the frequency of the PWM is often hundreds or thousands of cycles per second. However, what you do see both visually and in the code, is that the duty cycle is varied from 0% to 100% and back to 0% (actually 0/255 to 255/255 to 0/255), one level at a time, only resting at each brightness level for just 30ms at a time. What is actually going on in this sketch is the following:

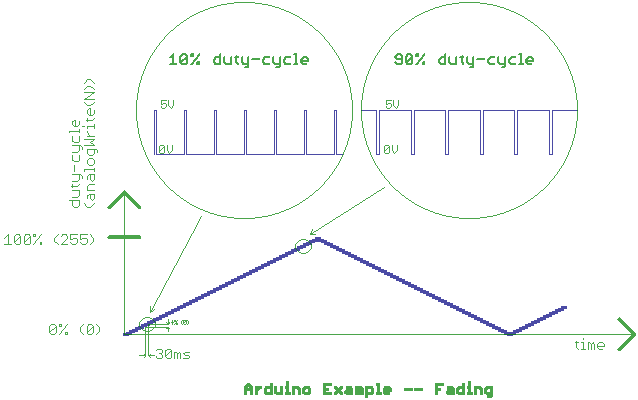


Figure 5. PWM signals generated by AnalogWrite() function. The AnalogWrite() function will generate a PWM signal having a duty cycle that varies between 0 and 100% in correspondence with a parameter between 0 and 255.

The end result is that you are seeing what looks like the analog (voltage or current) input to the LED varying from low to high, without being able to see the micro-transitions (the PWM signal) within each of those levels of brightness. However, what you can see is that the duty cycle is changing. We’ll use the oscilloscope next to observe the signal that is powering the LED. As shown in Figure 6, insert a black wire into the GND female header (nearest the upper-lefthand corner of the SES), and any other-colored wire into the header for Digital Pin 9, as shown in the photo below. As the program runs, observe the behavior of the output signal over time. Use a digital photo to record the output at both low and high duty cycles.

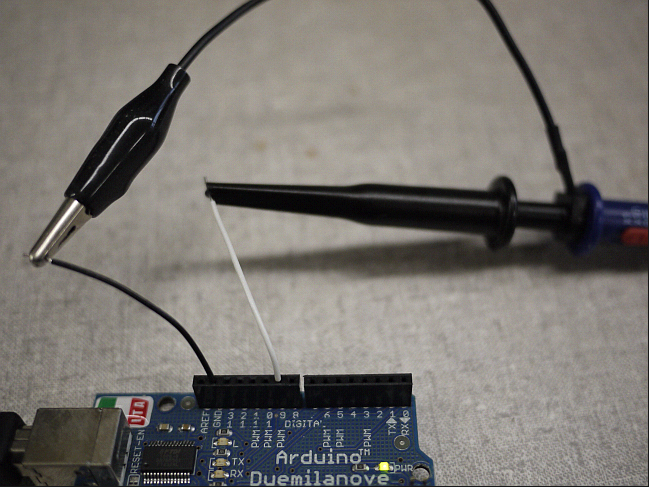


Figure 6. Connection of the oscilloscope to display a PWM signal.

### Arduino example – Servo

The word *servo* in the context of mechatronics usually refers to a motor having a single control line interface that can control the angular orientation of its output shaft. Hobby servos are very commonly used in RC (radio-controlled) airplanes, cars and boats, and are frequently used in robotics applications. The majority of servos have a range of motion of 0-180 degrees. The desired angle of the servo is specified by feeding it electrical pulses of a specified length and frequency on its control line. The other two pins on the servo's three-pin connector are for a 4.5-6.0V power source and ground to drive the motor and its embedded circuit. The control circuitry inside consumes very little current and can be connected directly to an Arduino's I/O pin. You'll learn a lot more about servos and other motor types later in the semester.

The Arduino software library contains a very easy-to-use *Servo* object that is used to control a servo using one output pin from the MCU. It is documented in the *Servo* section of the [Arduino Libraries Reference](http://arduino.cc/en/Reference/Libraries). Among other useful background information, this page describes in detail the limitations your programs may have if the *Servo* object is used. Points like these can be *critical* to appreciate when designing projects, from both a hardware and software standpoint. **It cannot be over-emphasized that you need to fully read and understand the use of documented functions before applying them in an application[[12]](#footnote-13).** What limitations are there on the use of the Servo library when using a standard, Atmega328-based Arduino?

Upload the *Sweep* sketch from the *Servo* sub-menu of the Examples to your board. Modify the sketch by replacing the 9 with 10 in the call to myservo.attach(), and then upload the modified sketch to the Arduino. Disconnect the USB cable after the transfer is complete.

Connect your servo to the servo header on the SES by orienting the female plug on the servo, so that the header pin labeled ‘**-**’ aligns with the black wire, the center pin with red power wire, and the last pin labled ‘**ctl**’, with the signal wire (usually white). The servo header is the three-pin header labeled **SERVO** at the center-top edge of the v1 SES (upper-left corner on the SESv2), with terminals labeled ***ctl***, ***+*** and ***-***, for the I/O control, **5V** and **GND** pins, respectively. The *ctl* pin is electrically connected to one of the I/O pins (which one?) on the Arduino. The power required by a small servo may be supplied safely from the five-volt regulator on the Arduino -- as long as it is not drawn through an I/O pin. A larger servo may require its own separate power supply to be connected to its (female) power and ground terminals, while still allowing its control line to be connected to the Arduino.

Connect the USB cable to the Arduino, and the sketch will begin immediately.

Describe the functioning of the RC servo as controlled by this sketch. Take a moment to look at the code and try to understand how it works. You'll be revisiting this code later in the lab.

### Arduino example – Audio

Open the example *toneMelody* (under the *Digital* sub-menu), and modify it using information in Appendix B to figure out how to access the speaker and play tones on the SES. As usual, you should consult the Arduino Reference to understand how any functions work that are unknown to you. Document any changes that you made to the code to get it to work on the SES.

While looking over the code, hopefully you noticed that there are symbols (names or variables) that are used but do not seem to be defined anywhere in the sketch's source file (like NOTE\_C4). One thing you may not have noticed is that this sketch is composed of *two* tabs in the IDE, one for *toneMelody* and one for *pitches.h.*  If you select the tab for *pitches.h,* you will see a bunch of definitions, some of which you should see used in the sketch's code.

Separating executable code from a collection of definitions is a very common method of organizing software code. Often large numbers of symbolic names are defined in one common place and are used by all sketches using that module. For example, there is a header (*.h*) file on the course website, ***SES.h,***  that lists all of the pin assignments and other features of the SES for sketches to use. If you flip back to the *toneMelody* sketch tab, you should see how the sketch references the contents of *pitches.h*. In short, the compiler sucks-in the contents of the referenced file where it is *#include*d, just as if it were present there in the sketch itself.

Although you do not have to run it, take a look at the example *toneMultiple* under the *Digital* sub-menu, and compare it with how *toneMelody* is written. Assume that you wanted to play a minute's worth of tones, say 100-200 of them. Which of these two sketches would you base your sketch on, and why? Also, why would one sketch have been written to put all of its code in setup() and the other in loop()? Finally, can you think of a way that the *toneMelody* could have been written to *behave* the same way with all of its non-initialization code in loop() rather than stuffed into setup()?

Connect the oscilloscope to the SES like you did in the exercise for the *Fading* sketch, but use the ‘scope to monitor the speaker signal instead. (Hint: Set the voltage resolution to 2.00 V/div and Time/div to 2.00 ms/div.) As the program runs, describe your observations of the waveform relative to what you hear on the speaker. Use a digital photo to capture one of the waveforms after an oscilloscope displays the signal of a known tone. Use the oscilloscope waveform timing functions to measure the characteristics of your captured wavefrom, and relate them to the piece of code in the sketch.

### Breadboarding example – one LED and resistor

Clearly there are projects for which pre-assembled or off-the-shelf hardware does not already exist. For these situations, most people use a solderless breadboard to prototype circuits, and from there connect to a microcontroller.

Start this exercise by uploading the *Blink* sketch that you worked with earlier, and ensure that it is working properly with the Pin 13 LED. Disconnect the Arduino from the USB cable, because you will soon be making electrical connections between it and your own small circuit.

Figure 6 shows a diagram of a circuit that you should exactly reproduce on your solderless breadboard.

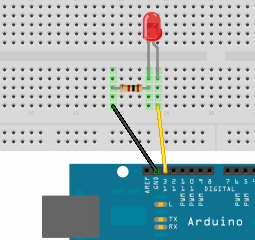


Figure 6. Connection of the Arduino to a circuit on a solderless breadboard. Build the circuit shown: black wire from the GND socket on the Arduino to the resistor (what value of resistance does the color code red-black-brown correspond to?), resistor to the shorter leg of the LED (cathode), and longer leg of the LED (anode) back to the Arduino pin 13 socket via a jumper wire of a color other than black (shown here as yellow, but could be something else).

The only non-obvious aspect of this assembly is that the red LED has one lead that is shorter than the other. This non-symmetry is key to the correct functioning (and non-toasting) of this component, as well as many others you will encounter throughout the semester. In particular, the longer LED lead, called the *anode*, is the positive side of the device. Take particular care when wiring your circuit to make this connection properly. As usual, when building or changing a circuit, power should be removed until your circuit changes are complete and have been checked, ideally by someone else.

Since the Arduino has already been programmed with *Blink*, all that is needed is to supply it with power. Rather than using the USB cable to supply the power, connect the 12V DC adapter (“wall-wart”) power supply to the barrel jack on the Arduino, and the supply to the wall socket. In similar fashion, one could use a 9V transistor battery (or any other battery that can supply a voltage between 6-20 VDC) with a 2.1 mm diameter plug to power the board.

Once you have assembled and tested your setup, demonstrate to your TA that it is working. Include a digital photo of the final circuit connected to your Arduino (even better with the LED actually on!).

**Software submission Words to Live By**

Beginning now (and throughout the remainder of your career) make your best effort to ensure that your code is an absolute joy to read by someone else other than yourself, using all of the techniques you learn about this semester. Ugly, obtuse, undocumented code will detract from your grade. You wouldn't hand someone a “finished” machined part with burrs and grease on it, so don't expect to do the same with your code.

You are also highly encouraged to write and keep your code “all nice and pretty” as you go, as it will make it easier on yourself, your lab partner, and any TA that you will need to ask for help (who may rightly *refuse* if you ask him or her to help you with your nasty code). Resist the temptation to knock out crappy code during the lab with the idea that you will pretty it up later before submitting your report, because you may not have the chance to retest your code before turing it in – never mind that you may actually break your code while "touching it up" outside of lab. Handing in untested code is not where you want to be going, so just don't go there.

Your best bet is to write your code before the lab period, taking time to document and comment it as you go. Then, while you are in the lab working with the actual hardware and debugging your code, you will personally see the benefit of “clean” code as you can easily navigate through what you have written earlier.

### Revisiting Blink4

Return to the *Blink4* sketch from earlier in the lab and modify the code to stop the flashing sequence as long as *any* of the four leftmost buttons are held down, and include (only) the modified portion code in your lab report. *Hint: you want to consider using the logical OR operator.*

### Putting *AnalogReadSerial* and *Servo* together

Using what you have learned from the *AnalogReadSerial* and *Servo* examples, create a sketch that sweeps the servo over its full 180 degree range of motion when the rotary potentiometer is turned back and forth – in effect, using the dial as a steering wheel to change the angle of the wheels in a vehicle. Demonstrate to your TA that your code functions properly, and include it in your lab report. *Hint: You may find the functions map()and/or constrain() useful for this exercise. You can find help on these functions in the Arduino Reference section of the website.*

### Putting the temperature sensor and PWM-ed RGB together

Using what you have learned about analog (PWM) output, create a sketch that produces the following behavior with the RGB LED when you touch the temperature sensor:

* At room temperature, full intensity green (100% duty cycle)
* At a slightly elevated temperature, the intensity drops to half-on green (50% duty cycle)
* At a slightly higher temperature, the color should be red with half-on intensity
* At or above the highest temperature threshold, the color should be red with full intensity.

You should be able to generate the three elevated temperature levels using the finger-pinch method you used earlier. If you cannot generate a wide enough temperature range with your fingers, ask your TA for help.

One thing that is probably not obvious about AnalogWrite() is that the output waveform, once started, continues on and runs in parallel with the rest of the code in your program. A PWM signal generated by the MCU hardware continues to run totally independently until you change the duty cycle, or use AnalogWrite(0) to turn it off altogether. This behavior will likely become evident when you make the transition between the green and red LEDs in this program. Demonstrate this code working for your TA when you are completed, and include it in your lab report. *Hint: You probably can get by mostly using if..else statements with comparison operators , along with a few analogRead()/analogWrite() calls sprinkled-in.*

### Appendix A – A little background on the Arduino platform

The “officially-produced” Arduino hardware is a small set of boards that use only a tiny fraction of all of the MCUs manufactured by Atmel. “Officially-produced” is used here to mean those boards that have been designed by the Arduino team. These are the only ones allowed to use the name *Arduino* for their products. Others making Arduino-compatible boards must name them something else. Although the Arduino team encourages others to make their own compatible boards (which is why they open-source their designs in the first place), others may only call their boards “Arduino-compatible”. This is the reason that many other boards are named *something*-duino – so that one can easily associate their product with the very popular Arduino.

The advent of the Arduino has helped to create a lot of pressure on competing MCU makers and their development system offerings – so much so that the average cost of entry-level development system has dropped from hundreds of dollars just a few years ago to probably around $50 at this time. Currently the cheapest retail price for a minimal development platform is $4.30 for an MSP430 Experimenter from Texas Instruments.. Because the competition is so fierce, it is now possible to go to a trade show and score one or more Arduino-class (or better!) hardware-software bundles for free, if only you are willing to sit through a 1-2 hour technical marketing presentations.

### Appendix B – Spartronics Experimenter Shield (SES) Arduino pin assignments

Table B1. Digital pin assignments (pin numbers in italics are PWM-able)

| **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | LED0 |  | LED1 |  |  | LED2 |  |  | LED3 |  |  |  |
|  |  |  |  | green |  |  | blue |  |  | red |  |  |  |
|  | SW0 |  |  |  | SW1 | SW2 |  |  | SW3 |  |  |  |  |
|  |  |  | servo |  |  |  |  | spkr |  |  |  |  |  |

Table B2. Analog pin assignments

| 0 | 1 | 2 | 3 | 4 | 5 |
| --- | --- | --- | --- | --- | --- |
| temperature | potentiometer | photocell |  |  |  |

### Appendix C – why the sample Blink sketch ain't so hot

### First, what they got *right*:

1. The code starts off with a brief explanation of the purpose of the program. Whether or not this initial comment block contains the name of the file is not so important (because it is usually the name of the file, after all).
2. The indentation of the program is well executed. Although the amount that every line is indented is somewhat a matter of personal preference, it should absolutely be in the range of two to eight spaces. Four is considered ideal by many programmers, because it provides a very clear horizontal indication of where you are, but not so much that if you are three to four levels indented, that you’re starting halfway across the screen.
3. Any general notices about the code are usually put here, like legal notices and attributions (if you started with someone else's code), and usually an indication of who authored the file, when, and the code's current version. Any easy way of comparing two versions (by date or version) of the same code to determine which one is the more recent is *imperative*.
4. The width of the code on the screen is reasonable, without one ungainly-long line requiring you to stretch the window (or 10-times worse, scroll side-to-side) to see it.

**What they *didn't get right*:**

1. Comments in the middle of a function are often more of a distraction than a help unless they are out of the vertical visual line of the code. Most of the time when you are reading code, you are trying to read the actual code, not trying to understand it for the first time. Having comments between lines of code is a distraction when all you want to see is code. Too many line-hogging comments means fewer lines can fit on the screen, and for most people, the more they can see without having to page back-and-forth is a huge productivity win.
2. Useful comments are useful (by definition!), but useless ones are at the minimum a distraction, and at the maximum a waste of your time and an insult to your intelligence. Comments that are necessary for the understanding of a function are generally best put in a comment block before the function itself, or to the right of an individual statement where necessary.

An example of a borderline useless comment is “initialize the digital pin as an output” when there is nothing else that that line could possibly interpreted as doing. However, the *classic* useless comment is:

c = c + 1; // add one to c

While we absolutely want to see commented code, too much of an otherwise good thing can be worse!

1. Any constant, scalar values, (usually) except 0 and 1 should ***not*** be hard-coded right in the code. Rather, they should be assigned some type of name that describes exactly what they mean. The reason for this is two-fold: first, it acts as clear documentation as to what that constant is for, and second, it makes modifying/tuning that value easier if the value exists in a single place, usually at the top of the file[[13]](#footnote-14).

You saw an example of point #3 when you had to change *Blink* from using LED digital pin *13* to pin *3*. Not only did you have to change the code in three different places, you had to carefully look through the entire program (and pretty much understand the whole thing along the way) to make sure that you found and replaced each and every constant that needed replacing.

The listing below demonstrates both of the most common examples of how one should handle coding constant values – either through the use of a variable with the const keyword or a #define.

The following listing shows how Blink should have been written ideally:

/\*

\* Repeatedly toggle an LED on and off, with DELAY\_TIME milliseconds

\* between each change.

\* We are using the only LED that is widely available, the one that

\* is built-in to most Arduino boards, the Digital Pin 13 LED.

\*

\* Last modified by Jack Black at the University of Rock.

\* Original written by Spinal Tap, and their versions went to 11.

\*

\* Version 12.0 – July 4, 2006

\*/

#define DELAY\_TIME 1000 // # of milliseconds between LED toggles

const byte theLED = 13;

/\*

\* One-time initialization function.

\* In our case, just setup the directions of all I/O pins used.

\*/

void setup() {

pinMode(theLED, OUTPUT);

}

/\*

\* Every time loop() is called, we toggle the LED on and off.

\* Because we return from this function, we'll immediately be

\* called again, and again.

\*

\* Because the LED is active-high and we want to start the

\* cycle on and finish with off, to do this we need to start

\* by setting the pin HIGH, and finishing LOW.

\*/

void loop() {

digitalWrite(theLED, HIGH);

delay(DELAY\_TIME);

digitalWrite(theLED, LOW);

delay(DELAY\_TIME);

}

### Appendix D – first DIY test program (blink4)

/\*

\* Intro-Arduino (Lab 2) first DIY program.

\* Flash the leftmost (4) LEDs in sequence, pausing

\* DELAYTIME\_BETWEEN milliseconds between individual LEDs,

\* followed by a delay of DELAYTIME\_AFTER after the last,

\* repeating forever.

\*

\* Written by Eric B. Wertz for ME106.

\* Last modified by Eric B. Wertz on January 13, 2011.

\* Version 1.02

\*/

#define DELAYTIME\_BETWEEN 100 // 100 msec (0.1s)

#define DELAYTIME\_AFTER 400 // 400 msec (0.4s)

#define PIN\_LED0 11

#define PIN\_LED1 9

#define PIN\_LED2 6

#define PIN\_LED3 3

#define PIN\_BUTTON0 12

#define PIN\_BUTTON1 8

#define PIN\_BUTTON2 7

#define PIN\_BUTTON3 4

// set of LEDs to use, and number of array elements

const byte ledPins[] = { PIN\_LED0, PIN\_LED1, PIN\_LED2, PIN\_LED3 };

const byte numberOfLeds = sizeof(ledPins)/sizeof(byte);

void setup()

{

byte i;

for (i=0; i < numberOfLeds; i++) {

pinMode(ledPins[i], OUTPUT);

}

pinMode(PIN\_BUTTON0, INPUT);

digitalWrite(PIN\_BUTTON0, HIGH); // enable internal pullup resistor

}

/\*

\* Flash the LEDs specified in the array in sequence, pausing at the

\* end only if Button0 (active-low) is pressed.

\*/

void loop()

{

byte i;

for (i=0; i < numberOfLeds; i++) {

digitalWrite(ledPins[i], HIGH);

delay(DELAYTIME\_BETWEEN);

digitalWrite(ledPins[i], LOW);

delay(DELAYTIME\_BETWEEN);

}

while (digitalRead(PIN\_BUTTON0) == LOW)

;

delay(DELAYTIME\_AFTER);

}

### Appendix E – How Arduino programs are structured

The most basic, standard form of a C (or C++) program is comprised of a main(), which is the starting point for execution. In the Arduino environment, there are some initialization functions that have to be called before your program begins. Because of this, the development environment provides the main() function itself and forces it to call this initialization before your code gets a chance to run.

Because most embedded programs tend to naturally fall into a pattern of one-time initialization code, and code that repeats forever within a loop, the Arduino team has formalized this structure in their environment. Sketches mostly contain two functions, setup() and loop(), with setup() being the function in which you perform your initializations, and loop() being the function called over-and-over again.

Basically, the entire C program that ends up being passed to the compiler looks something like this:

<the contents of your sketch is here, followed by...>

void \_init()

{

// the Arduino-environment initialization code...

}

void main()

{

\_init();

setup();

while (1)

loop();

}

Just in case you were wondering where that main() function went, that's the story!

### Appendix F – Arduino power budgeting (and how not to let out the magic smoke)

There are a number of key points to remember about managing and budgeting your power use on an Arduino. All of these limits must be respected when you start to design a project that connects components to pins on an Arduino. Failure to do so could potentially destroy the MCU, or the transistors just inside any individual pin on the MCU.

1. The microcontroller, running full-blast, may itself consume up to about 20mA.

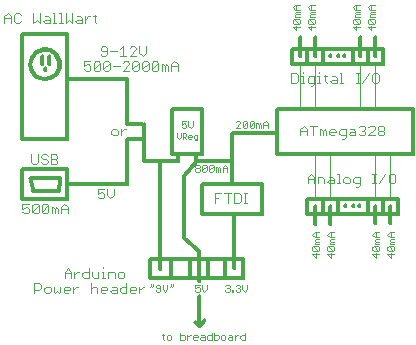
2. The absolute maximum current that can be sourced or sunk *at the same time through all I/O pins at the same time* through the entire chip is 200mA, including the chip's own power consumption mentioned above.

3. The *absolute maximum* current that should be *sourced* (supplied) by any one pin at any time is 40mA. When given a choice, you should attempt to limit current consumption to 20-25mA. Similarly, the same values hold true for the amount of current that can be *sunk*  (absorbed) by a pin at ground (GND, 0V)[[14]](#footnote-15).

4. If you power your Arduino from a USB connection, the maximum amount of current that you can draw from the PC is typically 500mA. However because there's a 200mA limit through the MCU itself, you may wonder how you could possibly exceed that 500mA limit. If you power any devices directly from the 5V POWER connector, this current comes out of that 500mA budget. An example of devices that you might try to power this way might be an RC servo, toy DC motor, or any external chips, power transistors, relays, etc. that you may be using on a breadboard.

5. If you power your Arduino from the barrel jack, you are limited to 800mA. 800mA is the maximum, assuming that your power source can even supply 800mA@5V. For example, it is unlikely that you can get 800mA@5V from a classic rectangular 9V “transistor radio” battery.

Figure F1. Power considerations for the Arduino board.



Question 1: How many pins does the Atmega328P MCU used on the standard Arduino have? Over what range of voltages will it operate?

Question 2: Assume that you have an LED connected to each of the 14 digital-only I/O pins on the Arduino. If all of the LEDs could possibly be on at the same time, what must the current be limited to through each of the LEDs?

Question 3: Assume that a project requires that a high-brightness LED be on any time that the Arduino is powered-on, and that this LED requires 350mA. What is the best way to supply power/current to this LED?

Depending on which limit(s) you violate, you may simply trip the self-resetting fuse on the Arduino board and not permanently damage anything. However, sometimes you're just not that lucky, and unfortunately it's not always immediately obvious which outcome has occurred. Sometimes your eyes or nose may give you some clear feedback if you’ve fried something.

The indication that you've (hopefully *only*) tripped the fuse is that the board will power itself off-then-on again. When this happens and you're using USB power, your PC will sound the falling-tone sequence, then the rising-tone sequence signifying that you've disconnected, then connected a USB device. This is your Arduino losing power and coming back up again. Unless you're reasonably certain how this occurred (you know where you accidentally shorted something), you should avoid repeating whatever you did to cause the fuse to reset.

### Appendix G – Limitations regarding the use of *Serial*

One thing that you need to know about the serial connection is that it communicates at a relatively slow rate, compared to the speed of the central processing unit (CPU) on the chip. The problem is that sending data over the serial link can slow down your programs wherever calls to Serial.print() are present. In situations where you cannot afford to spend multiple milliseconds waiting for data to get sent to the PC, you need to avoid using *Serial* print functions. To put some numbers on it, it takes approximately one millisecond per byte (character) to send to or receive from the PC.

***Question 4: Can you think of a way to use the oscilloscope to measure the time it takes to print out the message in the sketch you are currently running? Possible hint: Digital pin 1 (TX, a.k.a. transmit) is the pin over which serial data is sent to the PC.***

### Appendix H – Other sources for Arduino information and hardware

Places to learn more about the Arduino software environment:

* todbot's Bionic Arduino, and Spooky Arduino tutorials (http://todbot.com/blog/spookyarduino/)
* Adafruit tutorials (<http://www.ladyada.net/learn/arduino/index.html>)
* Arduino.cc Learning (<http://arduino.cc/en/Tutorial/HomePage>)
* Arduino Libraries (<http://arduino.cc/en/Reference/Libraries>)

Books: Getting Started with Arduino, Making Things Talk, C Programming with Microcontrollers

A few of the better places to buy Arduino-related hardware:

NKC Electronics (Florida) <http://www.nkcelectronics.com/>

SparkFun (Colorado) <http://www.sparkfun.com/>

Adafruit (NYC) <http://www.adafruit.com/>

SeeedStudio (China) <http://www.seeedstudio.com/depot/>

SureElectronics (China) <http://stores.ebay.com/sureelectronics>

IteadStudio (China) <http://iteadstudio.com/store/>

A few (and there are many more) popular places to buy electronic components for projects:

Digi-Key (USA/Canada) <http://www.digikey.com/>

Mouser (USA/Canada) <http://www.mouser.com/>

Future (USA/Canada) <http://www.futureelectronics.com/en/Pages/index.aspx>

Jameco (Belmont) <http://www.jameco.com/>

HSC (Santa Clara) <http://www.halted.com/>

Futurlec (Thailand) <http://www.futurlec.com/>

From the ME 106 website, see also Component Information (<http://www.engr.sjsu.edu/bjfurman/courses/ME106/componentinfo.htm>)

and Design Resources ([http://www.engr.sjsu.edu/bjfurman/courses/ME106/othermechatronicslinks.htm#Design%20resources](http://www.engr.sjsu.edu/bjfurman/courses/ME106/othermechatronicslinks.htm%23Design%20resources))

1. The process of sending code from the computer it was written on to the target microcontroller is usually referred to as ‘downloading’. The folks who developed the Arduino programming environment refer to this process as ‘uploading’. [↑](#footnote-ref-2)
2. [Really Bare Bones Board from ModernDevice](http://shop.moderndevice.com/products/rbbb-kit) [↑](#footnote-ref-3)
3. [Arduino Skeleton](http://geeker.com/electronics/arduino-skeleton-–-look-mom-no-pcb/), [Carboarduino](http://geeker.com/electronics/arduino-skeleton-–-look-mom-no-pcb/) [↑](#footnote-ref-4)
4. The software for Windows is distributed as a ZIP file, so no desktop icon or ***Start*** menu entry is automatically created. Because there are many different ways that you can remedy this according to your own particular tastes, we will leave it up to you to do this on your own personal machine. In the lab, we have placed a launch icon for the Arduino IDE under the ***Start*** menu. [↑](#footnote-ref-5)
5. This process is fully described in <http://arduino.cc/en/Guide/Environment#uploading> for all operating systems. [↑](#footnote-ref-6)
6. It used to be the case with old computer monitors that if you tried to use an inappropriate (or buggy) driver, or if they were set to an invalid resolution, that you could destroy the monitor because the electron gun could overrun limits internal to the monitor that did not have any other safeguards. Hence, you could literally torch a monitor with just software. Oops! [↑](#footnote-ref-7)
7. By the time that you read this, it may still be the case that we haven't gotten the 1.0 release of the Arduino software installed on the lab PCs. One difference pre-1.0 is that the filename extension for Arduino sketches is .pde, and 1.0 and later it's .ino. Just be prepared to see both extensions used or referred to this semester. [↑](#footnote-ref-8)
8. In general we do not recommend working directly off of a USB flash drive that you have brought with you, although this is up to you. The issue is that it can be slow working off of a flash drive, and the lab PCs are pathetically slow to begin with. At the end of the lab, you can save your sketch folders, or zip (we use 7-Zip on the lab computers) them up and email them to yourselves. Because the lab PCs can be wiped clean at any time, it is a bad idea to assume that you can come back another time to retrieve your files. [↑](#footnote-ref-9)
9. Serial.print() and Serial.println() [↑](#footnote-ref-10)
10. The constant 4883 is derived from the voltage range (5 V) being chopped-up into 1024 ranges by the ADC, or   
    4883 micro volts/division. 10000 represents 10000 millivolts/degree (10 mV/degree), the output sensitivity of the temperature sensor. Therefore, TC = ((n divisions)(4883 mV/division))/(10000 mV/degree). If you stick to math using integers rather than floating point values, the code both runs substantially faster and is considerably smaller on an 8-bit processor. [↑](#footnote-ref-11)
11. PWM is used for two reasons. Some components run more efficiently at full-power and zero-power intermittently, than continuously at some fractional power level. The other is that many microcontrollers cannot directly generate a voltage between fully-on or fully-off. In applicatons where a real analog output *is* required, the pulsed output can be smoothed to a fairly steady analog value with a capacitor, inductor or other type of filter circuit, which you'll learn about in lecture. [↑](#footnote-ref-12)
12. It is all too common for a project team to fail to get something working for days or weeks because they did not bother to read the paragraph or two about the proper use of a function or object in the documentation. [↑](#footnote-ref-13)
13. The argument for naming constants is similar to that for giving variables useful names, like *rotationCount* and *stepNum*, rather than *i* or *j.* The only exception is when those variables are used for nothing other than as an index inside of a *for* loop. If the variable has any meaning *at all*, please honor it by giving it a real name. [↑](#footnote-ref-14)
14. It's not always the case that the source and sink capabilities of a pin are the same, but in this case they are. [↑](#footnote-ref-15)