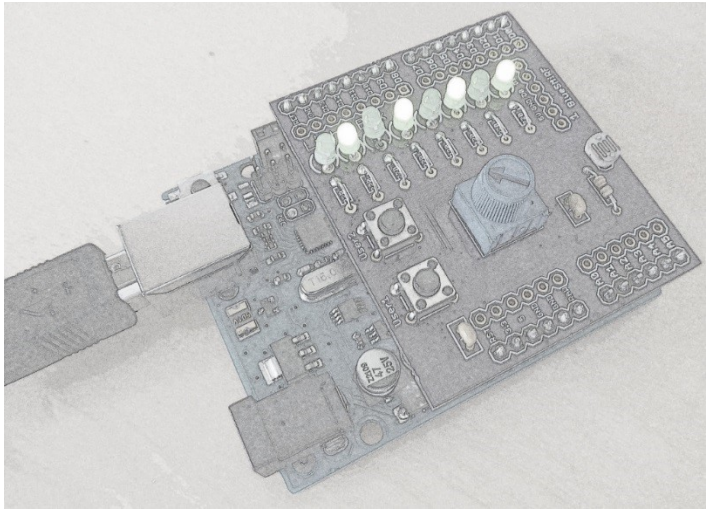
Learning C Programming

with an Arduino and a LightBox

Kevin D'Angelo

Fall 2019



Introduction

This document is a guide to learning the Arduino C programming language through programming exercises that generate patterns on the 8 LightBox LEDs in response to inputs from two push buttons, a potentiometer, and a photo cell all mounted on the LightBox.

The hardware for the accompanying exercises consists of an Arduino Uno board and a LightBox shield. The Arduino is a microcontroller-based platform that runs the C/C++ programming language. A computer (MAC, Windows or Linux) with a standard A to B connector USB cable (the one with the old fat connector) is also required.

Contents

1. Installing the Arduino Integrated Development Environment

The Integrated Development Environment (IDE) is the software used to edit the Arduino code, also known as a sketch. The IDE compiles the code, uploads it and burns it into the Atmel (now Microchip) ATMega microcontroller on the Arduino Uno board. The ATMega contains non-volatile (flash) memory embedded in its circuitry which is where your sketch goes.

The IDE is located here:

http://arduino.cc/en/Main/Software

Excellent instructions for getting started can be found here:

http://arduino.cc/en/Guide/HomePage

Download and install the IDE on your computer by selecting Windows or Mac OS X from the Homepage. Remember you have an Arduino Uno.

For older Windows, notice that the drivers may require manual installation (step 4).

For Macs, you might get a message such as “A new network interface has been detected”. Just cancel that, your Mac thinks the Uno is something else.

Once installed, Mac or Windows, the programming language is identical. An excellent reference is found on the website with a convenient tab along the top of the page. Refer to it regularly as you create your programs and you will develop a better understanding of the language.

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

There is also a web-based editor, where you can develop your code and store it in your Arduino account, obviating the need to install the IDE on your computer. It can be found under SOFTWARE on the Arduino home page.

2. Downloading an Arduino Example Sketch

Continue following the instructions and download the Blink example:

File -> Examples -> 01.Basics -> Blink

The sketch will load into the IDE (and expose you to some code).

Next, upload the sketch by clicking on the icon of a right-pointing arrow located in the upper left corner. The sketch will compile and then hopefully upload. If it doesn't, return to the Serial Port menu and double-check that you have selected the USB port with an Uno attached.

An LED on the Uno board will blink on and off at about 30 Hz when successful.

3. Understanding Program Flow

Here we will examine a few features of the programming language by referring to an older version of the Blink example, with modified comments, shown here.

```
////////////////////////////////////////////////////////////////
// Blink.ino
// Kevin D'Angelo
//
// Turns on an LED on for one second, then off for one second,
// repeatedly, ad nauseum.
//
// 10/21/2015
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
// Global Variables
////////////////////////////////////////////////////////////////
int LEDpin = 13;

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

```
//////////////////////////////////////////////////////////////////
// loop()
//////////////////////////////////////////////////////////////////
void loop() {
 digitalWrite(led, HIGH); // turn the LED on (Voltage = HIGH)
 delay(1000);             // wait for a second
 digitalWrite(led, LOW);  // turn the LED off (Voltage = LOW)
 delay(1000);             // wait for a second
}
```

Let's start with comments. Comments are important in documenting your code so you or someone else can understand what is going on when revisiting and/or debugging the program. Comments appear gray in color. They are defined by a // (double slash) in front of the single line comment, or bounded by /* *comments here* */ (slash star star slash) with the single or multi line comment in between. A header is a useful group of comments at the top of your code. It should include the name of the sketch (Blink.ino), your name, a description and a date.

In general, instructions are executed in the order they are given in the code. So the first instruction executed here is

```
int LEDpin = 13;
```

This instruction defines and initializes a variable (more on that later).

The next line of code is

```
void setup() {
```

This is a special subroutine whose contents are executed once at power up, or when the reset button is pushed. Every instruction between the open and close curly brackets belongs to this routine. In this case there is only one instruction:

```
pinMode(LEDpin, OUTPUT);
```

This instruction tells the micro how to configure port 13 which is connected to the LED on the Uno board. (n.b. port and pin are used interchangeably)

The next routine is the loop routine.

```
void loop() {
```

The instructions inside the loop routine curly brackets are executed in the order they appear and repeat over and over until power is removed, or the reset button is pushed.

All sketches require a `setup()` and a `loop()` routine.

Inside the loop routine, we have the `digitalWrite` instruction telling port 13 to drive its output HIGH (5 volts), to wait for 1,000 milliseconds (that would be 1 second), to drive its output LOW (0 volts), to wait for another second and then, like Sisyphus, to repeat this task over and over by returning to the instruction immediately following the open curly bracket at the top of loop, and so on.

4. A word on syntax

In the C programming language and for that matter, most programming languages, syntax is important. Things that matter:

Case – C is case sensitive, e.g. `Pinmode` is meaningless; `pinMode` is an instruction. The Arduino IDE will help by changing the color of the instruction.

The **semicolon** – all instructions end with a semicolon, i.e. one of these ; Conditional statements whose contents begin with a { do not require a ; on the conditional statement line.

Curly brackets – subroutines and conditional constructs encase their associated instructions in open and close curly brackets, i.e. between { and }.

Indentation – although not required, indentation makes it easier to keep track of your embedded conditional constructs' curly brackets. The IDE makes it easy.

If you start an open { followed by a return, the next line will automatically be indented. Likewise, a close } will align itself with its corresponding {. Be sure you have the exact same number of {'s and }'s. You will screw this up, and the compiler will indicate as much, albeit cryptically, and you will find yourself in curly bracket hell trying to find the mismatch. The IDE has an **Auto Format** tool to help with this. Auto Format will clean up indentation throughout the code. You can find it by right-clicking inside the edit space of the IDE.

5. Understanding Variables

Variables hold values used in and modified by your programming. They can be embedded in mathematical operations or in conditional constructs, or simply hold a number in a more intuitive form. Often, you will refresh a variable value based on sensory input. One of the hardest things about using variables is coming up with a decent name for a given variable. Once you name a variable, that name gets used throughout your code. No pressure.

Another benefit of using variables is having the ability to globally change your program. If you reference the same variable in multiple places in your code and you wish to change its value, simply change the value in the variable declaration. To save limited RAM (Random Access Memory) memory space on variables your program does not change, you can use the **const** keyword to define, say, an integer constant that is used in your program. The compiler will store the constant value in fixed program space rather than taking up precious RAM space.

For example, a **variable declaration**:

```
int led = 13;
```

Precede this instruction with the const keyword to make it a **constant**:

```
const int led = 13;
```

Here we name the variable or constant, led, define it as an int (short for integer), and give it a value of 13. An `int` is a 16-bit value, so pretty big (from -32,768 to 32,767 which is -2^{15} to $2^{15}-1$, the 16th bit being the sign bit). The size is defined by the architecture of the ATMega which is a 16-bit machine (i.e. mathematical operations and memory IO is done 16 bits at a time). Another Arduino, the Due (get it?) is available with a 32-bit architecture micro which opens up the platform to fast big number applications.

Other useful datatypes (`int` is a datatype) are

long – this is the Uno's 32-bit integer variable. On the ATMega, 32-bit operations take longer to execute than 16-bit operations.

float – the 32-bit variable for non-integer numbers.

Variables are assigned values by instructions in your code that can appear a little odd at first. For instance,

```
i = i + 1;
```

In this instruction, the variable appears on both sides of the equation, but it's not really an equation, it's an assignment (otherwise it would reduce to 0=1). This instruction increments the variable i by one. The C language has special instructions for common instructions like increment. In this case,

```
i++;
```

accomplishes the same thing as `i = i + 1;`

You can get complicated with variable assignments too. For example

```
SineTable[n] = Tpwm/2*sin(2.0*pi*n/64)+T/2;
```

This instruction uses several variables: `SineTable, n, Tpwm, pi, and T`. It is used in a "`for`" loop to populate a lookup table, or array, of sinusoidal values.

Global variables can be used throughout your program. ***They are declared at the top of your program outside of any curly brackets.*** This gives them a prominent, easy to find location. Variables may be assigned a value in the declaration. This value can serve as an initialization value for the variable, or simply serve as a constant, as is the case in general for port variables (same result as using `const`). For some variables, you may not care about initializing and they will default to 0. Your program can begin using these variables and manipulating their values.

Arrays are convenient structures that store multiple variables in, well, an array. They are declared anywhere a variable is declared and are treated the same way. In section 7 we will talk about the 8 ports for the 8 LightBox LEDs. An array is a good way to manage initialization and control of the LEDs. In this case, we can declare an array and initialize it by loading it with the port values for the LEDs.

```
int LED[8] = {4,5,6,7,8,9,10,11};
```

Now we have 8 variables, where `LED[0]` has the value 4, `LED[1]` has the value 5, etc., and `LED[7]` has the value 11. Notice how the indexing of the array starts at 0. Be very careful not to assign a value to `LED[8]` because we have not reserved space for `LED[8]`. The compiler will not object, but there will generally be something assigned to that memory location and you will unknowingly overwrite what WAS there, and invariably screw something up. It is a confounding problem to hunt down.

6. Understanding Control Structures

Control structures do the decision making in your code. Control statements come in many forms and the same outcome can often be realized with different control statements. Common control statements are “if”, “if else”, “for”, and “while”. An “if” statement will conditionally execute instructions if the logical expression inside the parenthesis is true. An “if else” statement will execute certain instructions if true, and other instructions if false. A “while” statement will execute instructions as long as the logical expression inside the parenthesis remains true.

For example:

```
if (someVariable > someNumber) {
     // execute the instructions between the curly brackets
}
```

In this example the greater than (>) operator is used. Other **conditional** operators are

== is equal to (do not confuse this with a single “=” assignment operator)
!= is not equal to
< is less than
> is greater than
<= is less than or equal to
>= is greater than or equal to

In C as well as in many other languages, ! (called bang), is the “not” operator. “!” negates its operand. e.g. if my_var=1, then !my_var has a value of 0.

The if statement can get you a lot of mileage, but **if** you need to repeat something, **then** you’re covered by using “for” or “while”, **else** use the if. Huh?

Here's a summary

Control Statements

if then else

```
int a = 89;
int b = 42;

if(a > b) {
 Serial.print("a is greater than b");
}
else {
 Serial.print("a is less than b");
}
```

for loop

```
for(int i=1; i <= 10; i++) {   // i is incremented from 1 to 10
 Serial.println(i);            // prints 1 to 10
 delay(500);                   // halt execution for 500ms
}
```

while loop

```
int i = 0;
while(i < 10) {
 i++;                 // equivalent to i=i+1;
 Serial.println(i);            // prints 1 to 10
 delay(500);
}
```

Boolean operators

The Boolean operators, and, or, and not, can be used to combine logical conditionals within control statements.

```
&& (and)
|| (or)
! (not)
```

For instance, if you need “a” to be greater than 50, and “b” to be less than 50, you can combine those two conditionals like this:

```
if((a > 50) && (b < 50)) {
  Serial.print("Values in normal range");
}
```

(Serial.print is explained in section 9)

Boolean operators are also useful if you need to check for something you’ve seen in mathematics, such as a range, e.g. 10 < a < 50, as in, is “a” between 10 and 50. Here you split the range into the two end points and check to see if both are true.

```
if((a > 10) && (a < 50)) {
  Serial.print("a in normal range");
}
```

Note that you do not have to condition on only one logical expression; you can combine as many logical expressions as you like using these Boolean operators.

A quick Boolean review of two common TRUTH tables:

AND (&&, only TRUE if both operands are TRUE. It’s like multiplication)

TRUE && TRUE = TRUE
TRUE && FALSE = FALSE
FALSE && TRUE = FALSE
FALSE && FALSE = FALSE

OR (||, TRUE if either operand is TRUE, it’s like addition)

TRUE || TRUE = TRUE
TRUE || FALSE = TRUE
FALSE || TRUE = TRUE
FALSE || FALSE = FALSE

7. LightBox Hardware

The LightBox is a custom PCB (printed circuit board) designed for this class. It attaches to an Arduino Uno board, which makes it a "shield." Many shields are available commercially to perform myriad functions, but most simple shields for learning involve a fragile breadboard for setting up circuits. The wires come out, and it can be time consuming to understand whether the problem is the circuit or the software. It's an age-old argument between hardware and software developers. If something is not working here, unless you botched the soldering, chances are it's not the LightBox. Do not fear, you will get the chance to experience hardware failures when you start making your own stuff.

Before we start programming, we need to spend some time going over a few basic electrical concepts in order to get up and running with the LightBox. You will see that writing Arduino code is not only about software. Rather it's more about firmware, which is in-between hard-ware and soft-ware, and involves a lot of both. Anytime you develop a program for a microcontroller like this, in a system surrounded by circuits, you are writing *firmware* for an *embedded system*.

An *embedded system* is a computer system with dedicated hardware and software that performs a specific function.

The LightBox circuitry consists of a potentiometer, a photo cell, two pushbuttons, and 8 LEDs along with a few discrete resistors and capacitors. There are extra sets of holes for adding or modifying circuits. There is also a set of holes marked BlueSMiRF. This is a connector for a dedicated Bluetooth module available from SparkFun, not covered here.

To program the Arduino for controlling the LightBox, it is necessary to know where the LightBox ports are connected on the Arduino.

Port Assignment Table

Circuit Component	Arduino Port
LED0	D4
LED1	D5*
LED2	D6*
LED3	D7
LED4	D8
LED5	D9*
LED6	D10*
LED7	D11*
Potentiometer Center Tap	A0
Photo cell	A1
“USER1” pushbutton	D2
“USER2” pushbutton	D3

* PWM Ports

Ports are the interface between the microcontroller and circuit components on the LightBox. They are assigned to circuit components with physical copper traces on the PCB as defined in the above table.

The D in the port names refers to the digital ports located on one side of the Arduino board. The digital ports can be configured as an OUTPUT or an INPUT and work with logic levels. OUTPUT ports present a logic HIGH as 5v on the port or a logic LOW as 0v on the port. From the Arduino Reference Guide, INPUT ports read a HIGH level if the voltage on the pin is greater than 3v, or a LOW level if the voltage is less and 3v. But I’m not buying it.

To be more exact with logic levels, we must consult the ATMega datasheet and learn that the input high voltage, $V_{IH\ MIN}$, is 0.6Vcc. Since Vcc=5v, V_{IH}=3v, so a logic HIGH must be greater than 3v. On the other hand, the input low voltage $V_{IL\ MAX}$ is 0.3Vcc, or 1.5v, so a logic LOW must be less than 1.5v. Less than 3v is not good enough.

The gap between the logic levels acts as a noise filter. Without it, a slight perturbation of 3v at the input would result in a chance of either a HIGH or a LOW rather than a guaranteed HIGH or LOW. The analog world creeps in.

Table 30-1. Common DC characteristics T_A = -40°C to 105°C, V_{CC} = 1.8V to 5.5V (unless otherwise noted)

Symbol	Parameter	Condition	Min.	Typ.	Max.	Units
V_{IL}	Input Low Voltage, except XTAL1 and $\overline{\text{RESET}}$ pin	V_{CC} = 1.8V - 2.4V V_{CC} = 2.4V - 5.5V	-0.5 -0.5		$0.2V_{CC}$ (1) $0.3V_{CC}$ (1)	V
V_{IH}	Input High Voltage, except XTAL1 and $\overline{\text{RESET}}$ pins	V_{CC} = 1.8V - 2.4V V_{CC} = 2.4V - 5.5V	$0.7V_{CC}$ (2) $0.6V_{CC}$ (2)		V_{CC} + 0.5 V_{CC} + 0.5	V

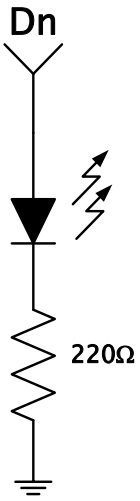
Back to ports, the A in the port assignment table refers to the analog ports which are located on the opposite side of the board from the digital ones. These ports read analog values from 0 volts to 5 volts. The voltage is converted **linearly** to a digital value between 0 (0v) and 1023 (5v) by an internal Analog to Digital converter.

In the engineering world you come across the term “linear.” It signifies exactly that, a line, as in $y=mx+b$. With the analog ports, think of the x-axis as the analog input voltage, and the y-axis as the output digital representation, i.e. “digital representation” = 1023/5 * “analog voltage” + 0.

Back to ports again, regardless of whether the port is a D or an A, in most cases it is referred to only by its number in the programming instructions. The compiler determines the context according to the instruction with the port number unless you use an A port as a digital port, in which case you need to include the “A” (e.g. `digitalWrite(A1, LOW)`).

8. LEDs

The LEDs (Light Emitting Diodes) are connected between a given port Dn and a current setting resistor which in turn is connected to ground.



LED Circuit Diagram

In an LED, current flows from the anode (top of the LED) to the cathode (bottom of the LED). My pneumonic is that current flows in alphabetical direction, from A (anode) to C (cathode). **The physically longer lead on a discrete LED is the anode.** With the above configuration, a HIGH level on the port turns the LED ON, and a LOW level on the port turns the LED OFF. If you define a variable, LED0, to be equal to 4 (from the port assignment table).

```
int LED0 = 4;
```

then, you can instruct LED0 to be turned ON and OFF with

```
digitalWrite (LED0, HIGH);     // ON
digitalWrite (LED0, LOW);      // OFF
```

9. Initialization Routine

The initialization routine begins with the statement

```
void setup() {
```

and it ends with

```
}
```

The code in the setup() routine is executed upon power up, or when the built-in RESET button is pressed. Here you group instructions that initialize things, such as

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

`Serial.begin` initializes the serial port for communication between the Arduino and the computer; a very important debugging tool. It is initialized to 9600 baud (bits per second) – a good number.

Along with `Serial.begin` is the instruction to send data to the serial monitor (a pop up window that appears when you select Tools->Serial Monitor or Ctrl+Shift+M, or by clicking the box in the upper right corner of the IDE).

```
Serial.print (variable_name);
```

`Serial.print` can be anywhere in your code including in the `setup()` routine.

Also in your `setup()` routine will be initialization for how you will be using the ATMega ports. Configuring the ports as outputs to drive LEDs is accomplished with the `pinMode` instruction:

```
pinMode (LED0, OUTPUT);
pinMode (LED1, OUTPUT);
```

etc. (for the other 6 LEDs).

Alternatively, we can use our ***array*** with a "for" loop to accomplish the same:

```
for (i=0; i<8; i++) pinMode (LED[i], OUTPUT);
```

Notice how since there's only one instruction in the "for" loop, I didn't need to use curly brackets.

Exercise 1 – reset sequence

As a first exercise, in the setup() routine, write a sequence of instructions that make your 8 LEDs blink in a fetching sequence. ***Describe your sequence in detail using a comment box at the top of your code***. Aim for simple and quick because this sequence will signify that your setup() routine is executing, and will play every time you power up or press the reset button. For this exercise, you will need to use the delay() function to space out the turning on and off of your LEDs. The delay() function takes one argument (the number within parenthesis) which specifies the number of milliseconds (1 millisecond = 0.001 seconds or 10^{-3} seconds) for your program to sit and do nothing. A 1 second delay looks like this

```
delay(1000);
```

Experiment with how 1ms feels compared to 10ms or 100ms or 1 second (hint, make your delay time a variable so you can change it easily). The Blink example uses delay().

Put your variable declarations at the beginning of your sketch, and follow with the setup() and loop() routines. (remember, even if your loop() routine is empty, the compiler still requires it.)

10. Subroutines

The subroutine is an important tool to use in making your program readable. Without using subroutines, you would have a difficult time telling the forest from the trees when analyzing and debugging your code. Without them, you would be hard pressed to get a decent grade in this course.

Subroutines are a way to have your program go off and do a bunch of stuff with a single instruction.

```
void setup() {
```

is an example of a subroutine, albeit one that is not explicitly called, rather it is automatically executed at power up or when the reset button is pushed, and is required in every Arduino sketch. Some subroutines can return values. For subroutines that do not return values, their declaration has the keyword "void" in them.

Here's an example

From within the `setup()` routine, call your subroutine

```
InitializationBlinky ();
```

Then, prior to your `setup()` and `loop()` routines, and after your variable declarations, create the subroutine:

```
void InitializationBlinky () {
     // code that does your initialization thing goes here
}
```

Sometimes we'll want to pass a parameter to a subroutine. No worries, simply insert the parameter inside the parenthesis. e.g.

```
InitializationBlinky(100);     // subroutine call
```

the subroutine looks like this:

```
void InitializationBlinky(int init_delay_time) {
  // code here can use init_delay_time as argument for delay()
}
```

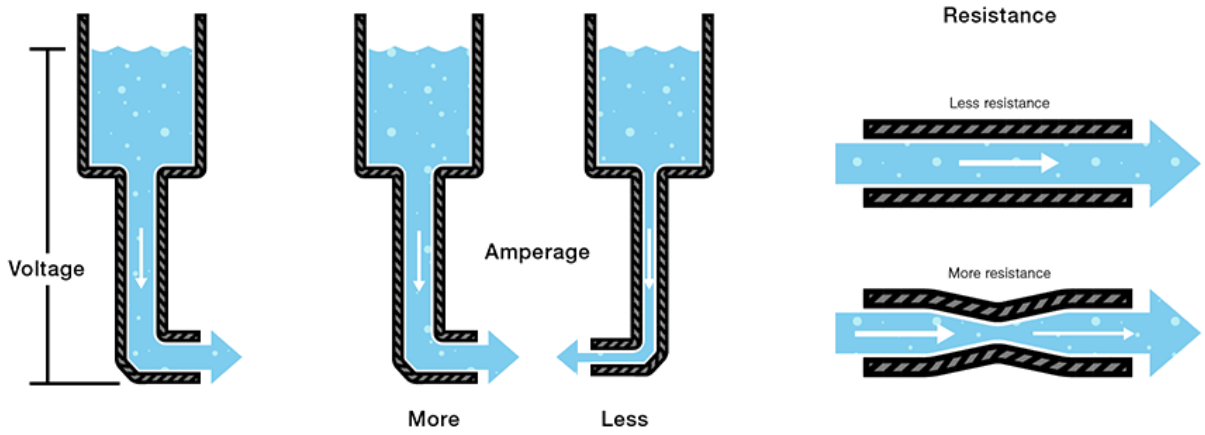
In the subroutine declaration, the variable init_delay_time is declared as an `int`. It is a "local" variable, only to be used inside the subroutine. Make sure it gets a unique name or you will end up with a conflict with a global variable. Bad.

Exercise 2 – subroutines

Now take the initialization code in your `setup()` routine, and create a subroutine for it. Pass at least one parameter to your subroutine. Call the subroutine within `setup()`.

11. Ohm's Law

Now for a few basic electrical principles you saw or will see in your Physics course. First, let's think about voltage and current in terms of something visible like water storage and flow by referring to this handy graphic from our friends at sparkfun,



where voltage magnitude is analogous to the height of a column of water, and current magnitude (amperage in this graphic) is analogous to the amount of water flowing through the pipe. Resistance is inversely proportional to the cross-sectional area of the pipe.

Ohm's law is the relation between voltage, current, and resistance. It states that the voltage drop across a conductor is equal to the current through the conductor multiplied by the resistance of the conductor. In other words, voltage is proportional to current with resistance as the constant of proportionality. With V as voltage, I as current, and R as resistance, Ohm's law is

$$V = IR$$

or, with a simple algebraic rearrangement

$$I = \frac{V}{R}$$

To relate back to our water example, a higher water column, or a larger cross sectional area of the pipe will result in higher current.

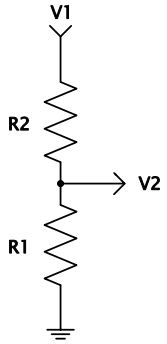
Both of these forms are frequently used, and in electrical engineering, Ohm's law is used ALL THE TIME.

12. Kirchhoff's Current Law

Here's a useful tool in analyzing and understanding circuits with resistors in them. It states, roughly, that any current flowing into a node has to also flow out of the node. So if you have two resistors in series, with nothing else attached, the resistors have the same current. (There is also a voltage law. It is mostly academic.)

13. The Resistor Divider

The LightBox contains two resistor dividers, so let's use KCL (Kirchhoff's Current Law) to analyze the resistor divider and see what we get. We'll also use Ohm's law (always).



Resistor Divider

First, we'll calculate the current flowing through both resistors. The resistors have the same current as long as there is no current flowing out of our node V2. To state it another way, let's say that a load on V2 draws zero current.

Ohm's law in action:

$$I = \frac{V1}{R1 + R2}$$

Now that we have the current, we can calculate the voltage at V2. Since R1 has one end connected to GND (ground), we know the voltage drop across R1 is equal to V2 (GND is commonly equal to 0 volts). In terms of current and R1,

$$V2 = I * R1$$

Looks like V=IR, but we can make it more useful by eliminating I from the equation by substituting the first equation into the second to get

$$V2 = V1 * \frac{R1}{R1 + R2}$$

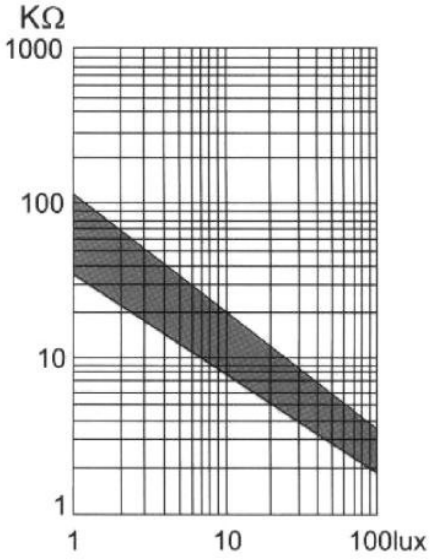
Voilà, the resistor divider. We'll use this common relationship for our potentiometer and for our photo cell.

14. Photo Cell as Input

A photo cell, also known as a photoconductive cell or a photoresistor, is a resistor with a resistance that varies by the amount of light it is exposed to. The photo cell we use here consists of a squiggly line of CdS (cadmium sulfide) encased in a clear plastic enclosure. Photocells are typically used as sensors of ambient light to provide feedback in a system that compensates by adjusting some other parameter such as screen brightness. My car has one in its rearview mirror as a means to reduce the reflectivity of the mirror at night when headlights are behind.

In no light (hard to find), the photo cell has a resistance of 1MΩ (1 mega ohm, or $1x10^6$ ohms). In direct light, such as when you shine your phone's LED at it, the resistance drops to about 1kΩ (1 kilo ohms, or $1x10^3$ ohms).

The datasheet provides a log-log plot of Resistance vs. Illuminance:



Resistance vs. Illuminance plotted on a log/log scale

The shaded area is the operating range of resistance vs. illuminance. This is a fairly crude device with a wide variation of 8kΩ to 20kΩ at 10Lux, but is good enough for many applications.

The photo cell in the LightBox is configured as a resistor divider with a fixed 10kΩ resistor. The 10kΩ resistor is on the bottom connected to ground, and the photo cell is the top resistor connected to 5v. The connection between the two is connected to analog input port 1, or A1.

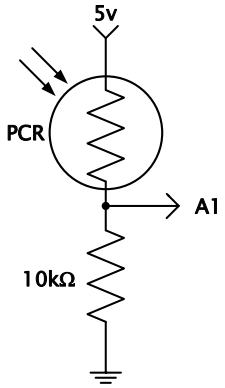
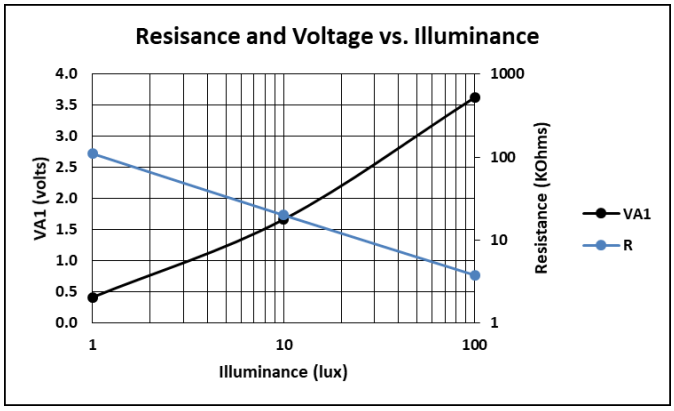


Photo Cell Circuit Diagram

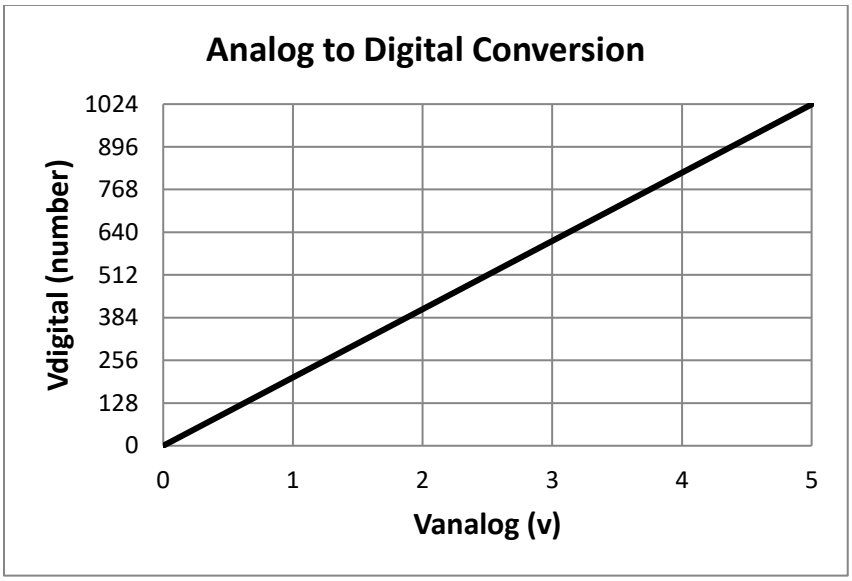
Using our resistor divider equation, for the voltage at A1, we have

$$VA1 = 5 * \frac{10\text{k}}{10\text{k} + \text{PCR}}$$

Here, notice that VA1 is inversely proportional to PCR, which is a non-linear relationship. Try plotting VA1 vs. PCR with a spreadsheet and see. The resistance vs. illuminance is not exactly linear either, in fact it is logarithmic, which is a common physical property of many devices. When you combine the log nature with the inversely proportional relation, you get a curve slightly bowed but somewhat linear with illuminance which is nice.



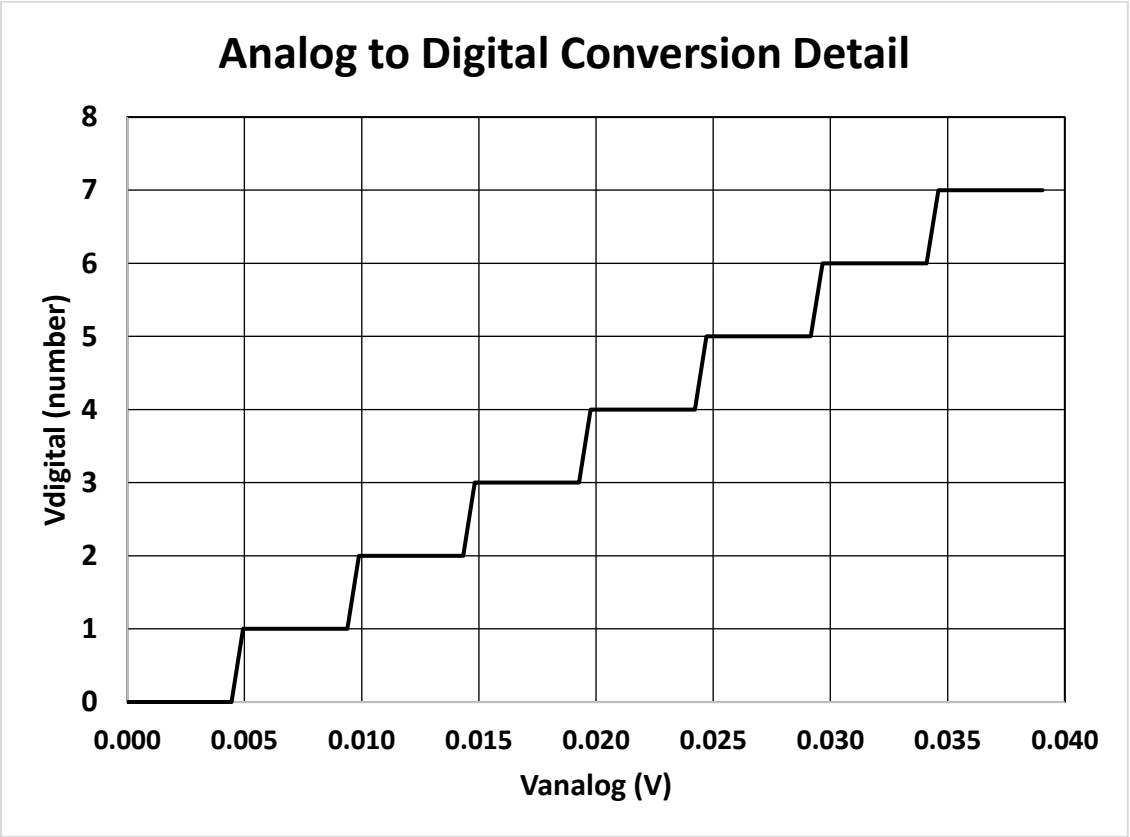
`analogRead()` is the instruction used to read an analog voltage on one of the analog ports and to convert it into a digital representation with the analog-to-digital converter on the ATmega chip. The digital representation goes from 0, which is equal to 0v, to 1023 which is equal to 5v. It is a linear relationship as seen in the following plot of digital number vs. analog voltage (the maximum "y" value is 1023).



The following code fragment can be used to read the voltage on A1 into a digital representation to be used as the input to a function:

```
int PCpin = 1;            // variable for port number
int Vs;                   // variable for sensed voltage
Vs = analogRead (PCpin); // assign port voltage to Vs
```

In reality, the analog to digital conversion is a discrete one and instead of having a smooth transfer function, it is a staircase with 1024 integer levels (including 0) of 1/1023 * 5, or about 4.9mv (millivolts) each. The converter truncates (lops off, does not round) any decimal to form an integer. Yes, it can be a little mind bending at first.



Exercise 3 - `analogRead()`

Continuously print the sensed voltage while you vary the light on the photo cell. Use a subroutine inside of `loop()`.

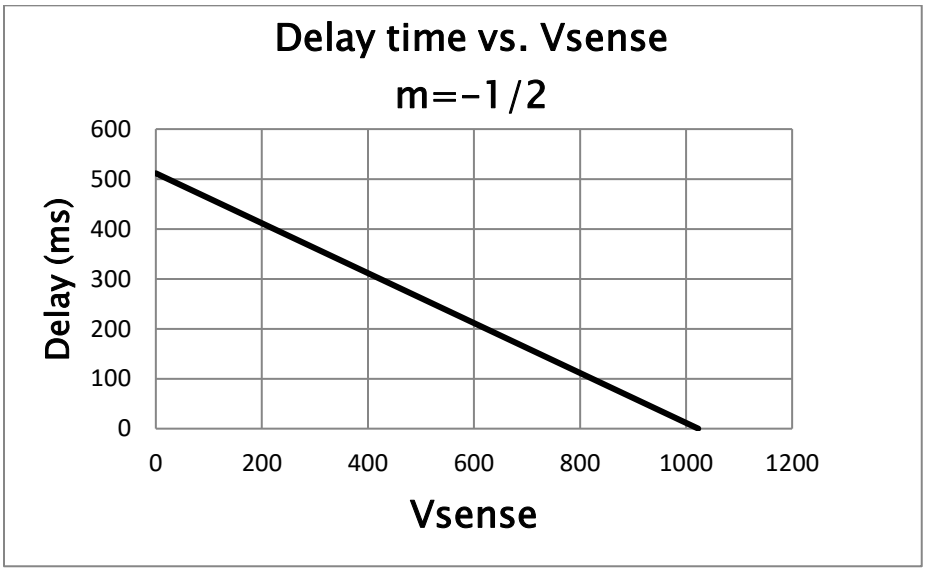
Hint: Use the `analogRead()` function along with `Serial.println()` and the serial monitor (Ctrl+Shift+M) to print out readings. If the numbers fly by too fast, insert a `delay()` after each `analogRead()`. Extrapolating from the voltage vs. sense curve, you should be able to get around 4.9v with a light source very close to the photo cell. Since you know that 5v = 1023, you can do a calculation to see if it agrees with what you are seeing. The calculation is a simple ratio, e.g. 4.9v/5v * 1023 = 1002. You should be in the ballpark of 1000+ for maximum illumination. At the other end of the scale, when the photo cell is in darkness, you should see around 120mv, or about 25. Use Algebra I skills to determine what voltage you are seeing at each end of the brightness spectrum.

Next, we will use the voltage generated by our photo cell as the input to a function that controls the LEDs.

Exercise 4 - `delay()`

Make the LEDs blink in sequence at a speed proportional to the amount of light hitting the photocell. Use a sequence of your choosing, such as blinking in sequence LED1 to LED8 then circling back to LED1 etc. Vary the slope of the curve (lines are referred to as curves), to find one you like.

Hint, use `delay()` to control the speed, but remember that the voltage goes up with illumination (is directly proportional), but the `delay()` time would need to go down. Think of a line with a negative slope. You may not want to use the full 1023 range with a non-unity slope (we call this non-unity gain). e.g.

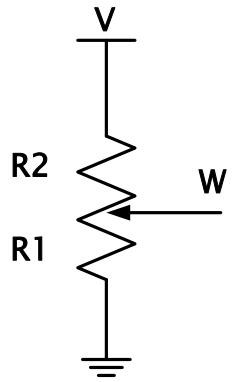


If you noticed that you have to wait a long time for the pattern to respond, try inserting an `analogRead()` before each `delay()` call to refresh the reading.

After you finish with that, for extra, unrelated to the delay function, create a light "thermometer" using the 8 LEDs (all LEDs on would be the brightest reading, none on the lowest). A personal favorite.

15. Potentiometer as Input

The Potentiometer (pot, colloquially) is used in many applications such as the controls for audio equipment, either logarithmic or linear, or as a joystick with a pot for each axis. We can program the LightBox potentiometer to perform any function. It is the rotary knob located right in the middle of the LightBox. A potentiometer consists of a long resistor split in two by a "wiper". If one end of the resistor is connected to 5v, as is the case here, and the other end is connected to GND, then a voltage between 5v and GND will be present on the wiper contact. When you turn the knob, you move the wiper up and down the resistor. Clockwise is up (higher in voltage), and counterclockwise is down (lower in voltage).



Potentiometer Circuit

The pot is a resistor divider with variable resistors, with the wiper voltage, Vw

$$Vw = V * \frac{R1}{R1 + R2}$$

Since R1 + R2 is constant (10kΩ for this particular pot), Vw is linear with R1 (i.e. R1 is your "x"). In this case, our pot has a linear taper, so it is also linear with the turning of the knob. Some pots have a log taper. There the response is logarithmic with the turning of the knob. Log taper pots are useful in things like volume control because human hearing has a logarithmic response.

The wiper terminal is connected to analog port 0. The digital representation of the voltage on it can be read into the variable *Vw* with the following code segment:

```
int Wpin = 0;             // variable for port number
int Vw;                   // variable for the wiper voltage
Vw = analogRead (Wpin);   // assign port voltage to Vw
```

Now that you have *Vw*, it can be used as the input to any function you can dream up.

Exercise 5 – potentiometer

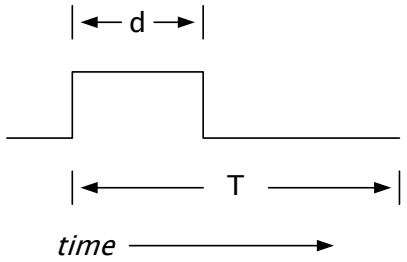
The potentiometer input is similar to the photo cell but gives a voltage with much more control. Why? Because it's linear (and it's a knob), and we like things that are linear. First, read and display the values of the pot as you did with the photo cell. To see the pot in action, take your photo cell subroutine and now instead of using the photo cell reading, use the potentiometer reading instead.

16. Pulse Width Modulation

PWM (Pulse Width Modulation) is used in many applications to control the amount of power delivered to a load. It can be used to vary the brightness of an LED or to vary the speed of a motor.

It is a repeating waveform with a defined ON time and period. The ratio of the ON time to the period is known as the duty cycle. The ON time is the percentage of available power delivered. For LEDs, a 50% duty cycle will give you 50% of the brightness of a constantly powered LED. For LED lighting, PWM is nice because brightness follows a linear relationship with duty cycle. By contrast, when controlling LED brightness with current, brightness goes in a logarithmic relationship, which is, you guessed it, the way your eye works too.

PWM Waveform:



In this drawing, the duty cycle is d/T.

Lucky for you, the Arduino programming language has a special instruction that generates this repeating waveform on certain ports. It is

```
analogWrite (pin, value); // PWM setting of "value" on "pin"
```

where value is the amount of ON time relative to the maximum always-on-value of 255. e.g. a value of 102 produces a duty cycle of 102/255 = 40%.

Exercise 6 – PWM

Read the potentiometer's voltage, and use it as the input to control the PWM duty cycle on the LEDs that have PWM available to them (see table in section 7 pg. 14). Make it a subroutine. Hint: the potentiometer voltage will be a value between 0 and 1023, while the PWM range is between 0 and 255. Scale (divide) the potentiometer voltage accordingly.

17. Binary Numbers

Familiarity with the binary number system is useful when working with microcontrollers. It helps explain why PWM only goes to 255 while the Pot voltage goes to 1023.

For starters, let's count from zero to three in binary

0: 00
1: 01
2: 10
3: 11

How many bits does it take to count to three? 2 bits. A 2 bit number (getting no respect here) has 2^2 values and goes from 0 to $2^2-1=3$. In the ATMega, the PWM register is an 8-bit register, so its maximum value is $2^8-1=255$. The internal analog-to-digital converter is 10 bits, so $2^{10}-1=1023$ which explains why the maximum values differ.

When reading a binary number into decimal, which is how normal humans think, know that each bit has a value of 2 raised to the power of the bit's position, starting with zero. We call the bit on the far right side the LSB (least significant bit), and the bit on the far left side the MSB (most significant bit). If a bit is set to 1, you count that position, if it's 0 you don't.

LSB: $2^0 = 1$
$2^1 = 2$
$2^2 = 4$
MSB: $2^3 = 8$
And so on. The binary number 0101 is converted to decimal by the formula

$0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 5$

We just showed that $0101_2 = 5_{10}$. In other words 0101 base 2 or binary is equivalent to 5 base ten or decimal.

Exercise 7 – Binary Conversion

Now for some fun with binary counting. Convert the digital representation of the potentiometer voltage into an 8-bit (a byte) binary number and display it using the 8 LEDs on the LightBox. Use a subroutine, and pass the decimal value to be converted to it as an argument (the example in the subroutines section).

Hint: You'll have to scale the potentiometer value (similar to the PWM exercise).

Extra: Incorporate hysteresis which is described in section 20a.

Binary Conversion

Start by understanding how to convert a decimal number to binary. Use the fact that the oddness or evenness of a number will tell you what the LSB will be. You can find the odd or evenness by dividing by 2 and seeing if there's a remainder. Let's look at 5 again, and convert it to a 4-bit binary number (a nibble, by the way).

5/2 = 2 plus a remainder, so set the LSB to 1: result is 1
Now use the result of that division (the integer 2) without the remainder to find the next significant bit
2/2 = 1 plus no remainder, so set the next significant bit to 0: result is 01
Same thing
1/2 = 0 plus a remainder, so set the next significant bit to 1: result is 101
And again because we have four bits to set (just saying)
0/2 = 0 plus no remainder, so set the next significant bit to 0: result is 0101

If you set the lower 4 LEDs in that order, you have your LED binary representation of 5 on the 8 LEDs:

OFF OFF OFF OFF OFF ON OFF ON

0 0 0 0 0 1 0 1

The programming language has a convenient numerical operator to facilitate these calculations. It is called the “modulo” operator and is represented by the percent sign, %. % returns the remainder of an integer division.

e.g. 5 % 2 = 1 (because 5 divided by 2 equals 2 with 1 left over, NOT what would come after the decimal point)

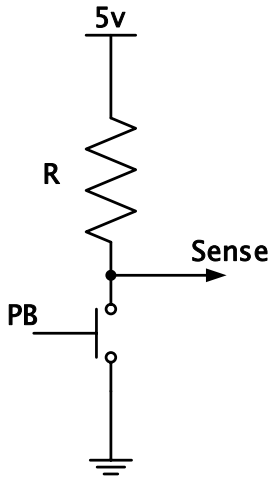
You can check to see if the remainder is 0 and not worry about the value.

Conveniently, with integer division, the remainder is truncated or dropped. So 5/2=2. (if you wanted that remainder, you’d have to use floating point numbers e.g. 5.0/2.0, then you’d get 2.5)

Hint: use % and / with an “if” statement for each LED to determine whether to set the LED HIGH, or LOW with digitalWrite.

18. The Push Button and the Interrupt

The Light Box has two pushbuttons, labeled User1 and User2. The pushbutton is a simple mechanical device that shorts its two terminals when the button is pressed. The Arduino code detects when the button is pressed by sensing a voltage change on one of the digital inputs. On the LightBox, the pushbuttons are configured like this:



Push Button Circuit

In this configuration, the voltage on Sense is normally high because it is resistively connected to 5v. When the button is pressed, the Sense node is shorted to GND and it becomes a logic low (note that R has 5v across it now). The resistor "pull-up" (R in the circuit) resides within the ATMega but is normally not connected. Connect it by configuring the port to have a pull-up by the following code fragment using pinMode in your `setup()` routine:

```
int User1_PB=2;
pinMode (User1_PB, INPUT_PULLUP);  //configure with pull up R
```

The internal pull up resistor has a value of 35kΩ. It has a large inherent variation, perhaps as much as +/- 40%, but that's OK since precision is not required.

Our code will detect if the button is pushed by employing something called an *interrupt*. In embedded systems the interrupt is the mechanism to communicate with real-world physical interfaces by detecting asynchronous events like a button push. The mobile phone is many things, including an embedded system. When you push a button and the phone comes to life out of hibernation, it is doing so with an interrupt triggered off of the button push. When the button is pushed, normal operation is suspended for an instant while the interrupt is processed by an ISR (interrupt service routine). Once the service routine completes, normal operation resumes where it left off. Interrupts generally take priority over whatever the micro is doing at a given time and within the micro there is even a pecking order for servicing different types of interrupts. They need to be short and quick to avoid fouling up another potentially time-sensitive operation. Lots of room for error if your interrupt service routine is a time hog.

That said, the Arduino language facilitates servicing the normal interface types of interrupts we will use, so we'll try and keep it simple and stay out of trouble. Our interrupt needs to detect when the button is pushed, which translates to detecting when the voltage on the input port goes from HIGH to LOW, i.e. presents a falling edge. When that occurs, we will direct the program to an ISR that will qualify the falling edge and set a variable to signal to the main routine that the event has occurred. We'll break it down.

The buttons, you recall, are named User1 and User2, so we'll try and use that nomenclature in our variables.

Here's what your ISR might look like:

```
void GetUser1 () {
     Button1Pushed = 1;        // set the variable
}
```

A one liner!

Great, so now we need to tell the main program how to use our ISR. Do it with these instructions:

```
volatile int Button1Pushed;   // our signaling variable
```

That's in the variable declaration section. In setup(), you need

```
interrupts ();                          // enables interrupts
attachInterrupt (0, GetUser1, FALLING); // we use interrupt 0,
                                        // ISR=GetUser1 and
                                        // on the FALLING edge
```

A few things to notice.

Number one is that we have a new kind of variable, a ***volatile*** one, where the keyword "`volatile`" is a variable qualifier. Volatile tells the compiler not to optimize the variable when translating the C code into machine language. Normally, variable usage is optimized by the compiler to save space. If a variable is referenced only once in the code, the compiler might eliminate it, or replace it with "TRUE." The ISR is considered separate from the main code, so the compiler does not see that the variable is used in both places, unless it is ***volatile***. You can get some screwy results that are difficult to correct if you forget this.

The other is the use of `attachInterrupt()` to assign our routine, `GetUser1()`, to interrupt 0 with FALLING edge detect. A FALLING edge is when the voltage on a pin goes from a logic HIGH to a logic LOW, for example when you push the button. Why interrupt 0? Because on the UNO, port 2 is assigned to interrupt 0, and port 3 is assigned to interrupt 1 (keeps us on our toes, I guess).

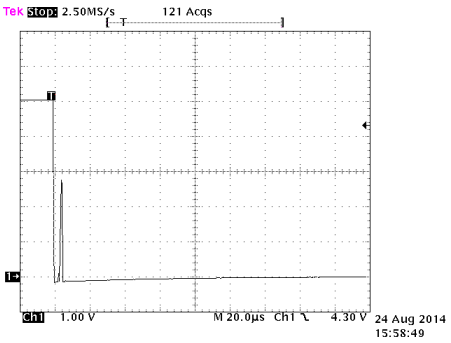
To use it, all you have to do is check to see if that volatile variable is set to 1 in your **`loop()`** routine and act on it. For instance you could increment a counter variable when the volatile variable gets set. Something like this:

```
if (Button1Pushed == 1) {
 counter ++;              // increment our counter variable
 Button1Pushed = 0;       // clear the detect variable
}
```

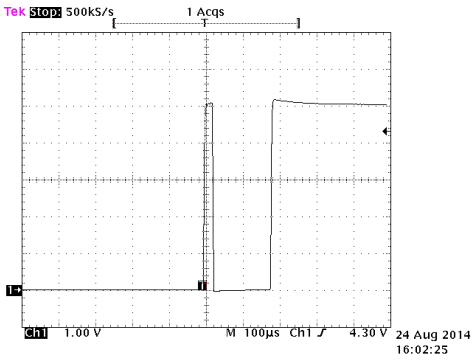
Exercise 8 – Push Button

Create a sketch implementing the code fragments shown in the previous section. Add a Serial.println() to watch the counter variable change (add a delay too, so it doesn't just fly by). *What happens? Does the counter variable always increment by one?* Try holding the button down and releasing. *Do you sometimes get an extra count when you release the button?*

If you answered yes to this question (duh), you are witnessing something called "bounce." Bouncing is a phenomenon associated with the contacts in mechanical switches when they open and close. The contacts don't just open and close, they chatter or bounce, and the switch actually opens and closes rapidly. To illustrate, here are the two waveform scope shots of port 2 from my LightBox:



Button Pushed



Button Released

Notice there's a little hiccup after or before the main rising or falling edge. Our unfiltered ISR will recognize these very short duration edges and act upon them.

What we need is a software "debouncer." It seems everyone has their own debounce code. I like my super simple one even though it uses a delay which in an ISR can be a no-no. A delay is a hog, but these are good push buttons and we'll keep it short, so not too hoggish. All you do is check, or sample the voltage on the port a short delay after the edge has been detected. Let's tweak the ISR to be

```
Void GetUser1 () {
     delayMicroseconds (500);       // wait 500us
                                    // then only set ButtonPushed
                                    // if the port is still low

     if (digitalRead(User1_PB)==LOW) Button1Pushed = 1;
}
```

Here we use `delayMicroseconds()` instead of just `delay()` because with this pushbutton, we can get away with a delay of less than a millisecond. Notice that the rising edge, or release, is actually worse than the falling edge. Referring to the scope shot, 500μs after the false falling edge on the button release, the voltage is a stable logic High, so an interrupt is not triggered and the Button1Pushed variable does not get set to 1.

Exercise 9 – Debounce

Repeat Exercise 8, but this time, make one push button increment the counter and the other push button decrement the counter. Check for debounce. Display the counter value on the LEDs using your binary routine from the previous exercise.

Exercise 10 – "Pong"

Create a 1-dimensional Pong game using the 8 LEDs and the two push buttons. Let the User1 push button have the serve. Let the potentiometer control the speed.

Describe your algorithm and how the game works.

Hints:

Create a subroutine that makes the LEDs go right when User1 is pushed.

Create a subroutine that makes the LEDs go left when User2 is pushed.

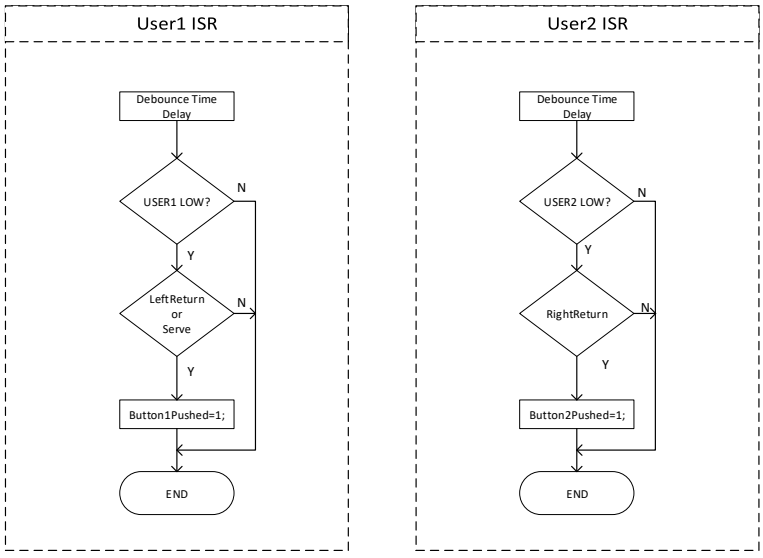
Open a window-in-time using a variable for allowing a return to be sensed when the end LED is illuminated.

An if statement with complex logic may be useful. e.g.

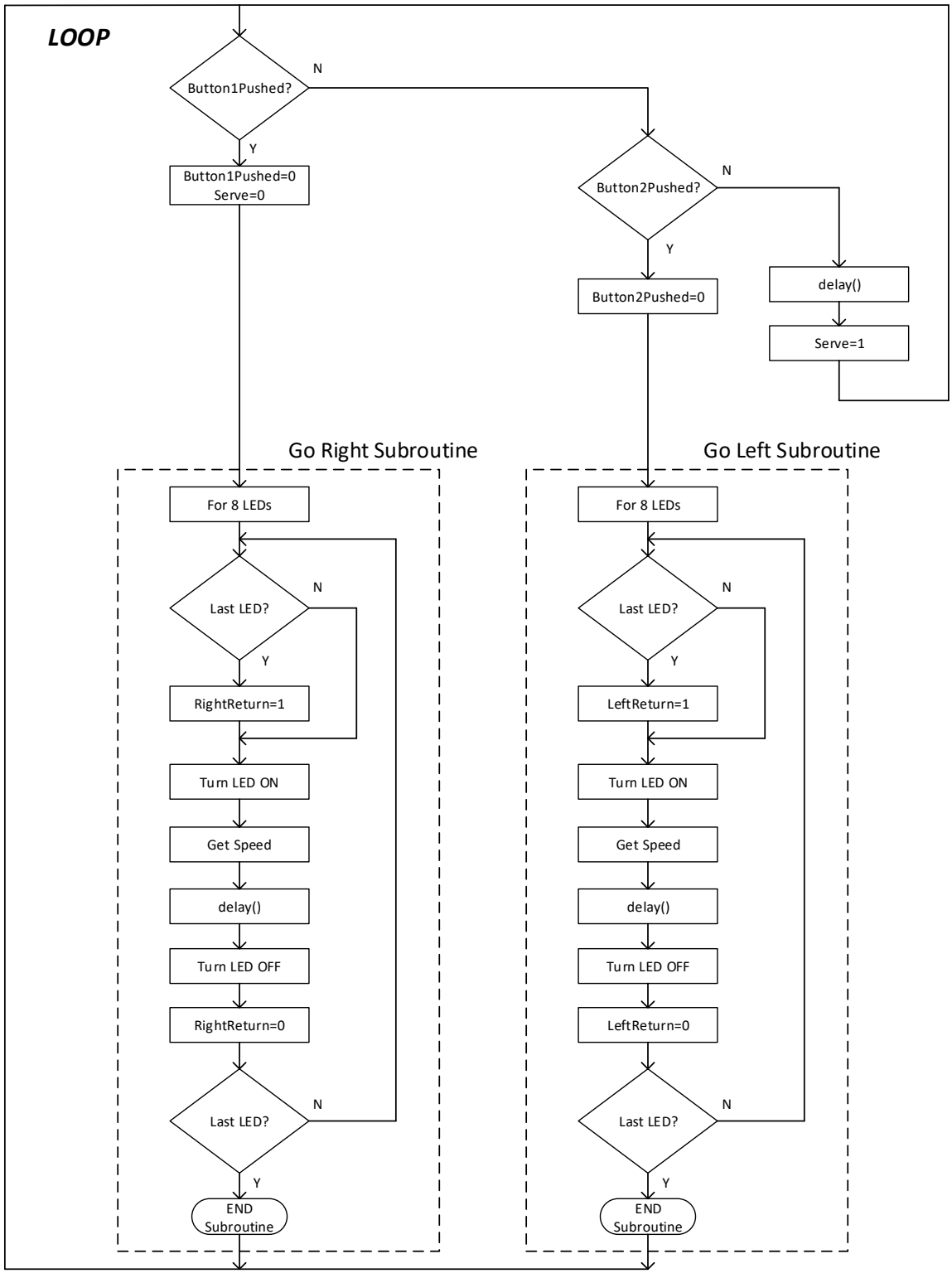
```
if ((a == 0) && (b==1)) <instruction>;
```

Many solutions are possible. Here are a few flowcharts of one possible implementation that may be of use:

Push Button Interrupt Service Routines



PONG Flowchart



Exercise 11 – LightBox

Create a sketch with all of your LED functions (Photo cell, PWM, Binary number, PONG). Sequence between the functions by pressing the User1 and User2 buttons. User1 increments and User2 decrements the function. Initialize the LEDs to OFF when changing functions. For extra credit, find a way out of PONG without using the reset button. Begin with mode 1.

Mode1: Photo Cell (Exercise 4)

Mode2: PWM Dimming (Exercise 6)

Mode3: Binary number display of Potentiometer position (Exercise 7)

Mode4: PONG (Exercise 10)

Create a user's guide for your LightBox with at least one paragraph per mode. Try it on a friend.

Hints:

Use the counter from Exercise 9 to keep track of the function.

Make each function a subroutine.

If a function IS NOT used, set a variable to initialize that function when it IS used. Reset the variable after the LEDs are initialized for the function.

Add PONG last after you have the other modes running (since you will be using the buttons).

Extra:

Find a way to get out of Pong and go to the previous mode.

Be creative and find an entirely new application for the sensors and LEDs.

19. Design, Development and Debugging

No intro to programming and electronics is complete without a little attention to general development principles. At the outset, you should consider yourself a designer because what you implement is your “design”. Down the road, you may find you prefer the discipline of analysis or definition over design, but for now, you are the designer. So let’s think like designers.

To begin with, think of your assignment, and how you want it to look and feel coming from a high level viewpoint; the viewpoint of the user. Then begin to break it into pieces and see how you might go about implementing your design. Then start creating your implementation in code. Don’t be afraid to go back and revisit your definition, or rip up your code. Ripping up code allows you to practice learning the coding language, in this case, C. Always remember the KISS principle (keep it simple stupid). An elegant design is generally a simpler one. Ask yourself if you can simplify throughout the design process.

While developing your code, develop and test in logical sections. The more you test as you go, the easier the debugging will be. There is some debate over simply compiling as you go to see if your syntax is correct, but very little debate about testing as you go. Compiling – whatever, testing – YES! A complex system can be built out of myriad simple individual pieces.

On debugging, always begin by assuming it’s your problem and not someone else’s or the IDE’s or the Arduino’s. Take ownership over your bugs. In searching for causes, remember Occam’s razor, which basically states that the simplest solution is the most likely (or more rigorously, the hypothesis with the fewest assumptions should be selected). It was proposed 700 years ago. I use it all the time. Data even quoted it in a Star Trek episode. I love debugging.

20. Going Further

Often there are students who wish to delve deeper into the subject matter. In this section a few extra topics to consider are presented.

20a. Hysteresis

Without hysteresis, the LSB LEDs in your binary conversion exercise are probably kind of blinky. This is due to power supply noise in the system. Enter hysteresis.

Hysteresis is a common solution to engineering problems where noise is an issue. With the LightBox, there is noise on the power supply when, for instance, the LEDs turn on and off or as the processor goes about its duties. Noise in this case is when a transient power load causes the power supply to change slightly. The Arduino uses a 10-bit analog-to-digital converter with 5v as maximum and 0v as minimum. So the step size, or LSB (least significant bit) of the converter is defined as 5v/1023 (remember $1023 = 2^{10} - 1$). This equals 4.9mV. If the power supply changes by this small amount, and your potentiometer setting is near a quantized state, you get a different reading even though you didn't touch anything. Fortunately, we can use software to implement hysteresis to clean up this sensitivity.

Hysteresis is a way to lock in a value. In this case, it is performed by only allowing the value to change if it changes by more than 1 or 2. Fortunately again for us, we can do this because we are reading a 10 bit value and converting it to an 8 bit value, so we have extra values (4 times extra).

A simple way to think about how to program hysteresis is to test to see if the new value is ***greater than*** the previous value. If it is, use it, and in this way, let the value ratchet up to the highest noise level. To go down, test if the new value is ***less than*** the previous value by 4 or so. If it is, use it. If not, don't use it. In other words, just don't use the numbers adjacent to the number you previously converted. Similar for decreasing values.

So read a new value and compare it to the previous value you have stored, and only use it if it meets that criteria.

With two variables, POT0 and POT, you can

1. Read the new potentiometer value and save as POT0.
2. Compare the new potentiometer value (POT0) with the previously converted number (POT). If POT0 is greater than POT, or if POT0 is less than POT-4, set POT equal to POT0. If not, don't do anything.
3. If (POT0 < POT-4) is true, ratchet down to the lowest noise level, and check if (POT0 > POT+4) to go back up.
4. Convert the value stored in POT to binary.

20b. ATMega Registers

What if we didn't have the Arduino language? Then, you'd have to know what's going on in commands like `digitalWrite()`. You'd find some rather cryptic looking code that uses bit manipulation operators to set and reset individual bits in a thing called a register. Registers hold bit information to say, put 5v or 0v on a particular port like D9 for instance. They are documented in the ATMega328 datasheet (all 463 pages you can find at www.digikey.com or www.microchip.com), which is worth a perusal even if you're not going to take it to this low level. We can see there that registers are generally 8-bits. If you were using a different microcontroller where there is no Arduino language, you'd be doing register bit manipulation.

First let's look at what registers are connected to the ports we are using in the LightBox.

Arduino ATMega Ports and LightBox Functions

Arduino Port	ATMega Port Pin	LightBox Function
13	PB5	
12	PB4	
11	PB3	LED7
10	PB2	LED6
9	PB1	LED5
8	PB0	LED4
7	PD7	LED3
6	PD6	LED2
5	PD5	LED1
4	PD4	LED0
3	PD3	USER2 Button
2	PD2	USER1 Button
1	PD1	
0	PD0	
A5	(ADC5) PC5	
A4	(ADC4) PC4	
A3	(ADC3) PC3	
A2	(ADC2) PC2	
A1	(ADC1) PC1	Photo Cell
A0	(ADC0) PC0	Potentiometer

Now let's take off the training wheels and do some basic operations.

Initialization

```
  DDRD = DDRD | B11110000; // D ports LED7, 6, 5, 4 as outputs

  DDRB = DDRB | B00001111; // B ports LED4, 3, 2, 1 as outputs
```

Turn LEDs ON

```
  PORTD = PORTD | B11110000; // set portD pins HIGH

  PORTB = PORTB | B00001111; // set portB pins HIGH
```

Turn LEDs OFF

```
  PORTD = PORTD & B00001111; // set only LED portD pins LOW

  PORTB = PORTB & B11110000; // set only LED portB pins LOW
```

```
//////////////////////////////////////////////////////////////
// BinaryDirect.ino
//
// LEDS: 7  6  5  4  3  2  1  0
// PORT: B3 B2 B1 B0 D7 D6 D5 D4
//////////////////////////////////////////////////////////////
int POT;

void setup() {
  Serial.begin(9600);
  DDRD = DDRD | B11110000; // D Ports LED7, 6, 5, 4 as outputs
  DDRB = DDRB | B00001111; // B Ports LED4 ,3, 2, 1 as outputs
}
void loop() {
  POT = analogRead(0) >> 2; // read Potentiometer and divide by 4
                            // same as shifting 2 bits to the right
                            // essentially lopping off the 2 LSBs
  PORTD = PORTD | ((POT << 4) & B11110000); // lower nibble set
  PORTD = PORTD & ((POT << 4) & B11110000); // lower nibble clear

  PORTB = PORTB | ((POT >> 4) & B00001111); // upper nibble set
  PORTB = PORTB & ((POT >> 4) & B00001111); // upper nibble clear
}
```

Here are a few notes on the operations in the above code.

The registers in use are defined in the “I/O-Ports” section of the datasheet.

Bit-shift: We can multiply and divide numbers by powers of 2 with a simple shift of bits left and right (it’s also a super-fast way to do it).

>> bit-shift to the right. The instruction `Var = Var >> 1;` does a divide by 2

<< bit-shift to the left . The instruction `Var = Var << 1;` does a multiply by 2

Individual bit manipulation: When we want to set a bit, we “or” that bit with a “1”. When we want to clear a bit, we “and” that bit with a “0”. We do this so that we do not disturb any bits we are not interested in because another function might have set or cleared them and we don’t want to override what another part of our code has done.

For example:

This instruction “sets” the upper four bits of the DDRD register to “1”.

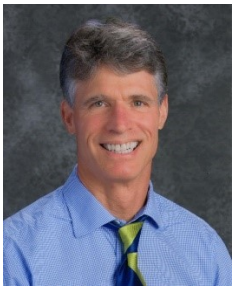
```
DDRD = DDRD | B11110000;
```

In the following two instructions, we have to do an “or” ***and*** an “and” because we need to set and clear bits depending on the value stored in the variable POT, which we do not know. (Upper and lower nibble here refers to the 8 LEDs.)

```
PORTD = PORTD | ((POT << 4) & B11110000); // lower nibble set
PORTD = PORTD & ((POT << 4) & B11110000); // lower nibble clear
```

This is still the C programming language, though it appears perhaps closer to *assembly language*. Nowadays, the microcontroller compilers (what converts C language into numerical lists, or machine code executed directly on the CPU) are so efficient that fewer engineers practice low level assembly coding. Some applications may require that level of detail, and you can take a class in college. In fact, it was a requirement for my BSEE from UCSD, I recall enjoying the class, and I used it once on the job circa 1983 to program an Intel 8051 micro.

21. About the Author



Kevin D'Angelo is a teacher of high school robotics. After a 30 year electrical engineering career he is enjoying living on the CA central coast and exposing students to the engineering profession. This text is an outgrowth of teaching an academic high school robotics elective at Stevenson School.