AN INTRODUCTION TO MICROCONTROLLERS AND EMBEDDED SYSTEMS

Tyler Ross Lambert

MECH 4240/4250 Supplementary Information

Summary

Embedded systems in robotics are the framework that allows electro-mechanical systems to be implemented into modern machines. The key aspects of this framework are C programming in embedded controllers, circuits for interfacing microcontrollers with sensors and actuators, and proper filtering and control of those hardware components. This document will cover the basics of C/C++ programming, including the basics of the C language in hardware interfacing, communication, and algorithms for state machines and controllers. In order to interface these controllers with the world around us, this document will also cover electrical circuits required to operate controllers, sensors, and actuators accurately and effectively. Finally, some of the more commonly used hardware that is interfaced with microcontrollers is gone over.

Table of Contents

1.   Introduction

An **embedded system** is a computer system with a specific, dedicated function that is not designed so that it should ever need to be reprogrammed (i.e. engine control units, implantable medical devices, appliances, etc.) The most common type of embedded system is a **microcontroller**, which is a small computer system on a single integrated circuit. Some common examples of this type of embedded system comes in the form of Arduino or Teensy microcontrollers (Figure 1).
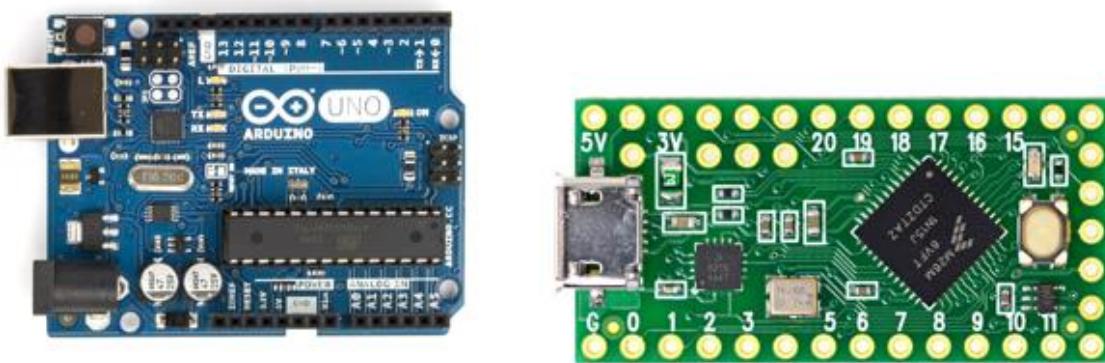
Figure 1. (*left*) Arduino Uno Microcontroller (*right*) Teensy 3.2 Microcontroller

Microcontrollers are adept at performing tasks such as reading sensors and implementing control laws, but it is important to note that these devices are **digital**, which means they are discretized in how they interpret data, in contrast to the real world in which we live which is **analog**, so that everything we see is continuous in nature. In order to reconcile this, a microcontroller will utilize both **digital-to-analog conversion** (DAC) to move from binary values to actual output voltages and **analog-to-digital conversion** (ADC) to move from an input signal to digital data that the microcontroller can use.

Two of the more common microcontrollers are the Arduino and the Teensy models, which will be the ones primarily discussed in this document. Older microcontrollers, such as a PIC microcontroller, might be better suited for alternative tutorials.

There are countless factors that go into selecting which microcontroller is the most suited for any specific project. Different microcontrollers have different sizes, different functionalities, different feature to footprint ratios, different software architectures, varying number input/output (I/O) pins available for use, different power requirements, different processing speeds, etc.

2. Numbering Systems

A **numbering system** is simply a way in which to represent a quantity. The most common numbering system in use is the decimal system (base 10), but other numbering systems such as binary (base 2) and hexadecimal (base 16) are often convenient to use. These numbering systems use symbols (numeric indicators) up to the value of the base in order to express a quantity. For example, the binary system uses the symbols 0 and 1. The decimal system uses 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The hexadecimal numbering system uses the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, where A represents the number 10 in decimal, B represents 11, and so on.

These symbols will be placed in a location in the number, and this location is known as its **digit**. The digit location, starting from right to left (reflecting smallest to largest quantity) is designated by a subscript on the constants in the following table that shows how each numbering system functions with respect to these digits and the system's base value (the mathematical expression being the conversion from the system to the standard decimal notation):

Numbering System	Base	Mathematical Expression
Decimal	10	$D = \cdots d_2 10^2 + d_1 10^1 + d_0 10^0$
Binary	2	$B = \cdots b_2 2^2 + b_1 2^1 + b_0 2^0$
Octal	8	$O = \cdots o_2 8^2 + o_1 8^1 + o_0 8^0$
Hexadecimal	16	$H = \cdots h_2 16^2 + h_1 16^1 + h_0 16^0$

Example. What is the decimal representation of the hexadecimal value A6?

$$A6 = A\{10\} \times 16^1 + 6 \times 16^0 = \mathbf{166}$$

To designate which numbering system is being used, common practice is to have the number be preceded by 0d (for decimal), 0b (for binary), 0o (for octal), or 0x (for hexadecimal).

3.   Variable Types and Memory

Data in a microcontroller is stored at its basest level as binary values. Each digit of a binary value is termed a **bit**. A collection of eight bits is known as a **byte**. This means that the working range for a byte is between 0b00000000 and 0b11111111 (which corresponds to decimal 0 to 255). This convention was started, in part, because of ASCII codes. In hexadecimal, a byte can range in value from 0x00 to 0xFF. Note that 4 bits of binary corresponds with one hexadecimal digit.

When a variable in a microcontroller is initialized, its value is stored using bytes. There exist several fundamental variable types/structures available for use (called **type definitions**):

- **Integers** – whole numbers, can be stored as signed values or unsigned values
- **Float** – short for floating point precision, can store decimal values
- **Double** – short for double point precision, can store decimal values with twice the precision as a float.
- **Characters** – unsigned eight byte values which are initialized as characters
- **Boolean Values** – values that are either 0 or 1

An integer takes binary values and gives the corresponding decimal value, if it is an unsigned integer. If it is a signed integer; however, the first bit corresponds with the sign of the value (0 for positive, 1 for negative). A basic example chart is below for three bit binary values:

Table 1. Three-bit binary Corresponding Signed and Unsigned Integer Values

Bits	Unsigned Integer Value	Signed Integer Value
011	3	3
010	2	2
001	1	1
0b000	0	0
101	5	-1
110	6	-2
111	7	-3
100	4	-4
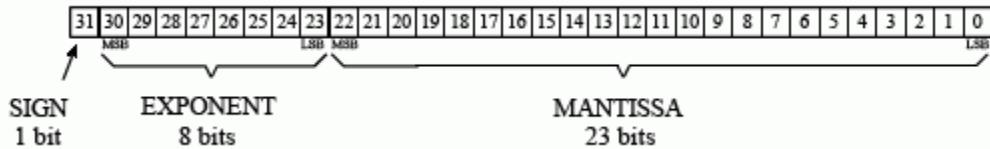
For an eight-bit unsigned integer, values between 0 and 255 can be stored. For an eight-bit signed integer, this range is shifted to -127 to 127. To add or subtract signed binary values, use the two's complement method where each bit is inverted before being added or subtracted to yield the correct result.

The most important thing to note about integers is that they do not store any information behind the decimal point. Any calculations involving integers that results in non-integer values will truncate anything after the decimal point, which can ruin the computation. This can be addressed by scaling up the integer variable values to avoid decimal places during calculation to some certain level of precision, but this is often not convenient.

Numbers that are stored as floating point types have the following four-byte data structure:

$$float = (-1)^{[sign]}(1 + [mantissa]) \times [base]^{[exponent]}$$

These bracketed values correspond to information stored in the binary of the floating point number. A floating point number is stored in four bytes (32 bits) that are structures like so:

The very first bit of a float is the signed bit that dictates the sign of the value. The next eight bits of information form the exponent byte, which gives an eight-bit integer value. This exponent value is biased by -127 from the pure decimal equivalent of the byte. The next 23 bits give the mantissa (fractional part of number), which is a number that ranges from 0 to 1 in decimal and is given by:

$$mantissa = M_1 2^{-1} + M_2 2^{-2} + \cdots + M_{23} 2^{-23}$$

Double point precision types use this same structure, but offer three additional bits for the exponent (so that the exponent can range from -1022 to 1023) and offers 29 more bits for the mantissa. This increases the precision on decimal values over the single point precision floating point types, at the expense of eight bytes of memory storage.

The character type converts a series of characters to unsigned eight bit integers. Each alphanumeric letter has a numeric equivalent. Characters can be printed either as the character (using %c) or as their integer equivalent. **Characters are NOT strings.** Characters are denoted by single quotes in C (i.e. 'A'); whilst strings use double quotes.

Type Identifiers

When using a microcontroller and initializing variables, each variable must have its type specified upon initialization, with the type keyword being placed immediately before the variable name. To specify an integer variable, the keyword `int` (for a signed integer) or `uint` (for an unsigned integer) could be used, but the amount of bits allocated to an integer varies with the computer. For portability of a program, it is better to use the type identifier **int*xx*_t/uint*xx*_t** (where xx corresponds with the number of bits to be used to variable storage). The type identifier for characters is **char**, for floating point numbers, it is **float**; and for double point precision values, it is **double**.

There are other type definitions, such as **long** (which is an integer value that is usually fairly large; by convention it is twice as large as int) or **bool** (which is a *boolean value*, meaning that it is a single bit reflecting 0 or 1). Another type definition is to declare a variable a **byte**. This allocates exactly eight bits of data for the variable which when references returns an unsigned integer between 0 and 255.

> **Example.** Store a signed 16 bit integer variable as the decimal value 32.
>
> int16_t variable_name = 32;

Attempting to store a value larger than the variable can store based on the number of bytes it is allocated will cause a "roll-over" effect. For example:

```
1   uint8_t var = 256;
```

This code attempts to store 256 into an unsigned eight bit variable, despite the fact that an eight-bit unsigned binary value can only range from 0 to 255. Because of this, the code will store the variable as the number 1.

Variables can temporarily be used as another type in what is known as **casting**. This is done by referencing the new type in parenthesis before the variable name during a calculation. This is very useful for doing math with integers, because it allows the user to avoid having decimals points truncated.

Example.

```
1  uint8_t var = 3;
2  float var1 = (float)var*0.1;
```

This returns `var1 = 0.3` as a floating point number.

Unions

If one space in memory is requested to store a variety of data types, **unions** can be used to accomplish this task. A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose. The syntax can be seen from the following example, initialize the union Data to store an integer, a floating point number, and a character string:

```
1 union Data {
2    int i;
3    float f;
4    char str[20];
5 }
```

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. To store this union as any of these data types, the following type of code could be used:

```
1  // Create union data from the Data union template
2  union Data data;
3
4  // Store data as integer "10"
5  data.i = 10;
6
7  // Store data as float "220.5"
8  data.f = 220.5;
9
10 // Store data as character string "C Programming"
11 strcpy(data.str, "C Programming");
```

4. Basic C/C++ Notes and Code::Blocks

To begin playing around with C and/or C++, one great open source program to use is **Code::Blocks**. The download link is available on their website **here**. Code::Blocks is a free, open source program that supports many compilers for several different programs, so it's a nifty tool for anyone looking to become proficient in a language. The program also has full breakpoint support, meaning that the code can be stepped through line by line in debug mode, which is a luxury that a microcontroller will not afford the user.

NOTE: Upon downloading Code::Blocks, make sure to follow the instructions to include the **GNU GCC compiler.** If installed correctly, inside of the Code::Blocks folder (likely inside of C:\Program Files (x86)\CodeBlocks), there should be a folder called **MinGW**.

- C projects can either be created as a **header file (.h)** or a **source file (.cpp)**. For execution of code done for practice, the source file is the project type that should be chosen.
- In C, code is executed in loops. The **main()** loop is the heart of the code, and returns values of the variable type indicated by the type identifier keyword specified immediately before the loop. The value to be returned is addressed by the **return** statement.
- A hashtag (#) indicated a compiler directive, meaning that these parts of the code are executed while the code compiles. A common example is the **#include** statement, which allows for additional **libraries** to be added to the program. A library is additional code that can establish functions for the user of the library to call upon instead of having to code them themselves. Several important libraries are required to utilize the full functionality of Code::Blocks, and these are shown along with their compiler directive to below:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
```

- The **sizeof(input)** function reveals the size, in bytes, of the input expression.
- The **printf("*string*", var1, var2, …)** function operates the same as in **MATLAB**
- Constants can be declared, and upon compiling of the code the constant will be replaced by its value to optimize memory storage. The syntax for this is simply:

```
1 | const int myConst = 3;
```

- Incrementing and decrementing a variable by 1 is done by using "++" or "--" after the variable name in a line of code. This is a shortcut provided by C and C++.
- Including the **cmath.h** library enables the use of mathematical functions. For example, the pow(base, exponent) function returns the value of the base input to the power specified by exponent. Other math functions include sqrt(num) that returns the square root of num, and many more covered here.
- Comments are started by double backslash (i.e. // this is a comment). A multiline comment can be initiated with an asterisk and a backslash, as shown: /* .... */
- A semicolon (;) terminates a line of code, and is required at the end of each completed line of code.

5.   Bitwise Operations

Bitshifting is the mathematical operation denoted by "<<" and ">>" that shifts every bit over to the left or to the right, respectively.

Example.

```
0b00000001 << 1 = 0b00000010
0b00000100 >> 1 = 0b00000010
```

This can be extremely useful in the situation where the user has two eight bit numbers and wishes to combine them into a sixteen bit value. The following code would accomplish this task:

```
1  uint8_t low_byte = 34;      //0b00100010
2  uint8_t high_byte = 250;   //0b11111010
3
4  uint16_t both_bytes = (high_byte << 8) + low_byte;
```

This code would return the decimal equivalent of 0b1111101000100010, or 64,034. Should this has only been allocated eight bits of memory, it would have only returned 34, the value of `low_byte`, because the variable would only use the lowest eight bits of information.

The **bitwise NOT** operation turns all 0s in a binary number into 1s and vice versa. In C++, this operation is denoted with a tilde character (~), like so:

```
1   int a = 103;     // binary:  0000000001100111
2   int b = ~a;      // binary:  1111111110011000 = -104
```

The **bitwise AND** operation is denoted by combining two binary values with an ampersand character (&). This operation will return a 1 in place of a digit of the new binary value if both of the combined binary value have a 1 as that digit, and will return a 0 for that digit otherwise, like so:

```
1  int a =  92;    // in binary: 0000000001011100
2  int b = 101;    // in binary: 0000000001100101
3  int c = a & b;  // result:    0000000001000100, or 68 in decimal.
```

The **bitwise OR** operation is denoted by combining two binary values with a vertical bar character (|). This operation will return a 1 in place of a digit of the new binary value if either of the combined binary value have a 1 as that same digit, and will return a 0 if both binary values have a 0 as that digit, like so:

```
1  int a =  92;    // in binary: 0000000001011100
2  int b = 101;    // in binary: 0000000001100101
3  int c = a | b;  // result:    0000000001111101, or 125 in decimal.
```

The **bitwise XOR** (exclusive OR) operation is denoted by combining two binary values with a carat character (^). This operation functions like an OR operation in that a 0 is returned if both binary values have a 0 for that digit, and a 1 is returned for the digit if either binary value has a 1 as that digit, but whereas the bitwise OR will return 1 if both binary values have a 1 as the digit, the XOR operation would instead return 0, like so:

```
1  int x = 12;     // binary: 1100
2  int y = 10;     // binary: 1010
3  int z = x ^ y;  // binary: 0110, or decimal 6
```

6. Arrays and Matrices

Arrays are group of values that span in a list. Each value of an array must be of the same data type, specified by the type identifier before the array. The syntax for initializing an array in C is given in the following example:

```
int myArray[4] = {45, 46, 47, 842};
```

The number in the square brackets is the number of elements in the array, and the array values are contained within the curly brackets. Arrays in C are zero-indexed, so the index of array values starts at 0 (unlike in MATLAB, where the first entry in an array is index 1). To reference a value in an array, take the following example:

```
printf("val = %d \n", myArray[3]);
```

This will simply return the string "val = 842".

Matrices are groups of information that is stored in columns as well as rows. If an array is a 1-D data element, then a matrix would be considered a 2-D data element. Matrices in C are initialized with the following syntax:

```
int myMatrix[2][4] = {{2, 4, 6, 8}, {10, 12, 14, 16}};
```

The first value inside of the square brackets is the number of rows, while the second is the number of columns. Referencing the values inside of a matrix is the same as with arrays, with the indexing starting at [0,0].

A very useful library exists in the form of **MatrixMath.h** that enables some matrix math functions that are commonly employed to be used from within C Code. The library is available on Arduino's website and is located here. It enables matrix or vector algebra and includes the following functions:

- `Matrix.Print`
- `Matrix.Copy`
- `Matrix.Transpose`
- `Matrix.Multiply`
- `Matrix.Add`
- `Matrix.Subtract`
- `Matrix.Scale`
- `Matrix.Invert`

The exact syntax and how to use the library is available in the attached link. Be forewarned to check the dimensions of your matrices beforehand, as this library will not return an error if matrix dimensions are mismatches.

7.   Loops

An iterative loop is a block of code that repeats itself a set number of times. Two main examples of these are **for loops** and **while loops**.

An example of **while** loop syntax is given below:

```
1  int i = 0;           // Initialize counter index variable i as an integer
2
3  while (i < 3)        // Run if i is less than 3
4  {
5     i++;              // Increment i up by 1
6  }
```

This example code will run the loop three times over, incrementing the counter index up by one upon every iteration of the loop.

A **for** loop is initialized via `for(variable, condition, increment)`. The variable in this expression must have its type defined, but the general syntax for a `for` loop can be demonstrated with a simple example:

```
1  for (int i = 0; i < 3; i++)
2  {
3     // code
4  }
```

This code accomplishes the same task as the while loop above. C also allows for a **do**…**while** loop, which is essentially a while loop that is guaranteed to run at least one time, but rarely can such a task not be accomplished with a `for` or a `while` loop.

Note that in C, any variable created within a loop can only be accessed inside of that loop or at even lower levels within loops that are nested inside of that loop. The areas of the code where the variable can be referenced is known as the variable's **scope**. If the variable is created outside of the loop, and is changed within the loop, the changed value is the value that will be accessed when the variable is accessed. Note that in this context, loop does not only refer to iterative loops, but to any block of code bounded by brackets. This includes function expressions and even the main loop itself.

8.   Logical Statements

The standard **if/else** logic used in most other languages is present in C, using the following syntax:

```
1  if (variable == condition)
2  {
3      // code
4  }
5  else
6  {
7      //code
8  }
```

Make note of the double equals sign symbols inside of the `if` statement. This corresponds with logical tests, whereas a single equals sign corresponds with assigning a variable a new value. The logical tests that can be used include:

```
1  if (variable == condition) {}    // Equals to
2  if (variable != condition) {}    // Not Equals to
3  if (variable > condition) {}     // Greater than
4  if (variable < condition) {}     // Less than
5  if (variable >= condition) {}    // Greater than or equal to
6  if (variable <= condition) {}    // Less than or equal to
```

The standard **switch/case** alternative to if/else logic does exist in C/C++. The syntax of this is seen below in a simple example:

```
1  int main()
2  {
3      char grade = 'B'               // assigns the grade
4
5      switch (grade)                 // opens the switch statement with the argument being the value of the grade
6      {
7          case 'A':                  // only executes if the grade was an A
8              printf("Excellent! \n");
9              break;                 // breaks out of the switch statement
10         case 'B':
11         case 'C':
12             printf("Well done \n");    // both grades of a B or a C result in the same outcome
13         default:
14             printf("Better try again!");
15     }
16 }
```

This example all takes place inside of the `main()` loop, and prints different results based on the result of the input of the variable grade.

9. Enumerations

An **enumeration** consists of a set of named integer constants (called the "enumeration set," "enumerator constants," "enumerators," or "members") and is considered a variable type. An enumeration type is initialized with **enum**. The elements inside of the enumeration, for all intents and purposes, behave the same as constants upon compilation of the code.

The example below demonstrates its usage:

```
1   enum DAY
2   {
3       saturday,
4       sunday = 0,
5       monday,
6       tuesday,
7       wednesday,
8       thursday,
9       friday
10  };
11
12  workday = (enum DAY)(day_value - 1);
```

This code would return a day based on the input of `day_value`.

10.   Compiler Directives

A **compiler directive** tells the compiler to compile or skip blocks of code during compiling based on some criteria during preprocessing of the code. These statements are prefaced with the hashtag symbol and are very helpful in regards to memory storage. Note that no semicolon is required to terminate the end of a line of code that consists of a compiler directive. An example is provided that would place the program in debug mode based on chip architecture and user input:

```
1   #define ARM
2   #define DEBUG   //comment out to not run debug mode
3
4   int main()
5   {
6       #ifdef ARM
7         printf("Chip Architecture is ARM");
8         #ifdef DEBUG
9           printf("Debug Mode");
10        #endif
11      #endif
12  }
```

Source lines handled in preprocessing, such as #define or #include are called **preprocessing directives**. Conditional compilation is handled via the `#if`, `#ifdef`, `#ifndef`, `#else`, `#elseif`, and `#endif` directives. The `#error` compiler directive can also be used to throw an error during the program.

11.   Pointer Variables

A variable is always stored at a certain location in memory, called its **address**. The address can be referenced using the & symbol (the **address-of operator**) followed directly by the variable name. **Pointer variables** are variables that reference the starting position of another variables' address and return the value held at that address. These variables are initialized by starting with an asterisk (called the **dereference operator**). This operator can be seen as saying "value pointed to by" An example is provided:
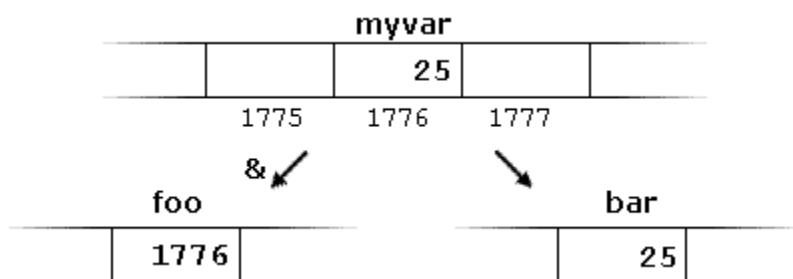
```
1   int main()
2   {
3      int A = 4;
4      printf("The value of A is: %d \n", A);
5      printf("A is stored starting at: %d \n", &A);
6
7      int *ptr = &A;   // this references the starting position of the address of A
8
9      A = 6;
10
11     printf("The value stored at %d is %d \n", &A, *ptr);
12  }
```

To visualize what is actually happening in regards to memory storage, take the very simple example:
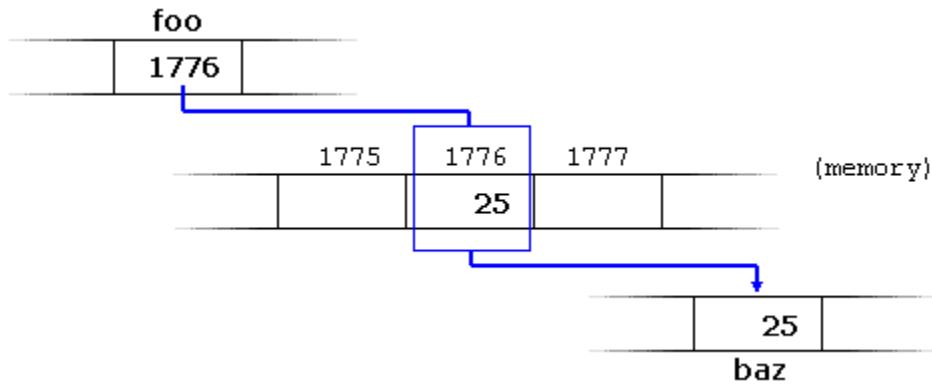
```
1  myvar = 25;
2  foo = &myvar;
3  bar = myvar;
4  baz = *foo;
```

The actual address of a variable in memory cannot be known before runtime, but let's assume, in order to help clarify some concepts, that `myvar` is placed during runtime in the memory address 1776. The values contained in each variable after the execution of the first three lines are shown in the following diagram:

Obviously, the variable `foo` is the pointer variable, because it "points to" the address of the variable it stores (`myvar`). The fourth line example could be read as: "`baz` equal to value pointed to by `foo`", and the statement would actually assign the value 25 to `baz`, since foo is 1776, and the value pointed to by 1776 (following the example above) would be 25.

In memory, this can be represented as:

Accessing data in an array is also possible using pointer variables, but the syntax to specify the address is the (array_name + index). An example of this is below:

```
1   int array[] = {4, 40, 400};
2
3   printf("Calling array[1] directly yields: %d", array[1]);
4   printf("The address of array[1] is %d", (array + 1));
5   printf("Calling array[1] using pointers: %d", *(array + 1));
```

12.   Functions

Functions in C can be initialized (termed **prototyping** if done at the start of code) with the following syntax:

```
type function_name(input_1, input_2, …)
```

The type is the variable type of the returned value of the function. If the function does not return any value at all, then this type should be left as **void**. If the inputs are not defined previously, their types must also be defined. A simple example is a function that takes an integer Q and seeks to return the values of this input incremented upwards and downwards by one. An attempt to accomplish this is provided below:

```
1  int updnFun(int Q, int up)
2  {
3      up = Q + 1;
4      int dn = Q - 1;
5      return dn;              // returns the integer dn
6  }
```

This function, if called, would return the value of dn, but the value of up cannot be accessed inside of any loop other than the function itself. The trouble with C is that functions can only explicitly return one value. Functions can also edit variables already created before the function is called, which can then be used outside of the function. This is inconvenient, and so pointers can be used in order to return more than one value from a function by directly editing the variable stored in memory. An example of this would be an updated function from the one featured above:

```
1  int updnFun(int Q, int *up)
2  {
3      *up = Q + 1;
4      int dn = Q - 1;
5      return dn;              // returns the integer dn
6  }
```

Now, the values of Q and the address of up are the inputs. The variable dn is initialized, modified, and returned. The memory that contains up is modified and this memory location contains the modified version of up for future use.

Perhaps even more useful is that if a function is called as a pointer itself, it can be used as an argument inside of another function. An example of this is provided using basic math such as addition and subtraction as the functions to be passed as an argument into another function:

```
1  // Pointer to Functions Example
2  #include <iostream.h>
3  using namespace std;
4
5  // Addition Function
6  int addition (int a, int b)
7  { return (a+b); }
8
9  // Subtraction Function
10 int subtraction (int a, int b)
11 { return (a-b); }
12
13 // Function that takes in two integers and a function as aruments
14 int operation (int x, int y, int (*functocall)(int,int))
15 {
16    int g;
17    g = (*functocall)(x,y);
18    return (g);
19 }
20
21 int main ()
22 {
23    int m,n;
24    int (*minus)(int,int) = subtraction;
25
26    m = operation (7, 5, addition);
27    n = operation (20, m, minus);
28    cout <<n;
29    return 0;
30 }
```

Another point to note about functions is that the keyword **static** before a variable type indicates that the function is always to use the value assigned to the variable from the last time the function was called.

13.   Structures

An array is used to bundle several values of the same type. To hold several data types together, a structure can be used. Structures are initialized using the **struct** keyword. The structure must have its contents defined, and then any structure can be created using the structure template. An example of initializing a structure is given below:

```
1   // Create the structure template named "person"
2   struct person{
3       int age;
4       float height;
5       char grade;
6       char name[10];        //allocate ten characters for a name
7   };
8
9   // Create the structure using the "person" template
10  struct person Bob;
11
12  Bob.age = 22;
13  Bob.height = 6.5;
14  Bob.grade = 'A';
15  sprintf(Bob.name, "Bob");
```

One benefit of structures is that they can be passed as an output of a function, so this a means of having multiple outputs returned from a function through the use of the return statement. The syntax for this type of function is provided with the following statistics example, in which an input array of float type variables is taken in and the maximum, minimum, mean, and standard deviation and returned in a structure of floats.

```
1   struct statistics statistics_array(float float_array[], int size)
2 ⊟ {
3
4     // Create the variables to be used in this function as floating point types
5     float maximum, minimum, mean, stdev, sum, deviation, deviation_sum;
6
7     // Initialize variables for the maximum, minimum, and array sum as the first value of the array
8     maximum = float_array[0];
9     minimum = float_array[0];
10    sum = float_array[0];
11
12    // Loop through each array entry
13    for (int c = 1; c < size; c++)
14 ⊟ {
15      // If new maximum value is found, change the value of maximum
16      if (float_array[c] > maximum)
17 ⊟   {
18          maximum  = float_array[c];
19      }
20      // If new minimum value is found, change the value of maximum
21      if (float_array[c] < minimum)
22 ⊟   {
23          minimum = float_array[c];
24      }
25
26      // Keep a running sum of the array entries
27      sum = sum + float_array[c];
28    }
29
30    // Divide the running sum by the size of the array to obtain the mean
31    mean = sum/(float)size;
32
33    // Initialize the variable deviation_sum
34    deviation_sum = 0;
35
36    // Loop through the array computing each squared deviation from the mean and sum these values
37    for (int c = 0; c < size; c++)
38 ⊟ {
39      deviation = pow(float_array[c] - mean, 2);
40      deviation_sum = deviation_sum + deviation;
41    }
42
43      float deviation_sum_average = deviation_sum/(float)size;
44
45    // Compute the standard deviation by taking the square root of the sum of the squared deviations
46    stdev = pow(deviation_sum_average, 0.5);
47
48    // Save these results inside of a statistics structure
49    struct statistics r = {maximum, minimum, mean, stdev};
50
51    // Return the structure containing the statistics data
52    return r;
```
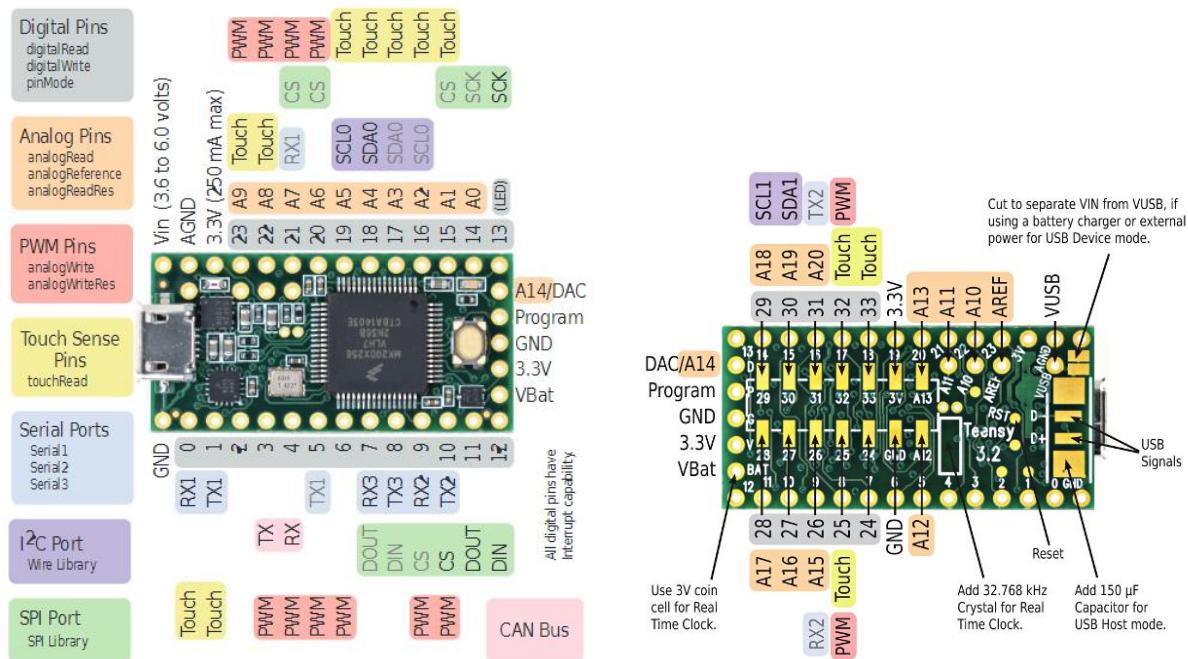
14. Microcontrollers and the Arduino IDE

The software package **Arduino** offers an IDE that allows for the user to write C programs for a microcontroller and is available for download here. Through the addition of the **Teensyduino** software package (available here), the same environment can be used for development with the **Teensy 3.2** microcontroller. Programs developed in this IDE are known as **sketches**. Several example sketches are provided with Arduino that show off the functionality of the program.

Embedded programs (sketches) have two primary loops: `setup`() and `loop`(). It should be noted that these have types of **void**, meaning they do not return any values. The `setup`() loop runs exactly once upon startup of the board, and the `loop`() loop then runs in perpetuity in a loop after completion of the `setup`() loop.

Variables that are global in scope are declared *before* the `setup`() loop (remember, variables are only accessible within their specific scope, and so a variable declared before all loops are run can be accessed anywhere in the sketch). The `setup`() loop is typically used to declare what the microcontroller **pins** will do and initialize other aspects of the program.

A pin is a specific metal interface that allows the electronics on the microcontroller to interact with the surrounding world. The pinout of the Teensy 3.2 microcontroller, for example, looks like:

There are a number of different types of pins, which are summarized below:

1) **Digital Pins** – pins that can read or write signals that are HIGH or LOW only
2) **Analog Pins** – pins that can read analog signals to within the resolution of the ADC
3) **PWM Pins** – pins that can write analog voltages to within the resolution of the DAC.
4) **Touch Pins** – pins that can sense capacitance
5) **Serial Ports** – pins that communicate using Serial Protocol (typically UART)
6) **I2C Ports** – pins that communicate with sensors using the I2C Protocol
7) **SPI Ports** – pins that communicate with sensors using the SPI Protocol

The details of each of these will be gone into a later.

To actually power a microcontroller, an external power supply can be used (as seen above, for the Teensy 3.2 this power would be provided to Vin and would range from 3.6 V to 6 V), or the microcontroller can be powered from the 5 V rail from USB and hooked into a computer.

The simplest program to run on a microcontroller is one that blinks the built in **LED** (Light Emitting Diode). As seen on the Teensy 3.2 pinout sheet, the built in LED is located on digital pin 13. A program that would blink this LED is provided below:

```
1  // Blink Program
2
3  // The setup loop runs once when you press reset or power the board
4  void setup() {
5    // initialize digital pin LED_BUILTIN as an output.
6    pinMode(LED_BUILTIN, OUTPUT);
7  }
8
9  // This loop runs over and over again forever
10 void loop() {
11   digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
12   delay(1000);                       // wait for a second
13   digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
14   delay(1000);                       // wait for a second
15 }
```

There are a number of functions that appear in this simple code that have not been addressed. These functions are all part of the **Arduino.h** library, which is a library that does not require a compiler directive to include, as it is built into all sketches made in the Arduino IDE. These functions are outlined below, and should be remembered as they are used in basically all sketches:

`pinMode(pin, mode)`

Purpose: Configures the specified pin to behave either as an input or an output. See the description of digital pins for details on the functionality of the pins.

Parameters: *pin*: the number of the pin whose mode you wish to set
   mode: INPUT, OUTPUT, or INPUT_PULLUP.

`digitalWrite(pin, value)`

Purpose: Write a HIGH or a LOW value to a digital pin. If the pin has been configured as an OUTPUT with `pinMode()`, its voltage will be set to the corresponding value: 5V (or 3.3V on 3.3V boards such as the Teensy 3.2) for HIGH, 0V (ground) for LOW. If the pin is configured as an INPUT, `digitalWrite()` will enable (HIGH) or disable (LOW) the internal pullup on the input pin. It is recommended to set the `pinMode()` to INPUT_PULLUP to enable the internal pull-up resistor. See the later sections on pullup resistors for more information.

Parameters: *pin*: the number of the pin whose mode you wish to set
   value: HIGH, LOW

`digitalRead(pin, value)`

Purpose: Reads the value from a specified digital pin, either HIGH or LOW, and returns this value.

Parameters: *pin*: the number of the pin whose mode you wish to set
          value: HIGH, LOW

`delay(time)`

Purpose: Pauses the program for the amount of time (in milliseconds) specified as parameter. (There are 1000 milliseconds in a second.)

Parameters: *time*: the number of milliseconds to pause (*unsigned long*)

A couple of important notes to make here:
- When reading a digital signal, microcontrollers will consider voltages higher than a certain threshold HIGH and lower than a certain threshold LOW. Voltages in between these thresholds will return inconsistent results. There is more on the idea of voltage later in this document.
- Using `delay()` with a negative time often crashes the program if it even compiles at all.

Another simple example that uses these functions is one that will set pin 13 (the LED pin) to the value of whatever pin 7 is (either HIGH or LOW), as shown:

```
int ledPin = 13; // LED connected to digital pin 13
int inPin = 7;   // pushbutton connected to digital pin 7
int val = 0;     // variable to store the read value

void setup()
{
  pinMode(ledPin, OUTPUT);     // sets the digital pin 13 as output
  pinMode(inPin, INPUT);       // sets the digital pin 7 as input
}

void loop()
{
  val = digitalRead(inPin);    // read the input pin
  digitalWrite(ledPin, val);   // sets the LED to the button's value
}
```

The exact method of getting pin 7 to read a digital HIGH or LOW will be discussed later.

15. Electricity and Basic Electronic Components

Electricity

       Electricity is the physical phenomena that has to deal with the flow of **charge**. Charge is a quantity of interest simply because oppositely charged particles (that is, positively and negatively charged) attract each other while like charged particles repel each other. The base unit of charge is known as the **Coulomb**. This force is the provider of the energy used in all of these applications, because the geometry and concentration of charged particles creates what is known as an **electric field** which creates a potential energy in all charged particles located in the field. By convention, the direction of the electric field lines move outward from positively charged particles and towards negatively charged particles to reflect the direction of the force that would be enacted to a test charge inside of that field, as shown:
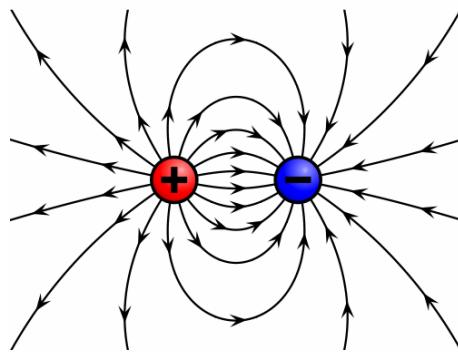
Figure 2. Electric Field Diagram

Coulomb's Law gives the magnitude of force between two charged particles (q_1, q_2) separated by a distance r such that:

$$F = k\frac{q_1 q_2}{r^2}$$

This looks very similar to the force that gravity enacts between two bodies with mass, except instead a gravitational constant *G,* there is **Coulomb's constant** k ($k \approx 9\times10^9 \; \frac{\text{Nm}^2}{\text{C}^2}$). The magnitude of the electric field itself is obtained by placing a test charge of one Coulomb inside of an electric field and observing the force applied to that test charge such that:

$$E = \frac{F}{q} = k\frac{q_{source}}{r^2}$$

Worth noting is that Gauss' Law, which uses the vector quantities of all of these values states that inside of any enclosed volume with surface area *A*, the integral of the electric field can be found so that:

$$\oint \vec{E} \cdot d\vec{A} = \frac{Q_{enclosed}}{\varepsilon_0}$$

where ε_0 is the **electric permittivity** ($\varepsilon_0 \approx 8.85 \; \frac{F}{m}$ where F is Farads, a unit discussed [here](#)). The derivation of this statement comes from the divergence theorem of multivariate calculus.

Much like gravitational potential energy is accumulated as one moves an object through a gravitational field, electrical potential energy is accumulated as one moves a charge through an electrical field a distance d so that:

$$\Delta PE = -Fd \text{ (assuming constant force and motion parallel to force)}$$

If one looks at the potential energy of a particle with one Coulomb of charge inside of the electric field, one obtains the **voltage** at that point in the electric field (measured in Volts [V], which is actually Joules/Coulomb). This is the electrical potential energy per unit charge, which can then directly be related to the electric field:
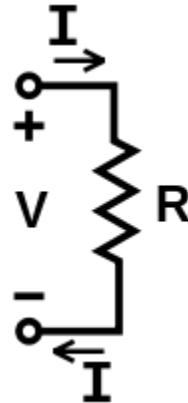
$$\Delta V = -Ed$$

If left free to move, a voltage will induce motion in charged particles, causing them to move. Because voltage is a potential energy due to placement in an electric field, much like gravity causes potential energy in objects in a gravitational field, voltage is a *relative measurement*. This means that to measure voltage, one must measure it against a reference voltage, which is commonly termed **ground** (or 0 V).

A voltage differential can be created through chemical reactions (such as in a battery) or by mechanical action (such as in a generator or alternator). The motion of charge caused by a voltage differential is termed **electric current** and is measured in Amperes [A] (which is actually Coulombs/second). An applied voltage differential is directly proportional to the current it creates via **Ohm's Law**, stated here:

$$\Delta V = IR$$

where ΔV is the voltage differential across the conduit, I is the current through the conduit, and R is the **resistance**, measured in Ohms.

The **power** consumed by a resistive element is simply the voltage drop across the element multiplied by the current across the element:

$$P = VI = \frac{V^2}{R} = I^2 R$$

Across something like a resistive element, this power will simply be given off as heat, but inside of something like a DC motor, it can be used to generate useful work.

It is worth noting that electric fields are present between static charges, but as charges begin to move, a **magnetic field** is generated. and the cumulative effect of the electric and magnetic fields is provided in the **Lorenz Force Law** in which the vector values of the force and field quantities are required:

$$\vec{F} = q(\vec{E} + \vec{v} \times \vec{B})$$

E is the familiar electric field in vector form so that it indicates directionality as well as the magnitude and q is again the charge inside of the field generated by a source charge. But now, the charge is moving with velocity v through a magnetic field B. The **Biot-Savart Law** gives the magnetic field such that for constant current I:

$$\vec{B} = \frac{\mu_0 I}{4\pi} \int \frac{d\vec{\ell} \times \hat{r}}{r^2}$$

μ_0 is an empirically gathered constant called the **magnetic constant** or **vacuum permeability** of space which is found as roughly $1.256 \frac{Vs}{Am}$. $d\vec{\ell}$ is a vector line element in the same direction of the current reflecting an infinitely small portion of wire. r is given as the distance between the location of $d\vec{\ell}$ and where the magnetic field is being calculated, and $\hat{r}$ is a unit vector in the direction of r (and so will point radially outward from the conduit). Due to the nature of the cross product, one can see that a force created by a magnetic field is going to act perpendicular to the flow of current and to the unit vector that acts radially outward from the wire. This means that a magnetic field will have the following appearance which corresponds to the often used **right hand rule**:
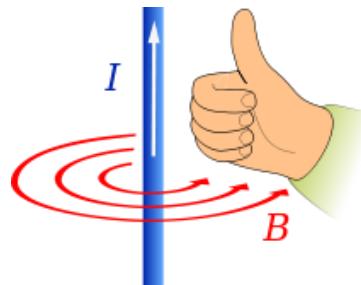

Figure 3. Magnetic Field Lines and the Right Hand Rule

Thus it can be seen that not only does any flow of current generate a magnetic field, but these magnetic field lines circle around the line of current and their magnitude decreased based on the inverse square law.

The magnetic version of Gauss' Law is **Ampere's Law**, which states that that for any closed loop path, the sum of the length elements times the magnetic field in the direction of the length element is equal to the permeability times the electric current enclosed in the loop, or in mathematical terms:

$$\sum B_{\parallel} \Delta \ell = \mu_0 I$$

This is just one way of stating that the magnetic field in space around an electric current is proportional to the electric current, which is intuitive because the electric current is the source of the magnetic field in the first place.

For a coil of wire such as a solenoid with n turns these computations are greatly simplified and the magnitude of the magnetic field is simply:
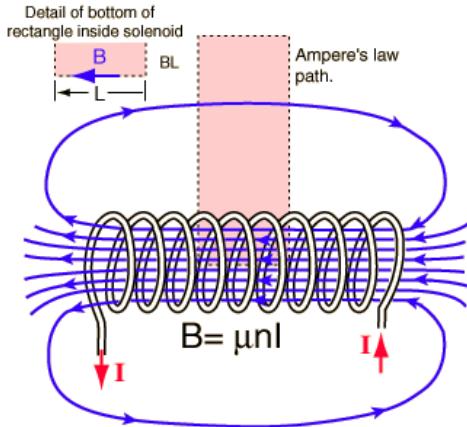

Figure 4. Magnetic Field Lines through a Solenoid

$$B = \mu_0 nI$$

A magnet is any material with a permanent **magnetic dipole** (created from the movement of electrons in the atoms). By convention, these poles of a magnet are termed North and South with the magnetic field lines moving from North to South. Materials capable of exhibiting this behavior are said to be **ferromagnetic** (*ferro* coming from iron, which is one of the main elements exhibiting this phenomena)
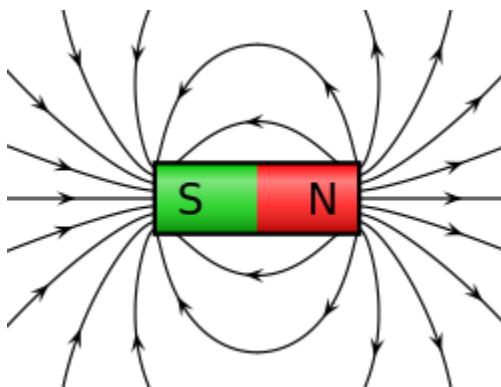
Figure 5. Magnetic Field Lines from a Magnetic Dipole

Based on the definition of voltage given earlier, one can see that a voltage cannot simply be defined only in terms of movement through an electric field, as the potential energy of a test particle can be changed by magnetic field lines as well. This is only applicable when the wire carrying the current is itself moving, so for most common cases it can be neglected. However, there are certain cases where it cannot be neglected and the voltage is changed by a changing magnetic field.

As engineers, the physics behind the relationship between electric and magnetic field lines to the voltages and current we see in practice is oftentimes not particularly useful information. The next sections will go over most of the commonly used electrical components with more of an emphasis on the practical sides of their use.

Resistors

A **resistor** is a passive two-terminal electrical component that implements electrical resistance as a circuit element. They are often branded by the colors on their ceramic component, as shown (Figure 6). These colors indicate the value of the resistor as well as the tolerance on that nominal value and can be read through the use of a color wheel as found here assuming a multimeter is not available.

Figure 6. Typical Axial Lead Resistor

To place a mechanical analogy onto how resistors work in regards to the flow of electrons, consider the following image showing a hydraulic pipe (Figure 7).
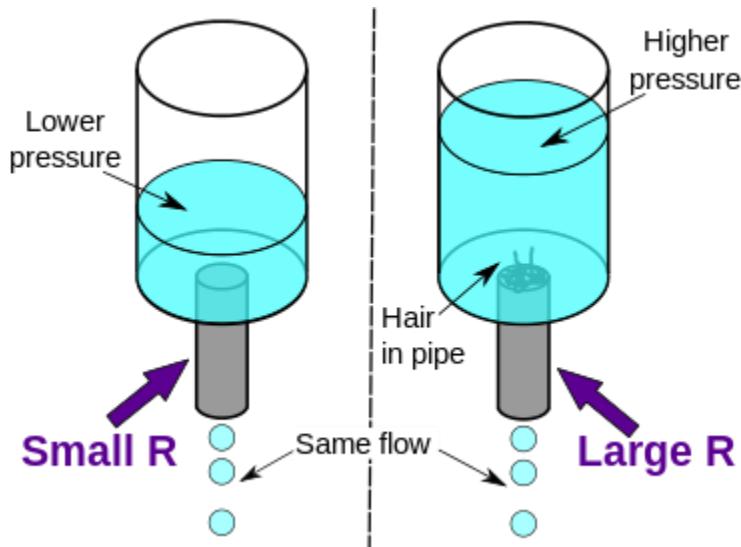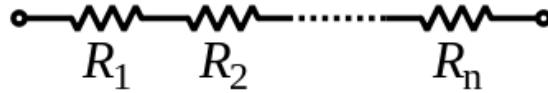
Figure 7. Hydraulic Analogy to Ohm's Law

In this example, the hair in the second pipe creates a larger resistance to the flow, which means that higher water pressure (voltage) is required (in the form of more water in the tank) in order to achieve the same flow of water (electric current).

Following through with Ohm's Law, one can analyze the effects of placing resistors in series with one another:
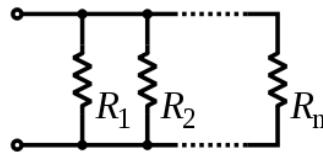

Because the same current passes through each resistor, an equivalent resistance for a single resistor that replaces all of these series resistors can easily be found.

$$\Delta V = IR_{eq} = IR_1 + IR_2 + \cdots + IR_n$$

$$R_{eq} = R_1 + R_2 + \cdots + R_n$$

Similarly, if resistors were placed in parallel with one another in the following orientation:

the effects can also be computed by acknowledging that the voltage drop is the same across each resistor.

$$I = \frac{\Delta V}{R_{eq}} = \frac{\Delta V}{R_1} + \frac{\Delta V}{R_2} + \cdots + \frac{\Delta V}{R_n}$$

$$R_{eq} = \left(\frac{1}{R_1} + \frac{1}{R_2} + \cdots + \frac{1}{R_n} \right)^{-1}$$

Every material has a material property known as its **electrical resistivity** (denoted here as ρ). This property is intrinsic, meaning it doesn't change based on the amount of the material present. It has base units of Ωm. From this material property, the electrical resistance of any conduit is found from the following relationship where A is the cross-sectional area of the conduit and L is the length of the conduit:

$$R = \frac{\rho L}{A}$$

This states that as the cross-sectional area goes up, the electrical resistance decreases and as the length goes up, the electrical resistance increases. For this reason, when making circuits, long stretches of wire are typically a bad thing because as Ohm's Law indicates, it creates a voltage drop across the wires.

While it is true that resistivity is a material property, it is a material property that varies with temperature. Oftentimes this relationship is expressed as a linear relationship so that:

$$\rho = \rho_{ref} + \alpha \Delta T$$

ρ_{ref} is the electric resistivity at a reference temperature, typically room temperature or 20 degrees Celsius, and α is the temperature coefficient of resistivity.

As a point of nomenclature, note that a materials' **conductivity** is simply the inverse of its resistivity.

A table of the electrical resistivity of common materials is given below:

Table 2. Electrical Resistivities of Common Materials

Material	Resistivity ρ (ohm m)		Temperature coefficient α per degree C	Conductivity σ x 10^7 /Ωm
Silver	1.59	x10^{-8}	.0038	6.29
Copper	1.68	x10^{-8}	.00386	5.95
Copper, annealed	1.72	x10^{-8}	.00393	5.81
Aluminum	2.65	x10^{-8}	.00429	3.77
Tungsten	5.6	x10^{-8}	.0045	1.79
Iron	9.71	x10^{-8}	.00651	1.03
Platinum	10.6	x10^{-8}	.003927	0.943
Manganin	48.2	x10^{-8}	.000002	0.207
Lead	22	x10^{-8}	...	0.45
Mercury	98	x10^{-8}	.0009	0.10
Nichrome (Ni,Fe,Cr alloy)	100	x10^{-8}	.0004	0.10
Constantan	49	x10^{-8}	...	0.20
Carbon* (graphite)	3-60	x10^{-5}	-.0005	...
Germanium*	1-500	x10^{-3}	-.05	...
Silicon*	0.1-60	...	-.07	...
Glass	1-10000	x10^{9}	...	...
Quartz (fused)	7.5	x10^{17}	...	...
Hard rubber	1-100	x10^{13}	...	...

* the resistivity of these materials (which are called **semiconductors**) is heavily dependent on the presence of impurities in the material (these impurities are called **dopants**).

Materials with a very low resistance, typically metals, are called **conductors**, and materials with a very high resistance are called **insulators**. Metals are able to perform as conductors because on an atomic level, they share their outer electrons very freely with neighboring metal atoms in what is termed a *metallic bond* (some have heard this termed the "sea of electrons"). Insulators are much more unlikely to lose electrons and as such are poor conductors of electric current. Materials that are somewhere in the middle of these two tend to be called **semiconductors**. A common example of a semiconductor is silicon, which is widely used in the manufacture of electronic chip packages. Other common semiconductors are Germanium and Gallium-Aresenide.

Capacitors

A **capacitor** is a passive two-terminal electrical component that stores electrical energy in an electric field. They come in many different forms, some of which are shown below (Figure 8).
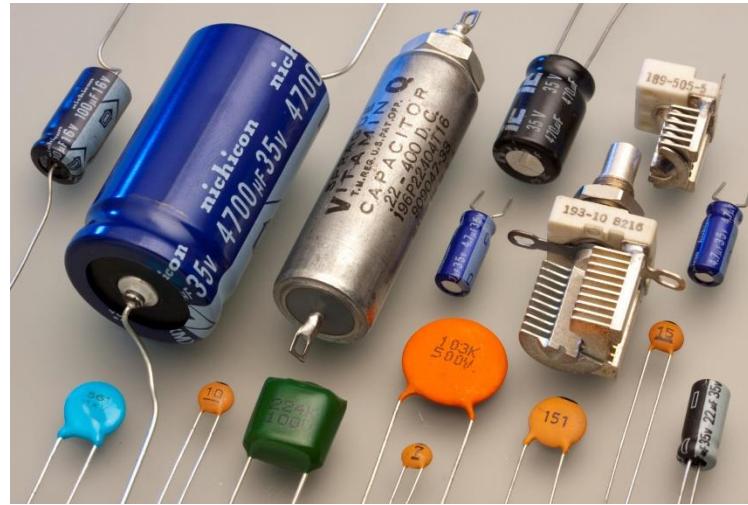

Figure 8. Variety of Different Sized Capacitors

While all things have some amount of capacitance (that is the measure of how well electrical energy is stored in a capacitor), the ideal capacitor is two parallel plates of conducting material separated by an insulating **dielectric** that prevents current from flowing between the plates, but instead allows current to "pass" through the capacitor by charge moving to accumulate on each plate (Figure 9).
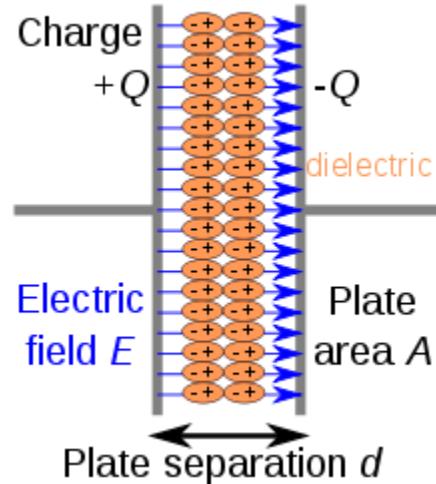
Figure 9. Parallel Plate Capacitor Diagram

The capacitance of any object is given, in Farads (F), by $C = \frac{Q}{V}$, where Q is the accumulated charge and V is the voltage across the gap between the conduits. Typically capacitors are much smaller orders of magnitude than a Farad, often on the order of picoFarads (pF) or nanoFarads (nF).

This definition of capacitance provides the basis of the voltage to current relationship for a capacitor:
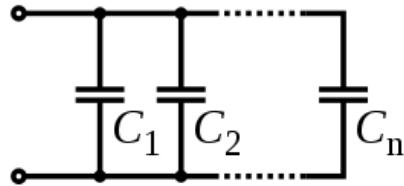
$$V(t) = \frac{Q(t)}{C}$$

$$\frac{dV}{dt} = \frac{I(t)}{C}$$

$$I(t) = C\frac{dV}{dt}$$

This relationship will prove to be fundamental to our understanding of how certain circuits work. Note that if voltage was a sinusoidal function as is the case with **AC (Alternating Current) power** sources, then the current is proportional to the derivative of this function, which ensures that the current waveform **leads** the voltage waveform by exactly 90 degrees.
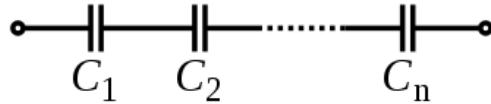
Capacitors wired in parallel can then be found to have an equivalent capacitance given by the fact that the voltage across each capacitor is the same:

$$I_{total} = C_{eq}\frac{dV}{dt} = C_1\frac{dV}{dt} + C_2\frac{dV}{dt} + \cdots + C_n\frac{dV}{dt}$$

$$C_{eq} = C_1 + C_2 + \cdots + C_n$$

And capacitors wired in series giving an equivalent capacitance of:

$$\Delta V = \Delta V_1 + \Delta V_2 + \cdots + \Delta V_n$$

$$\Delta V = \frac{Q_{total}}{C_{eq}} = \frac{Q_1}{C_1} + \frac{Q_2}{C_2} + \cdots + \frac{Q_n}{C_n}$$

Because the capacitors are in series, the same amount of current must pass "through" them. This implies that the charge differential across each capacitor is the same, and the equivalent capacitance can then be given by:

$$C_{eq} = \left(\frac{1}{C_1} + \frac{1}{C_2} + \cdots + \frac{1}{C_n}\right)^{-1}$$

A **supercapacitor** is a capacitor with capacitance values much higher than normal capacitors (typically ten to one hundred times higher and on the order of 1 F) used for the express purpose of energy storage. As such, they provide an alternative to batteries. Supercapacitors do not use the dielectric that is standard for ordinary capacitors, instead opting for physical phenomena known as electrostatic double-layer capacitance and electrochemical pseudocapacitance, neither of which is discussed here. The voltage range for supercapacitors is lower than normal capacitors, and they normally can only be used safely in well-defined ranges that can loosely be stated to lie between 2.1 and 4 V. To achieve higher voltages from supercapacitors, several units will have to be used in series with one another.

Inductors

An **inductor**, also called a coil or reactor, is a passive two-terminal electrical component that stores electrical energy in a magnetic field when electric current is flowing through it. The general idea here is that a moving charge has a magnetic field perpendicular to the motion of the charge. This magnetic field, when passing through some conduit, can have an equivalent magnetic flux computed based on the magnetic field density through the cross sectional area of the conduit. It is a fact of the nature of the universe that things are resistant to change, and as mechanical engineers we are often inclined to view this as the inertia of different objects. Magnetic flux is no different, and when the magnetic flux changes, either through the current levels changing or through the current flow changing direction, a voltage is induced (hence the name inductor) that will generate magnetic flux that attempts to resist the change.

In other words, when the current flowing through an inductor changes, the time-varying magnetic field induces a voltage in the conductor, described by **Faraday's law of induction**. According to **Lenz's law**, the direction of induced electromotive force (e.m.f.) opposes the change in current that created it. **As a result, inductors oppose any changes in current through them.**

An image of various types of inductors is provided below (Figure 10).
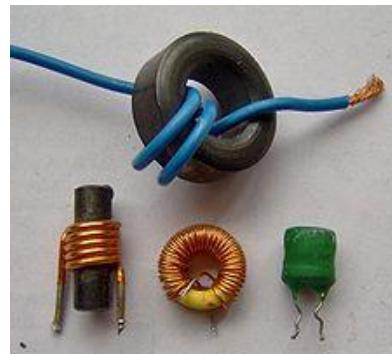
Figure 10. Variety of Inductor Types

An inductor is characterized by its inductance (L), measured in Henris (H) which is given by the following relationship:

$$L = \frac{d\emptyset}{dI}$$
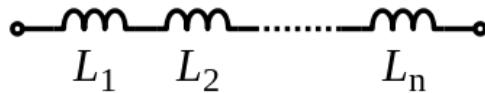
where $\emptyset$ is the magnetic flux and I is the current. Faraday's Law of Induction then states that the induced voltage is given by:

$$V_L = \frac{d\emptyset}{dt} = \frac{d}{dt}(LI) = L\frac{dI}{dt}$$

So inductance is also a measure of the amount of electromotive force (voltage) generated for a given rate of change of current. For AC power sources, this relationship ensures that the current waveform will **lag** the voltage waveform by exactly 90 degrees.

Inductors behave identically to resistors when wired in series or parallel in regards to the construction of an equivalent inductance, as shown below:

Series Inductors
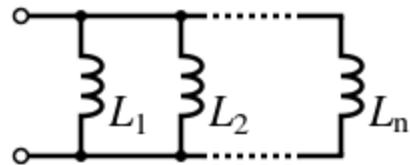
$$\Delta V = \Delta V_1 + \Delta V_2 + \cdots + \Delta V_n$$

$$L_{eq}\frac{dI}{dt} = L_1\frac{dI_1}{dt} + L_2\frac{dI_2}{dt} + \cdots + L_n\frac{dI_n}{dt}$$

Because the current moving through each inductor must be the same:

$$L_{eq} = L_1 + L_2 + \cdots + L_n$$

Parallel Inductors

$$\Delta V = L_{eq}\left(\frac{dI_1}{dt} + \frac{dI_2}{dt} + \cdots + \frac{dI_n}{dt}\right)$$
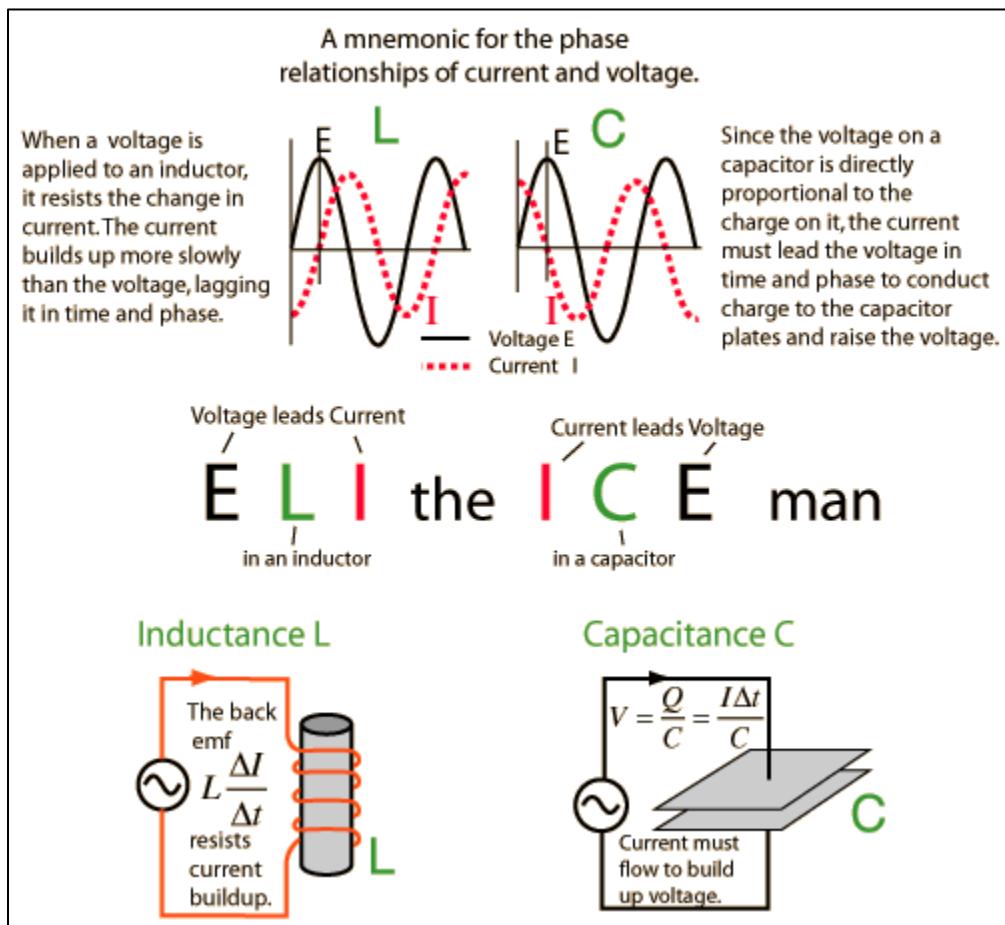
$$\Delta V = L_{eq}\left(\frac{\Delta V_1}{L_1} + \frac{\Delta V_2}{L_2} + \cdots + \frac{\Delta V_n}{L_n}\right)$$

Because the voltage across each inductor must be the same:

$$L_{eq} = \left(\frac{1}{L_1} + \frac{1}{L_2} + \cdots + \frac{1}{L_n}\right)^{-1}$$

A Note on Reactive Power and AC Power Sources

Capacitors and inductors are called **reactive elements** in a circuit, whereas a resistor is called a **resistive element**. For DC circuits, reactive elements are only a factor during the transient response (such as when a circuit initially is connected and begins to charge capacitors or alter the magnetic fields through inductors). For AC circuits, their presence is much more of a factor as mentioned earlier. A simple illustration and mnemonic called "ELI the ICE man" is provided to reemphasize this significance:

A mnemonic for the phase relationships of current and voltage.

When a voltage is applied to an inductor, it resists the change in current. The current builds up more slowly than the voltage, lagging it in time and phase.

Since the voltage on a capacitor is directly proportional to the charge on it, the current must lead the voltage in time and phase to conduct charge to the capacitor plates and raise the voltage.

—— Voltage E
▪▪▪▪ Current I

Voltage leads Current
E L I the **I C E** man
in an inductor   in a capacitor

Inductance L

The back emf $L\dfrac{\Delta I}{\Delta t}$ resists current buildup.

Capacitance C

$V = \dfrac{Q}{C} = \dfrac{I\Delta t}{C}$

Current must flow to build up voltage.

In this way, whereas a resistor carries the property of resistance, (denoted with capital letter R) capacitors and inductors are said to carry **reactance** (denoted with capital letter X). Reactance is also measured in Ohms, much like resistance; but because its effect is not to limit current or voltage, but to shift its phase, it cannot be treated in the same way as resistance. Because of this, the joint effects of resistance and reactance are combined to form the net **impedance** of the circuit. Impedance is typically represented by the capital letter Z and does follow the familiar and linear Ohm's Law such that:

$$\Delta V = ZI$$

Euler's formula which relates Euler's number (e) to sinusoidal functions provides an exceedingly convenient and compact way to express impedance. Recall the following form of Euler's formula proven by taking the Taylor series expansion of all the terms present in the equation:

$$e^{ix} = \cos(x) + i\sin(x)$$

Euler's formula essentially demonstrates exactly how periodic expressions can be mapped onto the complex plane.

Modelling resistance as the real portion of impedance and reactance as the imaginary portion of impedance, both the effects of limiting current flow from resistive elements (which restrict current flow and cause heat to be given off as real work) and limiting current flow from reactive elements (which does no real work but ties up energy in the form of electric and magnetic fields in reactive elements) shows that impedance can be represented as a complex number. Euler's formula demonstrates this is convenient, because the complex plane can be used to compactly represent the periodicity of the impedance. In Cartesian notation, impedance is then represented as:

$$Z = R + iX$$

When viewed in this fashion, the magnitude of the impedance is found as the hypotenuse of the triangle formed by the resistance and the reactance on the real-imaginary plane:

$$Z = \sqrt{R^2 + (X_L - X_C)^2}$$

The reactances are directly proportional to the capacitance and inductance of the components being analyzed such that:

$$X_L = 2\pi f L$$

$$X_C = \frac{1}{2\pi f C}$$

where f is the frequency in Hertz. This leads to the intuitive notion that as the frequency goes up, inductors will block current and capacitors will be more likely to let current pass. Mathematically, one might look at the expression $(X_L - X_C)^2$ and be confused as to why X_L and X_C are not additive.

This follows from the fact that, in polar notation, the reactance of a capacitor is found such that:

$$X_C = \frac{1}{2\pi f C} e^{-i\frac{\pi}{2}}$$

This is a way of stating that the AC voltage across a capacitor lags the AC current by 90 degrees. The application of Euler's formula to this expression yields:

$$X_C = \frac{1}{i2\pi f C}$$

The final trick comes from the nature of imaginary numbers. Note the following:
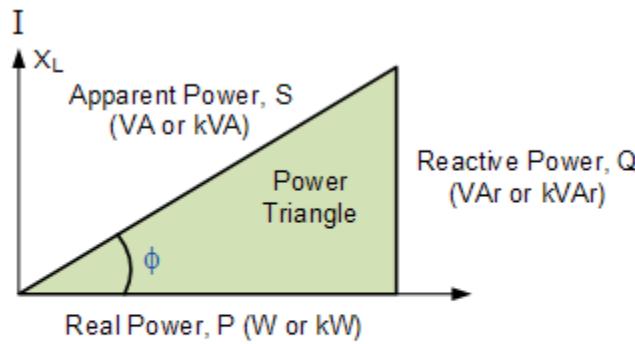
$$\frac{1}{i} = \left(\frac{i}{i}\right)\left(\frac{1}{i}\right) = \frac{i}{i^2} = -i$$

This fact leads to the final form of a capacitors reactance:

$$X_C = -i\frac{1}{2\pi f C}$$

So, when finding the magnitude of impedance, the capacitors reactance is subtracted from the inductors reactance.

When looking at real circuits in which resistors, capacitors, and inductors are all present, the current will end up being out of phase with the voltage by some value that is less than or equal to 90 degrees. Not all of the power being put into the system will be used to do real work, as some of it is tied up in the reactive elements, which leads to the idea of the **power triangle** as shown below.

$$P = VI\cos\phi, \quad S = VI, \quad Q = VI\sin\phi$$
$$VA = \sqrt{P^2 + VAr^2}$$

Figure 11. Power Triangle

Only the real power can be used to do real work, but the apparent power is what must be supplied in order to get the real power out of the circuit. It is worth noting that real power is denoted in Watts, apparent power is denoted in VA (Volt Amperes) and reactive power is denoted in VAr (Volt Amperes reactive).

Solid State Relays

A **relay** is an electrical device, typically incorporating an electromagnet, that is activated by a current or signal in one circuit to open or close another circuit. An example of a mechanical relay exists in the form of the solenoids that work inside of the starter motor of a vehicle to send power from the main battery to the engine, in a way similar to the following image (Figure 12):
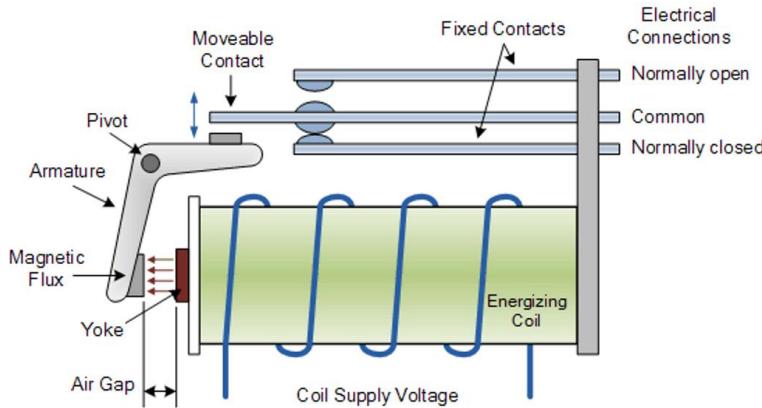

Figure 12. Example Mechanical Relay

A **solid state relay (SSR)** is just what it sounds like; an integrated circuit that acts like a mechanical relay. A solid state relay allows to control high-voltage AC loads from lower voltage DC control circuitry. Solid state relays, have several advantages over mechanical relays. One such advantage is that they can be switched by a much lower voltage and at a much lower current than most mechanical relays. Also, because there's no moving contacts, solid state relays can be switched much faster and for much longer periods without wearing out. They accomplish this by using infrared light as the 'contact,' a solid-state relay is really just an IR LED (covered in next section) and a **phototriac coupler** (a device that transfers the input signal while isolating the input and output) sealed up into a little box. Thanks to the fact that the two sides of the relay are photo-coupled, you can rely on the same type of electrical isolation as in mechanical relays. An example of a kit that includes such a relay is provided by SparkFun here (SparkFun Beefcake Relay Control Kit) which takes in 4-6 V DC from a microcontroller and can output 220 VAC.

Solid state relays can generate a large amount of heat due to large voltages and currents passing through them. Because of this, they are often equipped with large heat sinks, such as the one shown below:

Figure 13. SSR-40 Solid State Relay with Heat Sink

Diodes

A **diode** is a two-terminal electronic device that ideally conducts current with zero resistance in one direction and infinite resistance in the other. A semiconductor diode, the most common type today, is a crystalline piece of semiconductor material with a **p–n junction** connected to two electrical terminals (Figure 14).
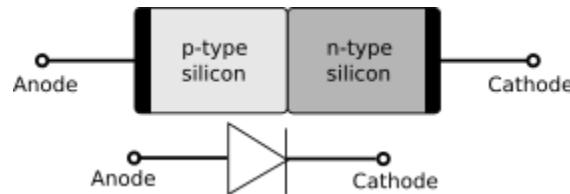

Figure 14. Basic Diode Diagram

A p–n junction is a boundary or interface between two types of semiconductor material, p-type and n-type, inside a single crystal of semiconductor. The "p" (positive) side contains an excess of "holes", while the "n" (negative) side contains an excess of electrons. The basic idea of a diode is that current can flow from the p-type to the n-type (from the **anode** to the **cathode**) but not the other way around. This is because the actual flow of electrons is opposite to the direction of the current (by convention) and the electrons will move from the material with excess electrons to the material with an electron deficiency, causing the diode to bridge between the cathode and anode and allowing current to pass through. However, it takes a very high voltage to induce current in the opposite direction (from the cathode to the anode), because the "electron holes" do not have any excess electrons that want to fill them. A Schottky diode is a very commonly used diode and has the following appearance, with the cathode being the end with the vertical gray bar (Figure 15).

Figure 15. Schottky Diode

A common mechanical analogy for a diode is a one-way spring loaded check valve where the electric current is represented as water and the mechanism of the diode is represented by the valve itself.

One common example of the usage of a diode is in an **LED (Light Emitting Diode)**, shown below (Figure 16).
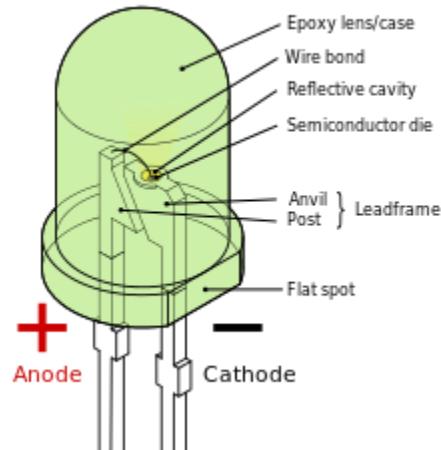

Figure 16. Basic LED Schematic

The actual behavior of a diode is given by a chart detailing how much current passes through it based on the applied voltage (Figure 17).
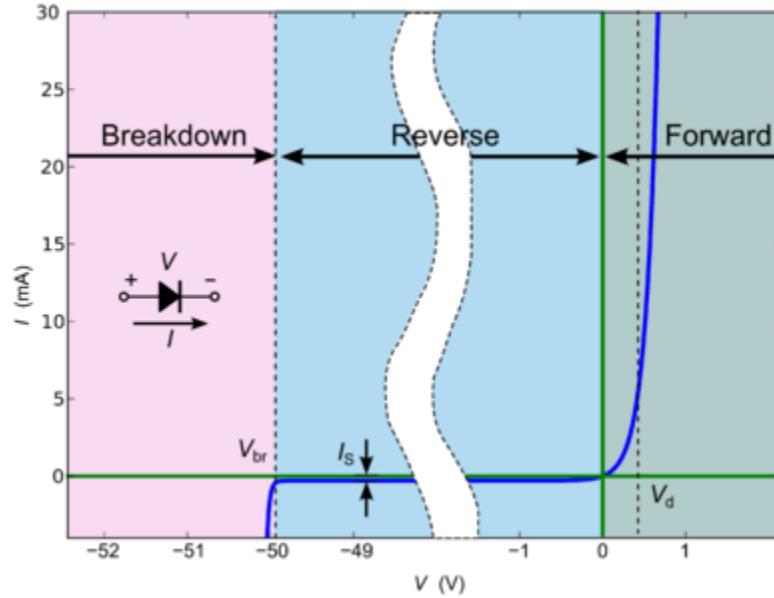

Figure 17. Typical Diode Current versus Voltage Relationship

Note that to get current that flows in the opposite direction than the diode intends, a very high voltage is required (known as the **reverse breakdown voltage**). Before this point, any current flow in the opposite direction is miniscule, and after this point current flows freely. The **forward breakdown voltage** is much, much lower.

The **ideal diode** has no reverse breakdown voltage and has behavior outlined by the nonlinear, exponential relationship given by:

$$I_D = I_S \left[e^{\frac{qV_D}{nkT}} - 1 \right]$$

where:

$I_D \equiv$ current through the diode (A)

$V_D \equiv$ voltage across the diode (V)

$I_S \equiv$ rated saturation current of the diode (A)

$q \equiv$ electron charge (1.6×10^{-19} Coloumb)

$q \equiv$ Boltzmann Constant $\left(1.38 \times 10^{-23} \dfrac{\text{Joules}}{\text{Kelvin}} \right)$

$n \equiv$ non $-$ ideality factor (≈ 1)

$T \equiv$ diode temperature (K)

There exists a special type of diode, known as the **Zener diode**. A Zener diode allows current to flow from its anode to its cathode like a normal semiconductor diode, but it also permits current to flow in the reverse direction when its **Zener voltage** is reached. Zener diodes have a highly doped p-n junction. Normal diodes will also break down with a reverse voltage but the voltage and sharpness of the knee are not as well defined as for a Zener diode. Also normal diodes are not designed to operate in the breakdown region, but Zener diodes can reliably operate in this region. A Zener diode is represented by the following symbol:

The relationship between current and voltage for a Zener diode may look like (Figure 18):
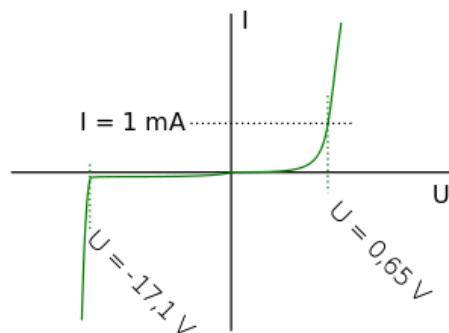
Figure 18. Zener Diode Current versus Voltage Relationship

Note how much lower the reverse breakdown voltage is. This value happens to be incredibly stable for Zener diodes, and so this fact is often taken advantage of to achieve a stable, reliable, regulated voltage to within 1% tolerance. In this way, a Zener diode (in series with a shunt resistor to limit current) can be used a simple **linear voltage regulator**.

Bipolar Junction Transistors (BJT)

A **bipolar junction transistor** (bipolar transistor or BJT) is a type of transistor that uses both electron and hole charge carriers. In contrast, unipolar transistors, such as field-effect transistors, only use one kind of charge carrier. For their operation, BJTs use two junctions between two semiconductor types, n-type and p-type. A schematic outlining their mechanism of action is shown below (Figure 19):
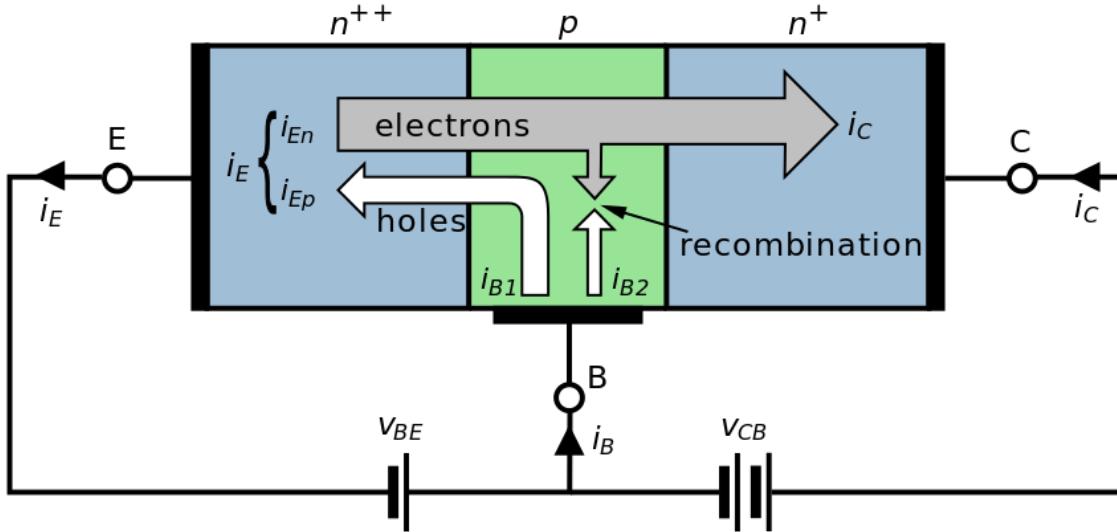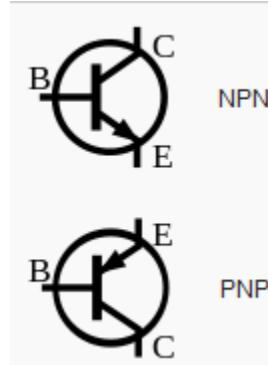

Figure 19. BJT Mechanism of Action

A BJT has three terminals: the **emitter** (E), the **base** (B), and the **collector** (C). In the shown orientation, the transistor is doped in a NPN type orientation, but PNP types exist as well (the current will flow in the opposite direction). The chief principle of a BJT is that it acts as a current activated switch. A small amount of current from the base to the emitter will allow for a very large current to be able to pass through from the collector to the emitter. In a circuit diagram, a BJT will typically look like:

A transistor can only pass through a finite amount of current based on the applied voltages to its terminals. These limitations are outlined by detailing the maximum rated power output of a BJT and constructing a diagram similar to the following (Figure 20):
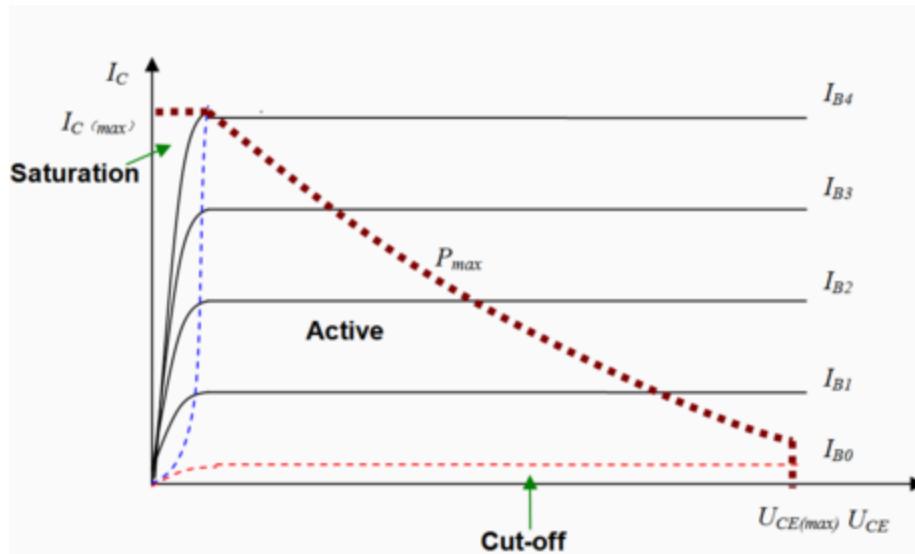
Figure 20. BJT Maximum Current Ratings

Saturation is said to occur when the maximum current is reached through the collector for a given current through the base. Underneath the **cut-off current**, no current will pass through the collector.

An example of what a BJT looks like is given below (Figure 21):

Figure 21. Typical BJT Appearance

Metal Oxide Semiconductive Field Effect Transistors (MOSFET)

A **MOSFET** is a three terminal device with the three terminals being the **Gate** (G), the **Source** (S), and the **Drain** (D). There is a fourth terminal, the **Body** (B), but this is almost always shorted and connected with the Drain inside of the MOSFET (Figure 22).
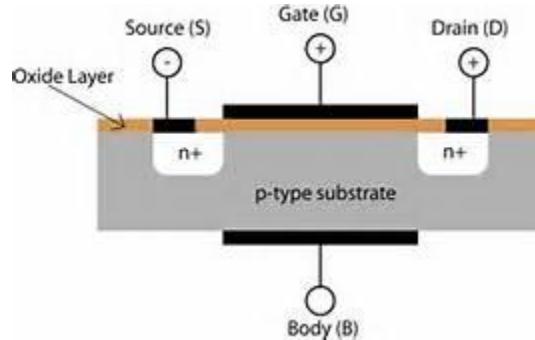

Figure 22. MOSFET Diagram

Essentially, whereas the BJT is a current activated switch, the MOSFET is a voltage activated switch. When a voltage is applied between the Gate and Body, the electrical field that is generated can alter the charge distribution inside of the substrate and allow current to pass from the Source to the Drain. This is why these transistors are called Field Effect Transistors. A small voltage applied to the Gate will allow large amounts of current to pass from the Source to the Drain.

A typical MOSFET might have the following appearance (Figure 23):
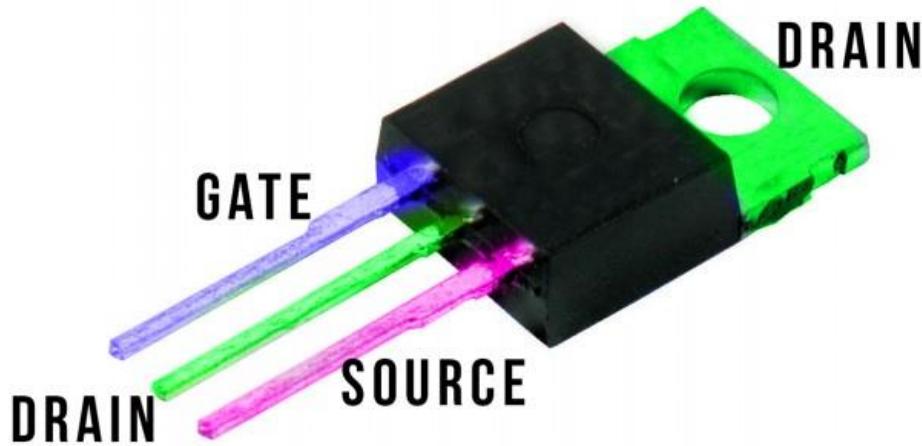
Figure 23. Typical MOSFET Appearance

These transistors are more robust and can typically handle more current than a BJT. A MOSFET can be operated in **enhancement mode**, as described above, or, less commonly, it can use what is known as **depletion mode**, where the application of the electric field actually causes the ability to transmit charge from Source to Drain to be drastically reduced.

A Note on Semiconductor Doping

Diodes and transistors all make use of P type and N type semiconductors to selectively pass current as the user supplies a small current or voltage to initiate it. The most common semiconductor in use is silicon, which has a lattice structure as shown:
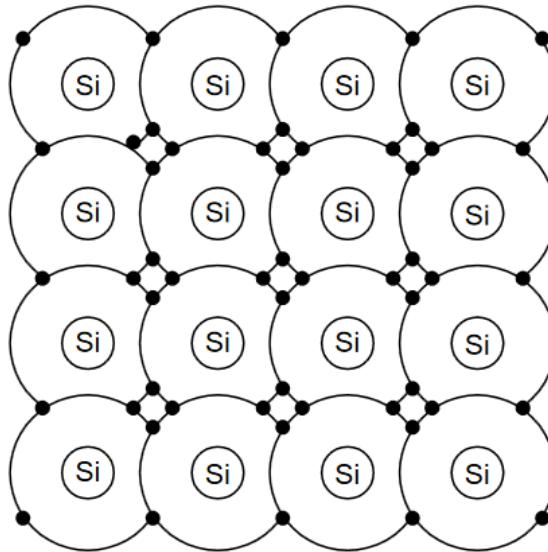

Figure 24. Silicon Lattice Structure

It is worth noting that silicon has four **valence electrons** (electrons in the outer orbitals that are more freely shared with surrounding atoms). The Resistivity of Silicon is 1.56 X 10 Ωcm, and is due to the presence of one electron out of 2×10^{13} that has sufficient energy (1.1 electron Volts [eV]) to jump from the valence band to the conduction band. Silicon forms what is called *covalent bonds* with itself (meaning the electrons are shared equally between the atoms), in contrast to *ionic bonds* (where electrons are not shared equally between the atoms).

The semiconducting properties of materials such as silicon can be modified by changing the atomic structure of single crystals of these materials. If one or more of the silicon atoms in the lattice is replaced with an impurity atom, or **dopant**, (e.g. boron or phosphorus), the conductivity of the modified structure is significantly changed. These impurities come in two primary forms:

- **N-Type Semiconductors:** Elements from the VA Column of the Periodic Table with five valence electrons in their outer shell such as Phosphorus, Arsenic, and Antimony.
  - ❖ If Phosphorus with its five valence electrons is introduced into the silicon lattice, the fifth electron is not used to complete covalent bonding. This extra electron, an extrinsic charge, is relatively free to carry current. Hence, the resistivity of phosphorus-doped silicon is lower than that of pure silicon. Silicon with dopant elements from the VA Column of the Periodic Table are Classified as N-Type Semiconductors, because they contain extra electrons That are negative charge carriers

- **P-Type Semiconductors**: Elements from the IIIA Column of the Periodic Table with three valence electrons in their outer shell such as Boron, Aluminum, and Gallium.
  - ❖ If silicon atoms in the crystal lattice are replaced with a dopant Atom with three valence electrons in its outer shell such as boron, then covalent bonding occurs between silicon and boron but the outer shell is not filled so that a single vacancy or "hole" exists. This hole is an accepter of electrons, and it acts as a positive charge carrier. The semiconductors with dopant elements selected from Column IIIA in the Periodic Table are classified as P-Type because of the extra positive charges, which are carried by the holes, are capable of Moving through the atomic lattice.

The resistivity of a semiconductor is inversely proportional the number of charge carriers are present in the silicon lattice. Ideally, the relationship is expressed as the rather esoteric function:

$$\rho = \frac{1}{eN\eta}$$

where e is the charge of the charge carrier, N is the number of charge carries, and η is the "mobility" of the charge carrier. A plot illustrating just how linear this relationship can actually be is provided below:
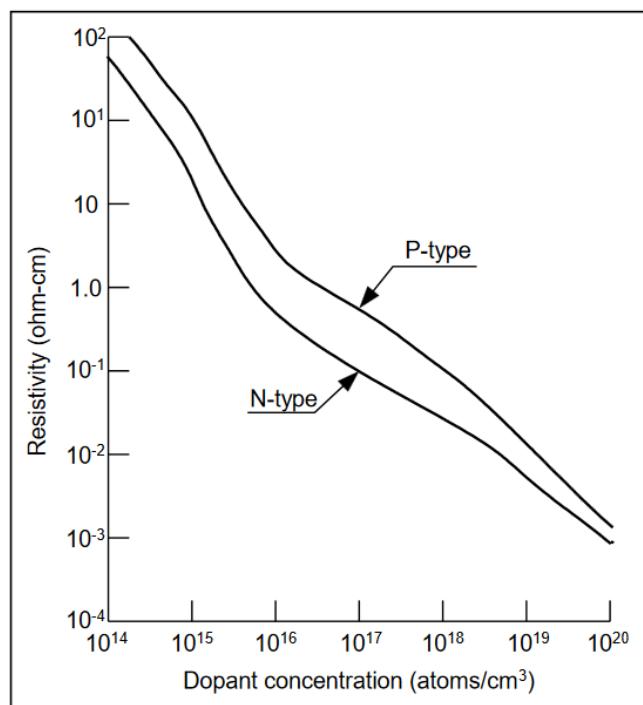

Figure 25. Semiconductor Resistivity versus Dopant Concentration

Unsurprisingly, the act of putting these impurities into the silicon lattice is called **doping**. Doping of semiconductors is a fundamental part of the manufacture of ICs from silicon wafers and is mostly done through selective diffusion of the impurities into the silicon.

Operational Amplifiers

An **operational amplifier** (or op-amp) is a complex system of electronic components such as resistors, capacitors, inductors, diodes, and transistors that works to create a device that outputs a voltage that works to make the differential voltage inputs equal to one another. Typically they are drawn as follows:
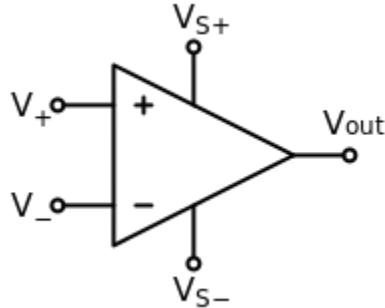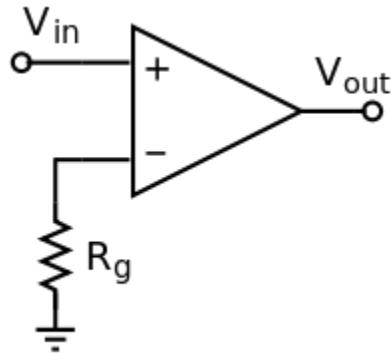

Figure 26. Basic Representation of an Op-Amp

An ideal operational amplifier has the following characteristics:

    a. Infinite Gain
    b. Infinite Input Impudence
    c. Zero Output Impedance
    d. No noise during operation
    e. Outputs any voltage between the two supply rails

In reality, operational amplifiers can only output voltage to within one or two volts of the supply rails, and they do add some noise to the operation. While the gain is not infinite, it is typically very large.

The simplest operational amplifier is the **comparator:**

This op-amp will compare the input voltage to ground, and if the input voltage is greater than ground, the output voltage will be positive, and if it is less than ground, the output voltage will be negative. This is the fundamental building block for how a microcontroller can achieve **Analog to Digital Conversion.** By sampling a voltage and using a comparator, the microcontroller can hone in to some degree of precision to what the sampled voltage was. There will be more on Analog to Digital Conversion later. Operational amplifiers can be used as analog circuits to do real math, such as addition, subtraction, multiplication, division, integration, and differentiation. More information on this is found here.

Batteries

An electric battery is a device consisting of one or more electrochemical cells with external connections provided to power electrical devices. The voltage in each cell of a battery is dependent upon the chemistry of the battery. A few very common examples are provided:

Table 3. Battery Types and Nominal Voltages

Battery Type	Nominal Voltage
Alkaline (AA, AAA, …)	1.5 V
Nickel Cadmium	1.5 V
Lithium – X	3.7 V

A **nominal voltage** is close to the average voltage of the cell over the lifetime of the battery. The **cell capacity** is essentially how "big" the battery is or how many cells are in the battery in parallel. By placing cells in a battery in series, the voltages of each cell are additive. By placing cells in a battery in parallel, the energy capacity and maximum current draw (measured in Amp hours [Ah]) increases. The performance of a battery is characterized by its discharge diagram, which shows battery voltage compared to stored energy. An example discharge diagram is shown below:
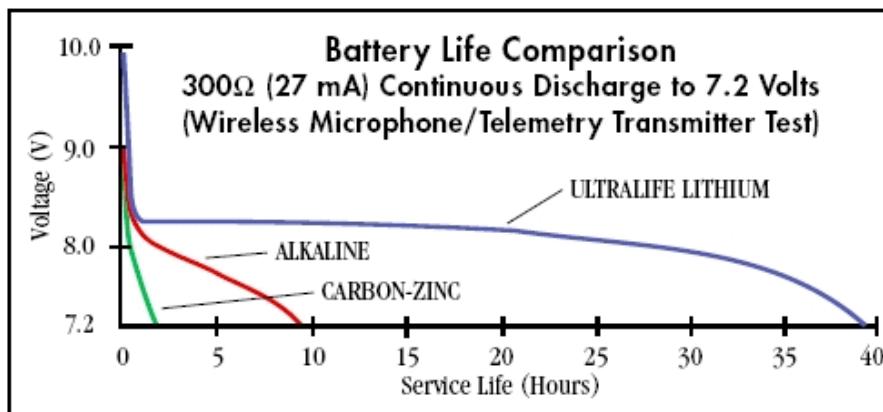

Figure 27. Discharge Diagram of Various Batteries

As a battery nears the end of its life, the voltage falls of dramatically. Also worth noting is that as current draw increases, the voltage decreases faster for a given amount of energy in a battery. For example, an alkaline 9V battery is actually just six 1.5 V volt cells wired in series. This means that the maximum current draw for a 9V battery is fairly low (on the order of 200 mA) and such a battery would never be used for high current applications such as driving a motor.

A battery's **C rating** gives the ratio of continuous current possible (mA) to the cell capacity (mAh). For example, a 1 Ah battery with a 2C charge and 8C discharge rating can charge with a maximum current of 2 A and discharge with a maximum current of 8 A. The important of this is that a battery will likely catch fire if these ratings are exceeded. To create a battery charger, the current must be limited (preferably far below the C rating).

Some general notes on batteries:

- the positive end of the electromechanical cell is referred to as the anode and the negative end is the cathode.
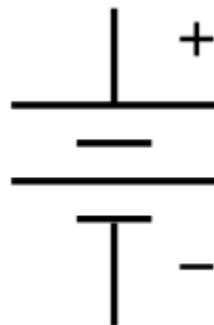
- Lead acid batteries are used in cars and are moderately expensive with a cell voltage of 2.1 V.
- Lithium ion batteries have a cell voltage of 3.6 V, but are very expensive and an **extreme fire hazard**.

Batteries come in all different types and sizes. A few different types can be shown as seen below:

**Figure 28.** Various Types of Battery Types and Sizes
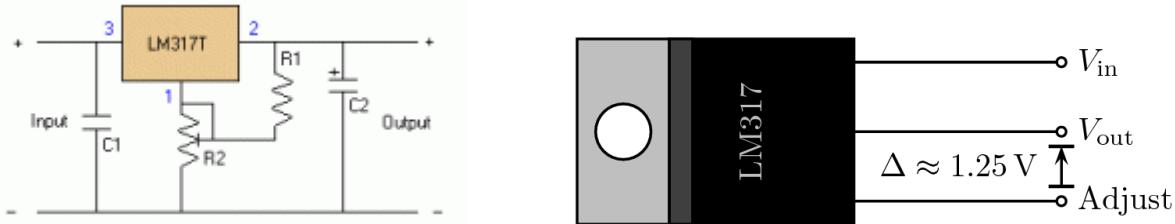
In a circuit, batteries are typically represented with the following symbol:

NOTE: In order to power a microcontroller safely with a battery and read data via its USB port, the USB power must be cut from the power supply of the microcontroller. This can be accomplished on a Teensy 3.2 by cutting the copper trace indicated on the bottom side of the board. These pads can be reconnected at any time with solder. Through the use of two diodes, it is also possible to enable powering of the Teensy with a battery or USB both at any time, with the higher voltage between the two being used if the two are plugged in at the same time.

Voltage Regulators

Through the use of Zener diodes to maintain a steady voltage, operational amplifiers to buffer the circuit, and resistors to adjust the voltage on one leg of the Zener diode, a stable voltage regulator can be crafted. This helps solve the problem of the unreliable voltages given off by batteries. Some of the most readily available voltage regulators are the LM317xx series (where xx corresponds to the regulated voltage output). For adjustable voltage regulators, the grounded voltage of the Zener diode can be adjusted by changing the resistors that are fed into the ADJ pin of the voltage regulator, based on the data sheet of the regulator.

The two capacitors are there to maintain constant power supply to the voltage regulator and the two resistors enable the output voltage to be adjusted (for those that offer adjustable outputs).

Voltage regulators tend to dissipate a lot of energy heat. Conservation of energy reveals:

$$P_{heat} = P_{in} - P_{out} = I(V_{in} - V_{out})$$

For a load attached the output, R_L, the current passing through the regulator will be:

$$I = \frac{V_{out}}{R_L}$$

So, the power lost to heating is given by:

$$P_{heat} = \frac{V_{out}(V_{in} - V_{out})}{R_L}$$

This means that lower resistances for the load create more losses due to heat (more current equates to higher heat losses in the regulator). The overall efficiency of a such a device would be given by:

$$\eta = \frac{P_{out}}{P_{in}} = \frac{V_{out}}{V_{in}} \approx 20\%$$

Often voltage regulators are the cause for the need of a heat sink to avoid overheating of electronics because they are terribly inefficient devices, but useful ones. There exist alternatives to raise the efficiency in the form of **switching regulators** but the increased efficiency tends to not justify their increased cost.

Through the addition of a shunt resistor immediately out of the regulator's output that leads into both the load and the GND input of the regulator, the ground reference of the Zener diode is read as the output of the shunt resistor. This keeps the current passing through the shunt resistor as constant, and turns a linear voltage regulator into a **current regulator**.

16.   Kirchoff's Laws and Basic Useful Circuits

Kirchoff's Laws are two important and fundamental laws to analyzing any electrical circuit. For electric current to flow, a voltage differential must be supplied to a conduit in a closed loop. If there is not a closed loop, any current flow will only occur for microseconds (if at all) while the conduit comes to an equilibrium. To analyze these circuits, some basic rules that emerge from conservation laws:

### f.   Kirchoff's Current Law (KCL)
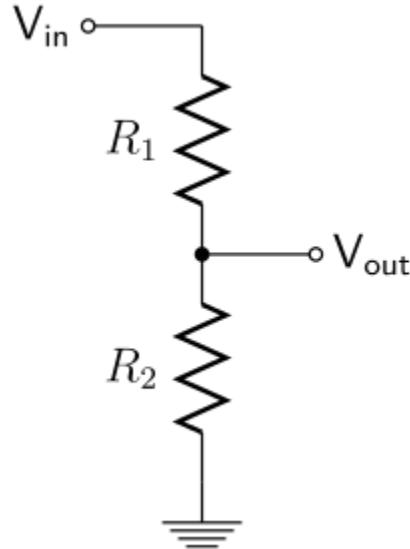
The sum of all current entering a node at any position in an electric circuit is zero.

g.   Kirchoff's Voltage Law (KVL)

The sum of all voltage drops in a complete circuit is zero.

Utilizing these laws and knowledge of the constitutive equations that make up the basic electronic components, circuits can now be analyzed.

Simple Voltage Divider

The current passing through the circuit is given by $I = \frac{V_{in}}{R_1 + R_2}$ which means that:
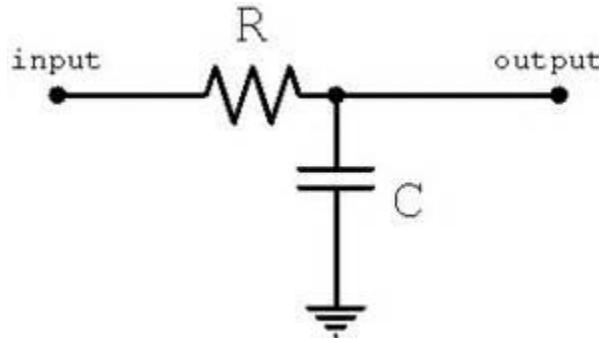
$$V_{out} = V_{in} - IR_1$$

$$V_{out} = V_{in} \left(\frac{R_2}{R_1 + R_2} \right)$$

It is worth noting that if the two resistors are the same, then the output voltage is exactly half of the input voltage. However, if the output voltage is wired into anything with a resistance of its own, that resistance will be considered to be in parallel with R_2 and the voltage divider will no longer function as predicted.

Analog First Order Low Pass Filter

A **filter** is something that removes unwanted portions of a signal while retaining the important information contained a signal. A common example of this would be electrical noise, which typically has high frequency components that skew the signal of interest. A low pass filter, put simply, helps attenuate (reduce the impact of) high frequency portions of signals.

By summing the currents at the output node (to the right of the resistive element) and invoking Kirchoff's Current Law (KCL), find that:

$$\sum I_{output} = I_R - I_C = 0$$

This shows that the current passing through the resistor must equal the current passing through the capacitor. From the constitutive equations developed earlier for both a resistor and a capacitor, find that:

$$I_R = \frac{V_{input} - V_{output}}{R}$$

$$I_C = C \frac{dV_{output}}{dt}$$

$$\frac{V_{input} - V_{output}}{R} = C \frac{dV_{output}}{dt}$$

Simplifying yields:

$$V_{input}(t) = RC \frac{dV_{output}}{dt} + V_{output}(t)$$

This is a first order differential equation with an eigenvalue of $\frac{1}{RC}$ (this means the response is that of exponential decay). In other words, the **homogenous** solution to this problem takes the form:

$$V_{output}(t) = A e^{-1/RC \ t}$$

where *A* is a constant that is determined by the initial conditions of the problem, such that:

$$V_{output}(t) = V_{output}(0) e^{-1/RC \ t}$$

The low pass filter, as specified above, is important because is able to attenuate signals that lie above a certain cutoff frequency. To see this effect, the dynamics of the problem must be transformed into the frequency domain via the use of the Laplace transform:

$$V_{input}(s) = V_{output}(s)(RCs + 1)$$

The transfer function of this system is then given by:

$$\frac{V_{output}(s)}{V_{input}(s)} = \frac{1}{RCs + 1}$$

Note that the variable s corresponds with, by definition, the relation $s \equiv j\omega$, where ω is the frequency of oscillation in the system. Therefore, the transfer function can be represented as:

$$\frac{V_{output}(j\omega)}{V_{input}(j\omega)} = \frac{1}{jRC\omega + 1}$$

To see the effects of how this system **gain** (ratio of output signal to input signal) changes with frequency, the magnitude of the transfer function must be taken:

$$\left|\frac{V_{output}(j\omega)}{V_{input}(j\omega)}\right| = \left|\frac{1}{jRC\omega + 1}\right| = \frac{1}{\sqrt{(RC\omega)^2 + (1)^2}}$$
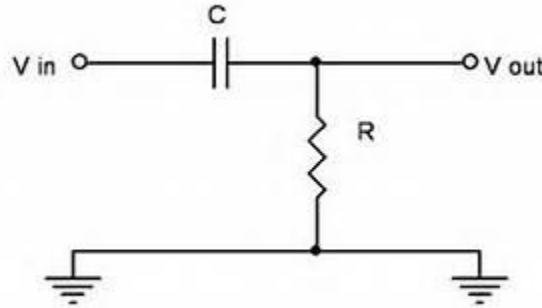
This is true because the magnitude of a complex number is given by the Pythagorean theorem applied to the complex plane. Note some interesting features of this magnitude is that as frequencies go higher, the signal gain moves towards 0, and as frequencies go towards 0, the signal gain moves towards 1 (the DC gain). This is characteristic of a low pass filter.The cutoff frequency (where the signal has been attenuated to -3 dB, or 70.7% of the DC gain) is given in this case by $\frac{1}{RC}$ (radians per second). An analysis of the phase response of a low pass filter would show that it also causes a **phase lag** in the output signal (45 degrees exactly at the cutoff frequency).

Physically, this circuit acts in way for two specific reasons:

- At low frequencies, there is plenty of time for the capacitor to charge up to practically the same voltage as the input voltage.
- At high frequencies, the capacitor only has time to charge up a small amount before the input switches direction. The output goes up and down only a small fraction of the amount the input goes up and down. At double the frequency, there's only time for it to charge up half the amount.

When a signal is attenuated through a low pass filter, it should be noted that this simply means that it begins to resemble a flat line at the average of the input signal. This is the fundamental idea behind **Digital to Analog Conversion (DAC)**. Because nature itself acts as a low pass filter, extremely high frequency digital signals will be filtered to resemble an analog signal.

Analog First Order High Pass Filter

By summing the currents at the output node (to the right of the capacitive element) and invoking Kirchoff's Current Law (KCL), find that:

$$\sum I_{output} = I_C - I_R = 0$$

This shows that the current passing through the resistor must equal the current passing through the capacitor. From the constitutive equations developed earlier for both a resistor and a capacitor, find that:

$$I_R = \frac{V_{out}}{R}$$

$$I_C = C\frac{d(V_{in} - V_{out})}{dt}$$

$$\frac{V_{out}}{R} = C\frac{dV_{in}}{dt} - C\frac{dV_{out}}{dt}$$

Moving into the Laplace domain once again:

$$V_{out}(s) = RCsV_{in}(s) - RCsV_{out}(s)$$

The transfer function of this system is then given by:

$$\frac{V_{out}(s)}{V_{in}(s)} = \frac{RCs}{RCs + 1}$$

Note that the variable s corresponds with, by definition, the relation $s \equiv j\omega$, where ω is the frequency of oscillation in the system. Therefore, the transfer function can be represented as:

$$\frac{V_{output}(j\omega)}{V_{input}(j\omega)} = \frac{jRC\omega}{jRC\omega + 1}$$

To see the effects of how this system **gain** (ratio of output signal to input signal) changes with frequency, the magnitude of the transfer function must be taken:
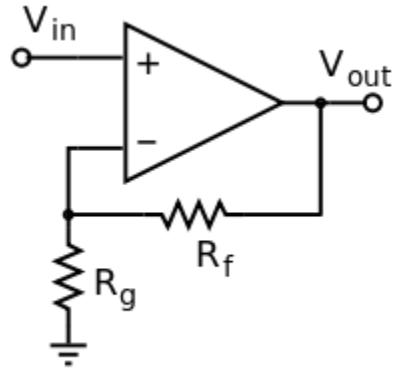
$$\left|\frac{V_{output}(j\omega)}{V_{input}(j\omega)}\right| = \left|\frac{jRC\omega}{jRC\omega + 1}\right| = \frac{RC\omega}{\sqrt{(RC\omega)^2 + (1)^2}}$$

Notice that $\lim\limits_{\omega\to 0}\left|\frac{V_{output}(j\omega)}{V_{input}(j\omega)}\right| = 0$ and $\lim\limits_{\omega\to\infty}\left|\frac{V_{output}(j\omega)}{V_{input}(j\omega)}\right| = 1,$ which is characteristic of a high pass filter. High pass filters add a **phase lead** to the output signal.

Worth noting that if a low pass filter and a high pass filter are connected, a **band pass filter** can be created that only passes a certain band of frequencies. This is also called a "lead-lag compensator".

Amplifier/Follower

By using an operational amplifier connected in a feedback loop with the inverted (negative) input as shown, the op-amp can be used to amplify the signal:

Because no current can pass through either input of the operational amplifier, and both inputs have the same voltage by definition of how an op-amp works, the current output is given by:

$$I_{out} = \frac{V_{in}}{R_g}$$
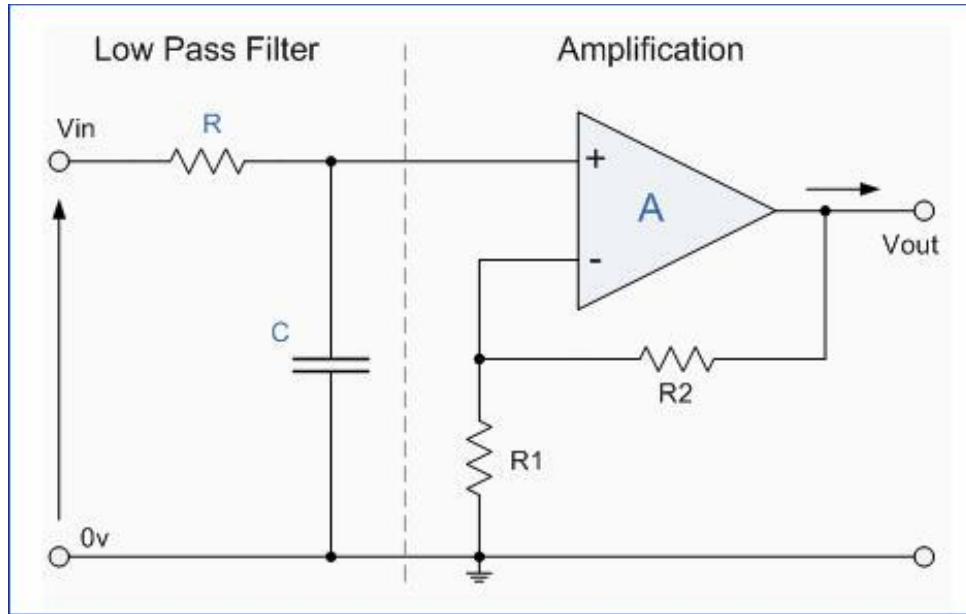
Kirchoff's Voltage Law then allows for the output voltage to be solved for:

$$V_{out} = V_{in} + IR_f = V_{in} + \left(\frac{V_{in}}{R_g}\right)R_f$$

$$V_{out} = V_{in} + IR_f = V_{in}\left(1 + \frac{R_f}{R_g}\right)$$

The signal is amplified by a factor based on the values of the two resistors. It is also worth noting that if $R_f = 0$, then the output voltage would simply equal the input voltage (R_g could even be removed from the circuit and achieve this same result). This has the effect of buffering whatever circuit gives out V_{in} from the output of the operational amplifier and creates what is known as a **follower**. Using a follower with the

voltage divider enables it to work regardless of what types of resistance the output voltage of the divider is exposed to on its leg of the circuit.
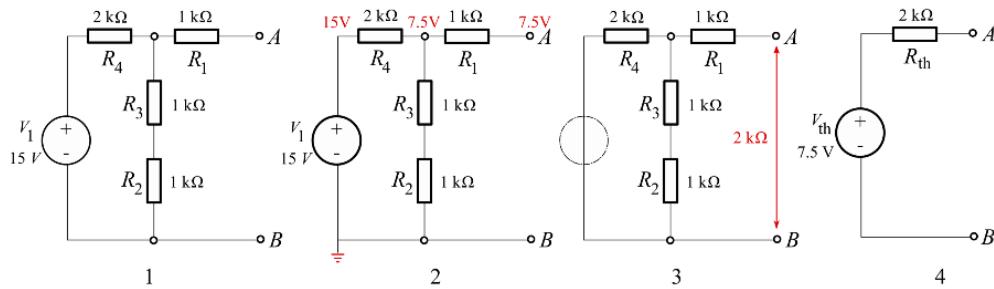

NOTE: It is left to the reader to research or show themselves the types of filters that can be created by combining elements such as capacitors, inductors, and resistors in series and in parallel (for example, an RL circuit in series creates a high pass filter when measuring voltage and a low-pass filter when measuring current). Generally, the order of the filter created is equal to the number of reactive elements (energy storing elements in the form of capacitors and inductors) that are present in the circuit. A second order filter will have attenuate signals twice as fast as a first order signal, but may have some frequency in which the gain is greater than unity (resonance).

It should also be noted that one special RLC circuit with a diode can be used to create a **switching regulator**, which is a voltage regulator that turns "on" and "off" so as to increase efficiency, but at the expense of increased cost of the regulator itself.

Note on Thevenin's Equivalent Circuits

A fundamental theorem exists in electrical circuit analysis known as **Thévenin's theorem** which holds that any electric network can be simplified into a power source, a Thévenin's equivalent resistor (R_{th}), and the load resistor which represents the system of interest. A simple example is shown below:

17.   Analog to Digital Conversion (ADC)

An **Analog to Digital Converter (ADC)** is a very useful feature that converts an analog voltage on a pin to a digital number. By converting from the analog world to the digital world, we can begin to use electronics to interface to the analog world around us.

A **successive-approximation ADC** uses a comparator to successively narrow a range that contains the input voltage. At each successive step, the converter compares the input voltage to the output of an internal digital to analog converter which might represent the midpoint of a selected voltage range. At each step in this process, the approximation is stored in a **successive approximation register (SAR).** For example, consider an input voltage of 6.3 V and the initial range is 0 to 16 V. For the first step, the input 6.3 V is compared to 8 V (the midpoint of the 0–16 V range). The comparator reports that the input voltage is less than 8 V, so the SAR is updated to narrow the range to 0–8 V. For the second step, the input voltage is compared to 4 V (midpoint of 0–8). The comparator reports the input voltage is above 4 V, so the SAR is updated to reflect the input voltage is in the range 4–8 V. For the third step, the input voltage is compared with 6 V (halfway between 4 V and 8 V); the comparator reports the input voltage is greater than 6 volts, and search range becomes 6–8 V. The steps are continued until the desired resolution is reached.

> **Example.** For a 3-bit ADC on a Teensy 3.2 microcontroller with a 3.3 V reference, what would the analog reading be for a 1.5 V signal?
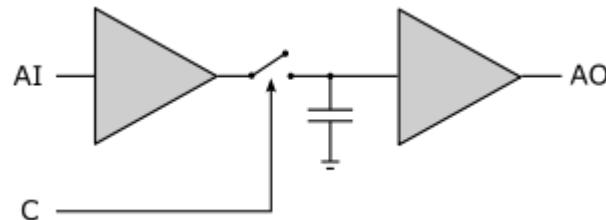
ADC Reference	Signal Higher than Reference?	SAR Bit Stored
1.625 V	No	0
0.8125 V	Yes	1
1.21875 V	Yes	1

> The SAR stores 0b011, which is decimal 3, out of a possible 0b111, which is decimal 7. So, the analog voltage read is:
>
> $$V_{read} = \frac{3}{7}(3.3\ V) = 1.414\ V$$
>
> As the resolution increases, the estimate of the analog voltage also gets better.
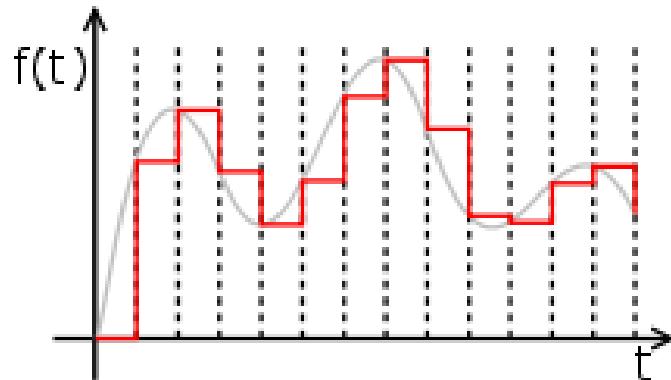
Inside of a microcontroller, the circuit responsible for performing ADC starts with a special configuration known as a **sample and hold (S&H) circuit**, which is simplified below:

A clock signal (from the microcontroller) causes the switch to close when a sample is to be taken (AI). After the switch closes, it is quickly reopened and the voltage that was sampled is stored on the capacitor.

The second operational amplifier shown acts as a follower, isolating the output from the input circuit. This effectively discretizes the input signal and ensures that each ADC avoids interference from a change in the input, resulting in something similar to:

The output of the sample and hold circuit is then passed to another operational amplifier acting as a comparator, which compares the voltage to a reference as dictated by the SAR. These references are always held relative to analog ground (AGND) on microcontrollers that have such a pin, such as a Teensy 3.2. One might notice that a microcontroller has two pins, both labelled for ground (**GND** and **AGND**). Inside of the microcontroller, AGND is directly connected to GND via an inductor, to separate digital signals from analog signals. The ADC circuit operates relative to AGND, **so it is imperative to remember that analog voltage readings should take place on circuits grounded to AGND**. If they are not, then excessive noise can be expected in the read analog signal.

The internal ADC can be used on a microcontroller using the built in Arduino function `analogRead()`, described below:

`analogRead(pin)`

Purpose: Reads the value from the specified analog pin. The Arduino board contains a 6 channel (8 channels on the Mini and Nano, 16 on the Mega), 10-bit analog to digital converter. This means that it will map input voltages between 0 and 5 V (3.3 V for a Teensy 3.2) into integer values between 0 and 1023. This yields a resolution between readings of: 5 volts / 1024 units or, .0049 volts (4.9 mV) per unit (3.2 mV on the Teensy 3.2). The input range and resolution can be changed using `analogReference()`.

It takes about 100 microseconds (0.0001 s) to read an analog input, so the maximum reading rate is about 10,000 times a second.

Parameters: *pin*: the number of the pin whose mode you wish to read an analog voltage on (must be an analog pin)

`analogReadResolution(res)`

Purpose: Set the resolution of the ADC. For a Teensy 3.2, the maximum value this can be is 12-bit.

Parameters: *res:* the number of bits dedicated to the SAR for the ADC

An example of using `analogRead()` would be to read an analog voltage of a device such as a **potentiometer**. A potentiometer is any device that can change resistance based on user input (such as turning of a screw). They typically have three terminals, and the mechanism in which they work is best illustrated below (Figure 29):
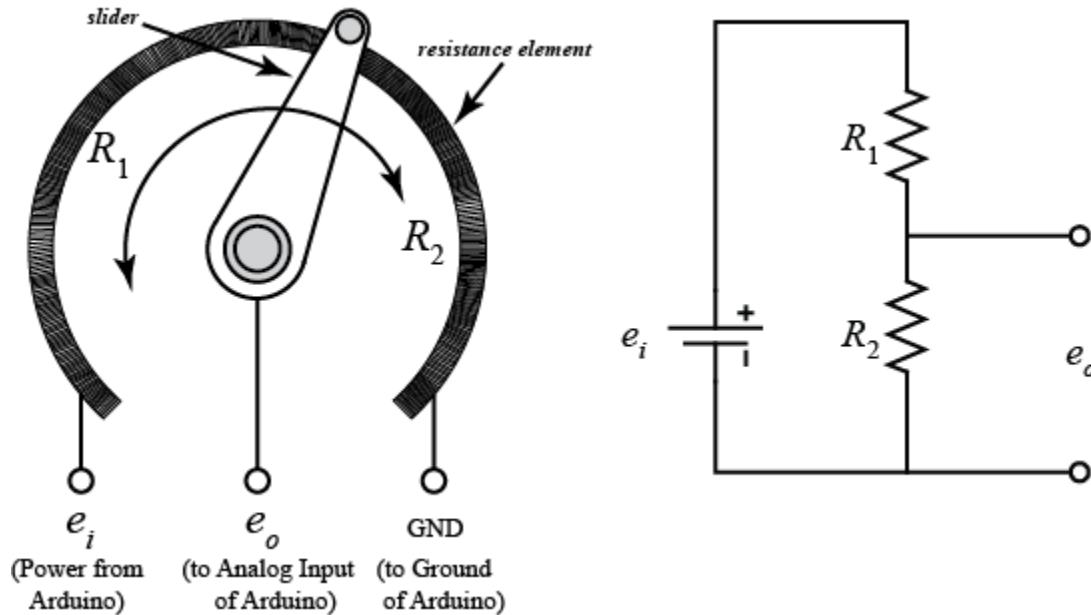

Figure 29. Potentiometer Schematic
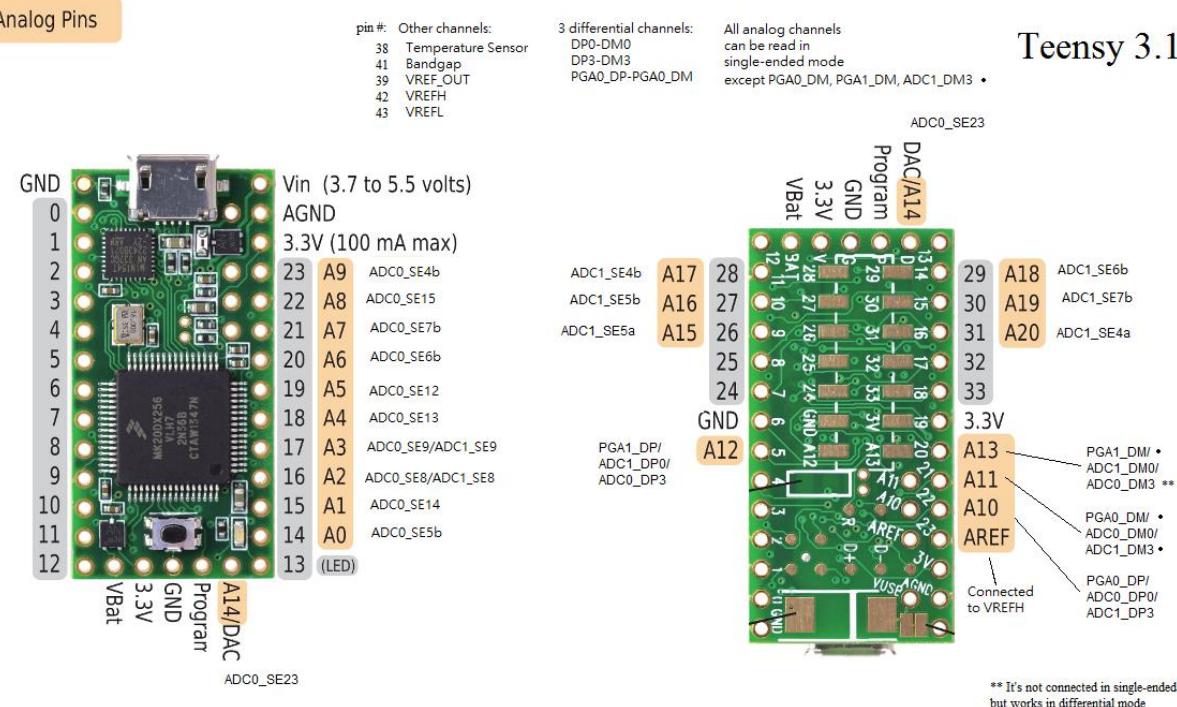
```
1  // Read an Analog Potentiometer Voltage
2
3  int analogPin = 3;     // potentiometer wiper (middle terminal) connected to analog pin 3
4  int val = 0;           // variable to store the value read
5
6  void setup()
7
8  {
9    Serial.begin(9600);           //  setup serial
10 }
11
12 void loop()
13 {
14   val = analogRead(analogPin);    // read the input pin
15 }
```
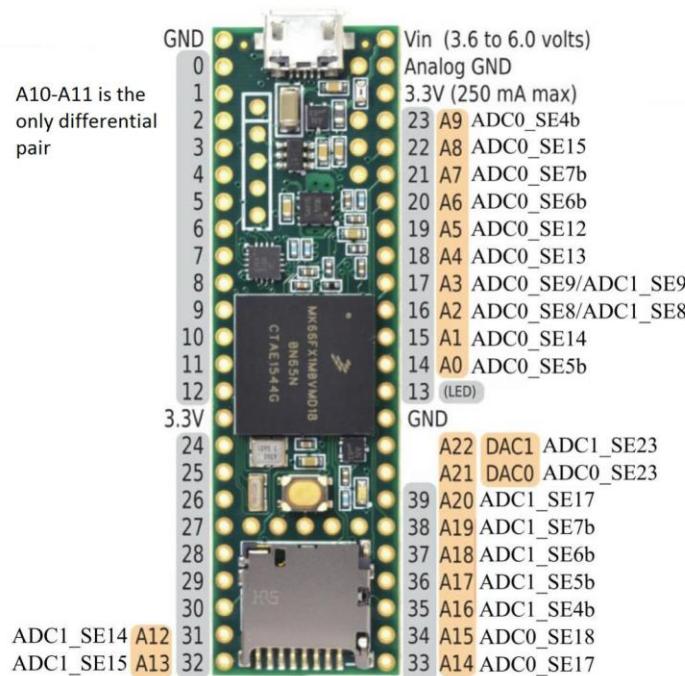
The Teensy 3.2 has two ADCs on board, making it possible to take synchronous measurements. Several functions, including some that change the resolution of the ADC and enable synchronous or continuous (or both) measurements are provided in the **ADC.h** library which must be included to activate that functionality. A list of the functions is provided here where the library was first introduced and more detail on the library is gone into in the next subsection.

ADC Library for Teensy Microcontrollers

As mentioned before, the ADC.h library allows for the user of a Teensy 3.0 or later microcontroller to easily take full advantage of the capabilities of the hardware. As stated, the Teensy has two onboard ADCs and they can only be used on specific pins with some overlap. For the Teensy 3.1/3.2 this breakdown is as follows:


For the newer Teensy 3.6 this breakdown takes the following form:

Obviously, the library must be included into the sketch using a compiler directive, but then the following functions become available (the old functions are also still available and can be called by putting "adc->" before the function):

```
ADC *adc = new ADC();
```

Purpose: Creates the new ADC object that is required to use the newly defined functions

Parameters: *none*

```
adc->setReference(option, ADC_x);
```

Purpose: Sets the voltage reference on one of the two Teensy ADCs (`ADC_0` or `ADC_1`) to one of three options:

(a) `ADC_REFERENCE::REF_3V3`: All boards have the 3.3 V output of the regulator (VOUT33 in the schematics). If VIN is too low, or you are powering the Teensy directly to the 3.3V line, this value can be lower. In order to know the value of the final measurement you need to know the value of the reference in some other way. This is the default option.

(b) `ADC_REFERENCE::REF_EXT`: The second option available to all boards is the external reference, the pin **AREF**. Simply connect this pin to a known voltage and it will be used as reference. The AREF pin is 3.3 V tolerant (higher voltages will likely ruin the microcontroller).

(c) `ADC_REFERENCE::REF_1V2`: Teensy 3.0 and 3.1/3.2 have an internal 1.2V reference. Use it when voltages are lower than 1.2V or when using the PGA in Teensy 3.1/3.2.

Parameters: *option:* the voltage reference to be used as defined above

    ADC_x: the ADC to set the reference on (either `ADC_0` or `ADC_1`)

```
adc->setSamplingSpeed(speed, ADC_x);
```

Purpose: Sets the amount of time to allow the sample-and-hold circuit capacitor to load up the voltage that is trying to be measured. The longer the time for the capacitor to load up, the more accurate the reading. For lower impedance readings, the speed can be raised, but for higher impedance readings, it should be lowered. The speed can be set based on some predetermined values from the ADC_SAMPLING_SPEED class enum (where *ADCK* is the ADC clock speed which can be changed):

- `ADC_SAMPLING_SPEED::VERY_LOW_SPEED` is the lowest possible sampling speed (+24 ADCK)
- `ADC_SAMPLING_SPEED::LOW_SPEED` adds +16 ADCK.
- `ADC_SAMPLING_SPEED::MED_SPEED` adds +10 ADCK.
- `ADC_SAMPLING_SPEED::HIGH_SPEED` adds +6 ADCK.
- `ADC_SAMPLING_SPEED::VERY_HIGH_SPEED` is the highest possible sampling speed (0 ADCK added).

Parameters: *speed:* the sampling speed in terms of how many ADCK are added between samples

    ADC_x: the ADC to set the reference on (either `ADC_0` or `ADC_1`)

```
adc->setConversionSpeed(speed, ADC_x);
```

Purpose: Sets the amount of time to allow for the actual analog to digital conversion, which depends on the bus speed. The speed can be selected from predetermined values of the ADC_CONVERSION_SPEED class enum:

- `ADC_CONVERSION_SPEED::VERY_LOW_SPEED` is guaranteed to be the lowest possible speed within specs for resolutions less than 16 bits (higher than 1 MHz), it's different from `ADC_LOW_SPEED` only for 24, 4 or 2 MHz.
- `ADC_CONVERSION_SPEED::LOW_SPEED` is guaranteed to be the lowest possible speed within specs for all resolutions (higher than 2 MHz).
- `ADC_CONVERSION_SPEED::MED_SPEED` is always greater than or equal to `ADC_LOW_SPEED` and less than or equal to `ADC_HIGH_SPEED`.
- `ADC_CONVERSION_SPEED::HIGH_SPEED_16BITS` is guaranteed to be the highest possible speed within specs for all resolutions (lower or equal than 12 MHz).
- `ADC_CONVERSION_SPEED::HIGH_SPEED` is guaranteed to be the highest possible speed within specs for resolutions less than 16 bits (lower or equal than 18 MHz).
- `ADC_CONVERSION_SPEED::VERY_HIGH_SPEED` may be out of specs, it's different from ADC_HIGH_SPEED only for 48, 40 or 24 MHz.

Parameters: *speed:* the conversion speed

ADC_x: the ADC to set the reference on (either `ADC_0` or `ADC_1`)

```
adc->setAveraging(num_samples);
```

Purpose: Establishes the number of samples to take in and average for each voltage measurement. This is very useful for a type of digital filtering known as oversampling which can under certain circumstances increase the resolution of the ADC.

Parameters: *num_samples*: the number of samples to take in and average.

```
ADC::Sync_Result adc->analogSyncRead(pin1, pin2);
```

Purpose: Takes a synchronous voltage measurement on two analog pins at the same time. It will set up both ADCs and measure `pin1` with `ADC_0` and `pin2` with `ADC_1`. The result is stored in the structure `ADC::Sync_Result`, with members *.result_adc0* and *.result_adc1* so that you can get both. ADC_0 has to be able to access `pin1` (same for `pin2` and ADC1). If the pin can't be accessed by the ADC you selected it will return `ADC_ERROR_VALUE`. See the example files for more details.

Parameters: *pin1, pin2*: the two analog pins on which the voltage is being read

```
adc->analogReadDifferential(pin1, pin2);
```

Purpose: Takes a differential voltage measurement between two pins and returns the value as an integer.

Parameters: *pin1, pin2*: the two analog pins on which the voltage is being read

```
adc->analogSyncReadDifferential(pin1, pin2);
```

Purpose: Takes a synchronous differential voltage measurement between two pins and returns the value as an integer. ADC_0 has to be able to access `pin1` (same for `pin2` and ADC1). If the pin can't be accessed by the ADC you selected it will return `ADC_ERROR_VALUE`.

Parameters: *pin1, pin2*: the two analog pins on which the voltage is being read

```
adc->startContinuous(pin, ADC_xx);
```

Purpose: Starts continuous analog to digital conversions on a pin using the specified ADC. It returns as soon as the ADC is set, use `analogReadContinuous()` to read the value.

Parameters: *pin:* the pin to start continuous conversion on

       ADC_xx: the ADC to set the reference on (either `ADC_0` or `ADC_1`)

```
adc->startContinuousDifferential(pin1, pin2, ADC_xx);
```

Purpose: Starts continuous differential analog to digital conversions between two pins using the specified ADC. If single-ended and 16-bit then it will be required to cast the result as an unsigned integer to avoid sign issues with voltage readings above 1.65 V

Parameters: *pin1, pin2:* the pins to start continuous conversion on

       ADC_xx: the ADC to set the reference on (either `ADC_0` or `ADC_1`)

```
adc->startSynchorinzedContinuous(pin1, pin2);
```

Purpose: Starts a continuous differential conversion in both ADCs simultaneously. It will set up both ADCs and measure `pin1` with `ADC_0` and `pin2` with `ADC_1`. ADC_0 has to be able to access `pin1` (same for `pin2` and ADC1). If the pin can't be accessed by the ADC you selected it will return `ADC_ERROR_VALUE`.

Parameters: *pin1, pin2:* the pins to start continuous conversion on

```
ADC::Sync_Result adc->readSynchorinzedContinuous();
```

Purpose: Reads the continuous synchronous voltage measurement on two analog pins at the same time. The result is stored in the structure `ADC::Sync_Result`, with members *.result_adc0* and *.result_adc1* so that you can get both. See the example files for more details.

Parameters: *none*

```
adc->printError();
```

Purpose: Prints any errors from all ADC, if any.

Parameters: *none*

18.   Pulse-Width Modultion and Digital to Analog Conversion (DAC)

Pulse-width modulation (PWM), or pulse-duration modulation (PDM), is a modulation technique used to encode a message into a pulsing signal. Although this modulation technique can be used to encode information for transmission, its main use is to allow the control of the power supplied to electrical devices, especially to inertial loads such as motors.

A pulse width signal is a digital square wave that is characterized by its **duty cycle** (or percent of the time the signal is HIGH) (Figure 30).
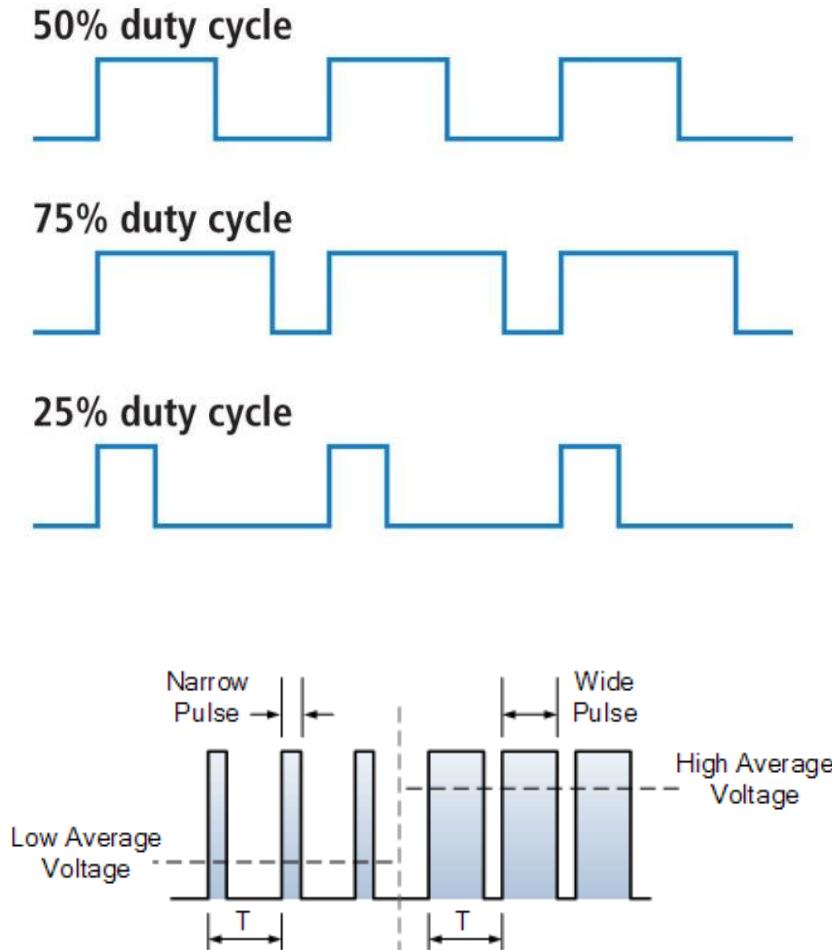
Figure 30. Pulse Wave

Because PWM waves have a very high frequency, they appear as analog signals in the real world. The power associated with one of these waves is given by $P = V_{RMS}I_{RMS} = DV_{peak}I_{peak}$, where D is the duty, a number between 0 and 1. Because of the nature of RMS (Root Mean Squared) voltages and currents, it should be noted that for analog voltage generated from a PWM wave:

$$V = V_{peak}\sqrt{D}$$
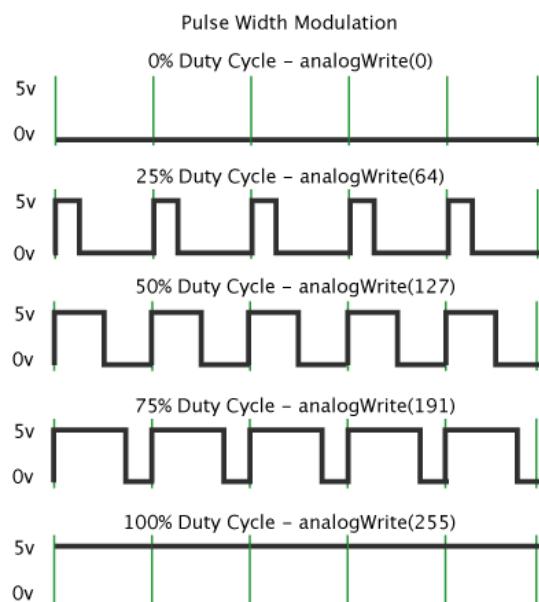
PWM signals can be written using the `analogWrite()` function.

```
analogWrite(pin, value)
```

Purpose: Writes an analog value (PWM wave) to a pin. Can be used to light a LED at varying brightnesses or drive a motor at various speeds. After a call to `analogWrite()`, the pin will generate a steady square wave of the specified duty cycle until the next call to `analogWrite()` (or a call to `digitalRead()` or `digitalWrite()` on the same pin). The frequency of the PWM signal on most pins is approximately 490 Hz. On the Uno and similar boards, pins 5 and 6 have a frequency of approximately 980 Hz. Pins 3 and 11 on the Leonardo also run at 980 Hz. You do not need to call `pinMode()` to set the pin as an output before calling `analogWrite()`.The `analogWrite()` function has nothing to do with the analog pins or the `analogRead()` function.

Parameters: *pin*: the pin that is being written to

        value: the duty cycle: between 0 (always off) and 255 (always on) for 8-bit resolution.

An example of using both `analogWrite()` and `analogRead()` is presented below in code that will light up an LED attached to pin 9 with a PWM signal based on the reading a potentiometer attached to pin 9.

```
1  int ledPin = 9;        // LED connected to digital pin 9
2  int analogPin = 3;     // potentiometer connected to analog pin 3
3  int val = 0;           // variable to store the read value
4
5  void setup()
6  {
7    pinMode(ledPin, OUTPUT);   // sets the pin as output
8  }
9
10 void loop()
11 {
12   val = analogRead(analogPin);   // read the input pin
13   analogWrite(ledPin, val / 4);  // analogRead values go from 0 to 1023, analogWrite values from 0 to 255
14 }
```

Functions exist for changing the characteristics of the pulse width generated by `analogWrite()`, such as:

`analogWriteResolution(res)`

Purpose: Set the resolution of the PWM generated by `analogWrite()`. The default is 8-bit resolution, but the maximum resolution can be as high as 32 bits on some microcontrollers. If a value is specified higher than the DAC capabilities of a pin on a microcontroller, the extra bits will simply be truncated.

Parameters: *res*: the resolution of the PWM wave in bits

`analogWriteFrequency(pin, freq)`

Purpose: Sets the frequency of the PWM wave. Depending on the microcontroller, there is likely an optimal frequency that this should be set to. There is a lower limit of a few Hz before this fails to accurately work.

Parameters: *pin:* pin on which to change the frequency of the PWM wave.

freq: the frequency of the PWM wave in Hertz

`tone(pin, freq, duration)`

Purpose: Generates a square wave of the specified frequency (and 50% duty cycle) on a pin. A duration can be specified, otherwise the wave continues until a call to `noTone()`. The pin can be connected to a piezo buzzer or other speaker to play tones (such as in the Arduino tutorial here). Only one tone can be generated at a time. If a tone is already playing on a different pin, the call to `tone()` will have no effect. If the tone is playing on the same pin, the call will set its frequency.

Parameters: *pin:* pin on which to apply the tone

freq: the frequency of the PWM wave in Hertz as unsigned integer type

duration (optional): how long the tone should be applied in milliseconds as unsigned long

`noTone()`

Purpose: Stops the generation of a square wave triggered by `tone()`, and has no effect if no tone is being generated. If you want to play different pitches on multiple pins, you need to call `noTone()` on one pin before calling `tone()`, on the next pin.

Parameters: *none*

Logic Level Conversion and H-Bridges

This entire document has focused on using either the 5 V Arduino or the 3.3 V Teensy microcontrollers. There are several situations that might arise where a certain sensor requires interfacing with a 3.3 V logic signal or some motor driver requires 5 V logic to function. To get over this obstacle you need a device that can shift 3.3V up to 5V, or 5V down to 3.3V. This is called **logic level shifting**. Level shifting is a dilemma so common that a board has been created that solves the issue: the **Bi-Directional Logic Level Converter**. The actual circuit to accomplish this is quite simple, requiring two pull-up resistors (discussed in [Mechanical Switches and Switch Debouncing](#)) and an N-channel MOSFET. The circuit is provided below for the SparkFun BSS138:
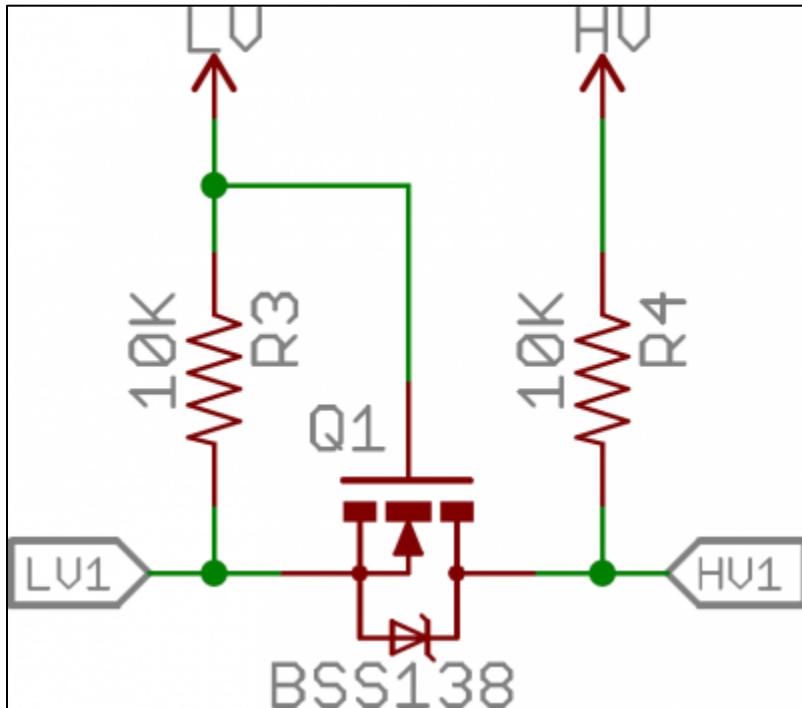

Figure 31. Bi-Directional Logic Level Converter

LV and HV are constant voltages at the low level and high level, respectively. LV1 and HV1 are the logic levels (one of which will be the input and one the output). The method of operation is simple, as LV1 is input LOW, the MOSFET will connect LV1 and HV1 so that HV1 is then LOW. As LV1 is input HIGH, the MOSFET separates the two legs and HV1 is pulled up to HV.

As HV1 moves LOW, the drain substrate diode pulls LV1 low and the MOSFET again becomes conducting and allows for level converting. Lastly, as HV1 in input HIGH, the MOSFET is not conductive and LV1 remains HIGH. Hence, the level can be converted in both directions.

This is one very convenient way to convert digital 3.3 V signals to 5 V signals or vice versa. The SparkFun Bi-Directional Level Converter takes on the following appearance, with four distinct data channels for conversion:
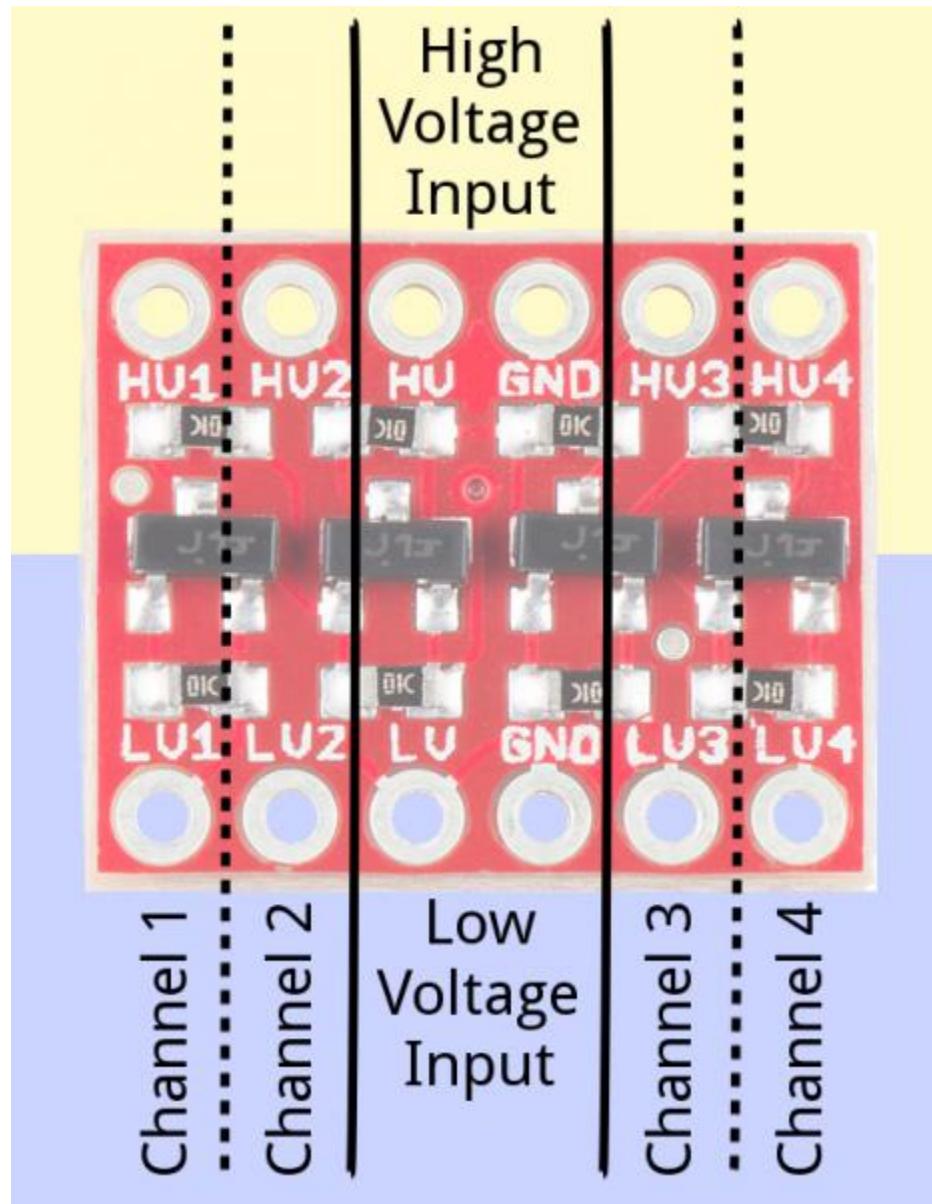

Figure 32. SparkFun Bi-Directional Level Converter

These types of level converters are limited to how much current they can pass and also cannot perform a task such as moving a DC motor in two directions. This type of task is better accomplished by a bit of circuitry known as an **H-Bridge**. An H bridge is an electronic circuit that enables a voltage to be applied across a load in either direction and is composed of nothing but MOSFETs and diodes.

In general, an H-bridge is a rather simple circuit, containing four switching elements, with the load at the center, in an H-like configuration:
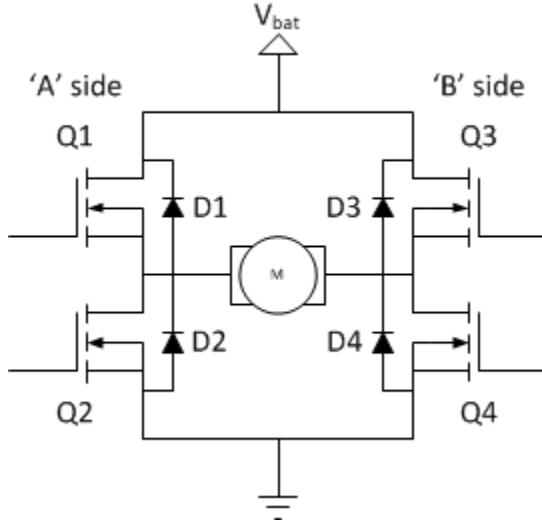

Figure 33. H-Bridge Circuit

The basic operating mode of an H-bridge is simple: if Q1 and Q4 are turned on, the left lead of the motor will be connected to the power supply, while the right lead is connected to ground. Current starts flowing through the motor which energizes the motor in the forward direction and the motor shaft starts spinning.
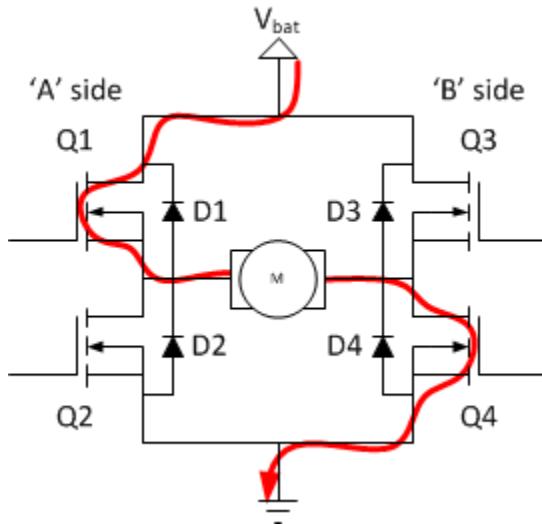

Figure 34. H-Bridge Driving Motor Forward

If Q2 and Q3 are turned on, the reverse will happen, the motor is energized in the reverse direction, and the shaft will start spinning backwards.
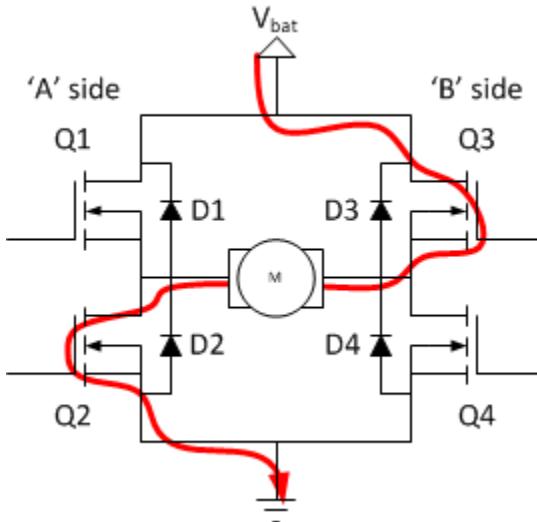

Figure 35. H-Bridge Driving Motor Backwards

In a bridge, both Q1 and Q2 (or Q3 and Q4) should never be closed at the same time. If you did that, you just have created a really low-resistance path between power and GND, effectively short-circuiting your power supply. This condition is called **shoot-through** and will almost assuredly destroy either the H-Bridge or some other part of the circuit.
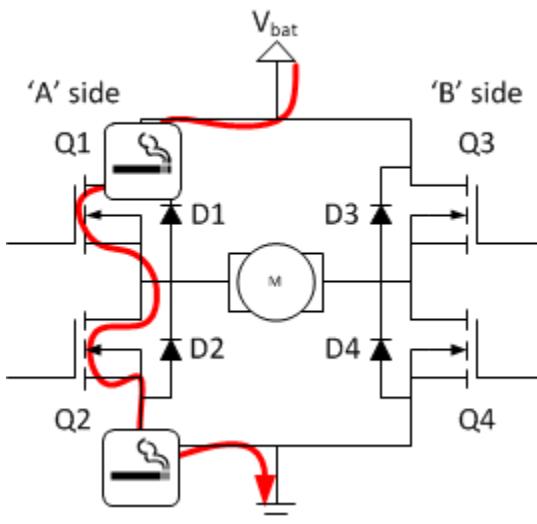

Figure 36. H-Bridge Short Cicuit (Shoot-through)

An H-Bridge that is designed to draw larger amounts of current and comes with a dedicated heat sink may be termed a **motor driver**.
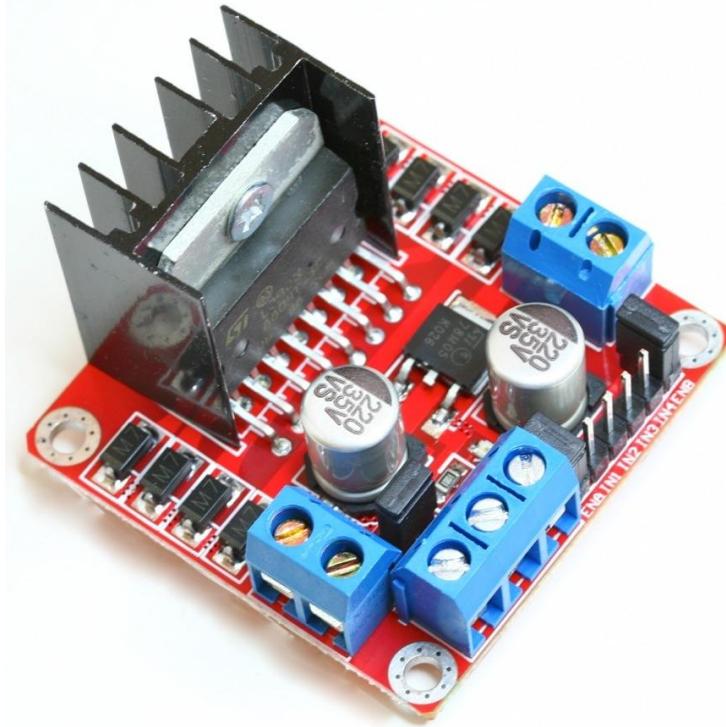

Figure 37. L298 Motor Driver

A motor driver will be powered by a high voltage source (typically on the order of 12-24 V but can be much higher depending on the specific driver ordered) and takes in either 3.3 V or 5 V microcontroller PWM logic on four pins (typically labelled IN1, IN2, IN3, and IN4). These inputs activate the transistors that act as the switches in the H-Bridge. The motors to be driven, which can be anything from a DC motor to a Peltier cooler, will have their inputs placed on the outputs of the motor driver. The above model has two separate screw terminals for these outputs.

The PWM output from the microcontroller dictates the voltage outputs such that fifty percent duty on IN1 and zero percent duty on IN2 might give half of the supply voltage to the motor in one direction. Placing fifty percent duty on IN2 and zero percent duty on IN1 might result in the same voltage differential, just in the opposite direction.

19.   Capacitive Sensing and Touch Pins

Capacitive sensing is useful for creating touch pads or buttons that do not require any force to activate. On Arduino boards and all Teensy models before the Teensy 3.0, capacitive sensing can be accomplished with two pins through the use of the CapacitiveSensor.h library. However, for later Teensy models an internal reference capacitor was added into the hardware that allows for a capacitor to be connected to a Touch pin and to ground and for its capacitance to be measured using the following function:

`touchRead(pin)`

Purpose: Reads the capacitance from the specified touch pin. This function returns a 16 bit number representing the capacitance on the pin with the number being equivalent to 0.02 pF. For example, if a 40 pF capacitor is connected to a touch pin, the number returned will be 2000. The measurement time depends on the capacitance. A worst-case measurement takes about 5 ms. Typical capacitances used for human touch typically read much faster. Even when cycling through all the touch sensitive pins, you get excellent sensitivity at very responsive speeds.

Parameters: *pin*: the number of the pin whose mode you wish to read the capacitance.

CapacitveSensor Library

While not as reliable or fast as measurements with the touch pins, using the CapacitiveSensor library does work well. Each sensor connects to two pins: send and receive. The CapacitiveSensor method toggles a microcontroller send pin to a new state and then waits for the receive pin to change to the same state as the send pin. A variable is incremented inside a while loop to time the receive pin's state change. The method then reports the variable's value, which is in arbitrary units. The send pin must connect with a large-value resistor, between 100KΩ to 50MΩ. Larger values allow more sensitivity, but with slower response. The receive pin may be connected with a wire, but a 1KΩ or higher resistor will help protect the Teensy's pin if a user directly touches the object and delivers an electro-static shock. The safest construction uses an insulating layer sensors can share a single send pin, but each must have its own receive pin. The idea of this type of layout is given below:
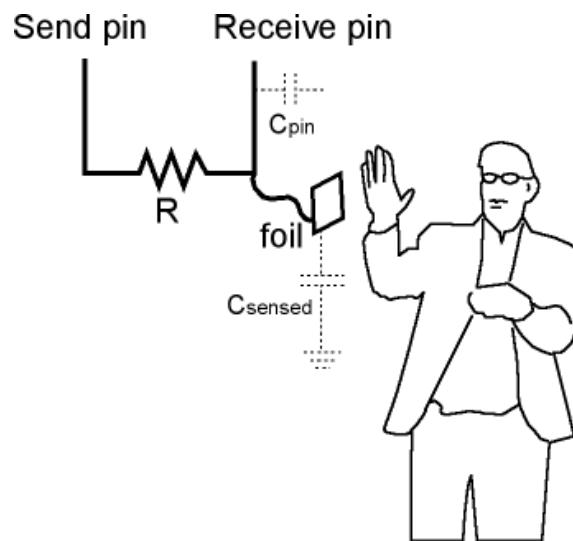

Figure 38. Capacitive Sensing Method of Operation

Teensy's ground pin should be connected to earth ground for best results. Normally the USB cable connects to a PC, which connects to earth ground by its power code. But when using a laptop on battery power or running without a computer, you may need to make a dedicated connection to earth ground.

The CapacitiveSensor library includes the following functions:

`CapacitiveSensor mySensor(sendPin, receivePin)`

Purpose: Create the `CapacitiveSensor` object, using a specific pair of pin. You should create a separate CapacitiveSensor object for each sensor. Sensors can share the same sendPin, but each needs its own receivePin.

Parameters: *sendPin*: pin that the microcontroller will toggle to a new state

receivePin: pin the microcontroller will wait to return data until it matches the state of *sendPin*

`mySensor.capacitiveSensor(numSamples)`

Purpose: Measures the sensor capacitance in arbitrary units (as it is proportional the elapsed time taken). The measurement is averaged based on the number of samples specified. More measurements increases sensitivity, but takes longer. A negative number is returned if any error occurs.

Parameters: *numSamples:* number of samples to average
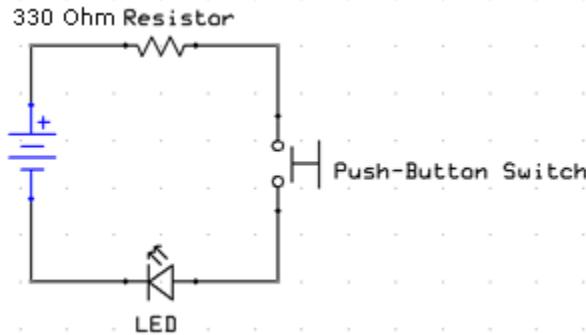
`mySensor.set_CS_Timeout_millis(timeout_millis)`

Purpose: Determines how long to take before timing out if the receive (sense) pin fails to toggle in the same direction as the send pin. A timeout is necessary because a while loop will lock-up a sketch unless a timeout is provided. CS_Timeout_Millis' default value is 2000 milliseconds (2 seconds).
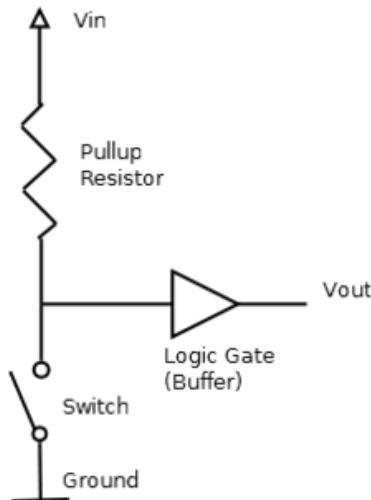
Parameters: *timeout_millis:* timeout threshold in milliseconds

20.   Mechanical Switches and Switch Debouncing

Mechanical switches are some of the most common interfaces to a microcontroller. They come in many forms, but the form many are familiar with are simple buttons. In theory, when the button is pressed down, it will complete an electrical connection between two terminals and when it is lifted up, the electrical connection will be broken and there will be an open circuit. The circuit could be as simple as a push-button switch being used to light up an LED, as shown. The resistor is placed to limit current, as an LED has a very low resistance.

Now is a good time to introduce the idea of a **pull-up resistor**. As shown in the following schematic, a pull-up resistor will simply make sure that the output voltage reads out HIGH when the switch is open, and the output voltage is moved LOW as the switch closes.

Often, a user wants to enable something like an LED that lights up when a button is pushed, with the button being attached to a digital pin on one side and ground on the other. This is a possibility because all digital pins on a Teensy 3.2 microcontroller have built in pull-up resistors that can be used to pull the signal HIGH when the button is released, and move it LOW when the button is pressed. This is enabled by using `pinMode(pin, INPUT_PULLUP).`

The problem with mechanical switches is that they are asynchronous to the microcontroller and that they are not electrically clean. This means that they are not likely to be pressed at the exact moment they are being checked on, so the user will have to hold the button down for some time, and sparks that fly as the

button is pressed/let go cause the signal to be messy. For example, the act of letting go of a button could generate the following response:
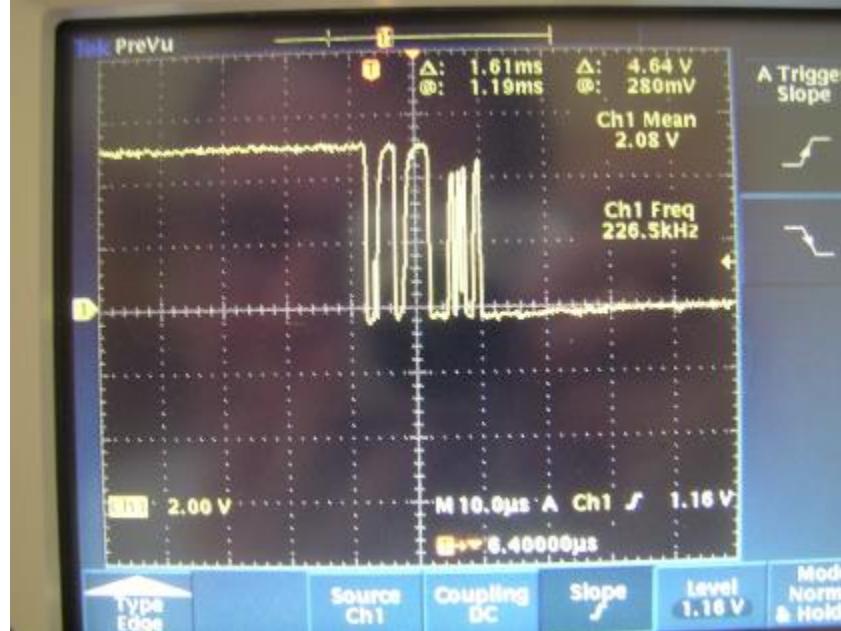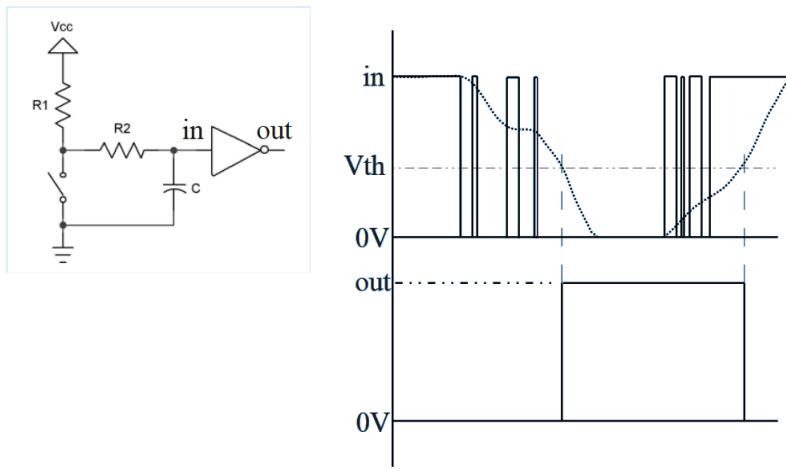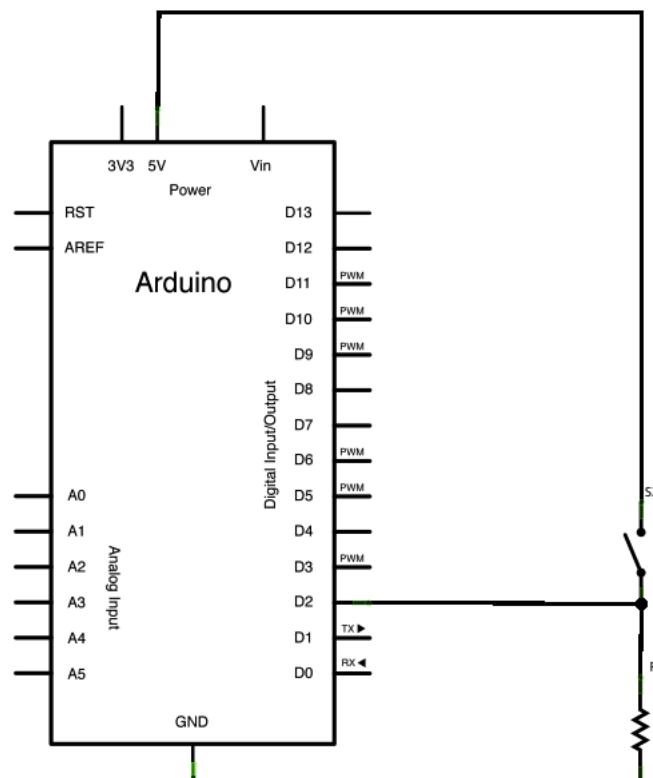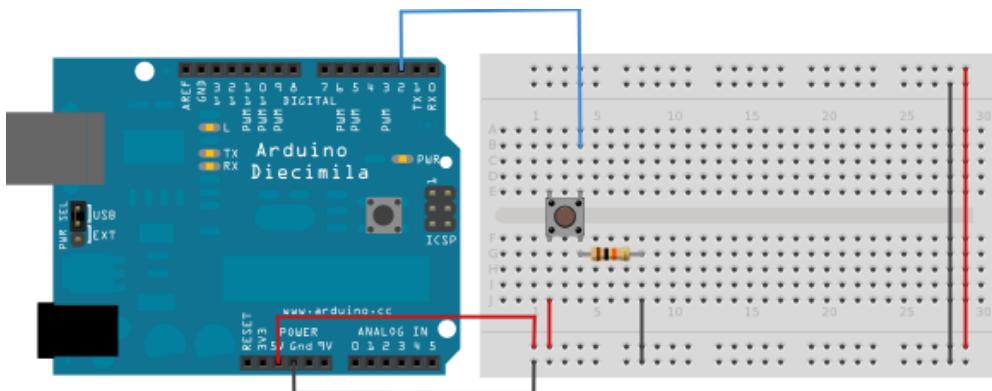

Figure 39. Raw signal from pushbutton

The microcontroller is likely to interpret this as the button being pushed several times, when all that really happened is that the button was released once. To resolve this, there are few options for what is known as **switch debouncing**. One such option that involves hardware is to incorporate a low pass filter, as shown (V_{th} is the threshold voltage):

However, this issue can be addressed in software as well and without the level of tediousness needed to select a resistor and capacitor that will correctly debounce the switch. Take the following schematic of a button wired up to a digital pin on an Arduino:

The general idea is to create some small amount of time and once a change in the state of the button (pressed or not pressed) occurs, wait that small amount of time and check the button again. If the change remains, register is as a true change. If the change has not remained, disregard it as noise. Note the use of `millis()`, which is the number of milliseconds the program has run (`micros()` could also be used which returns the number of microseconds that the program has run). The code that accomplishes this is listed below:

```
1  // Switch Debouncing
2
3  // Set Pin Numbers
4  const int buttonPin = 2;      // the number of the pushbutton pin
5  const int ledPin = 13;        // the number of the LED pin
6
7  // Initialize Variables
8  int ledState = HIGH;          // the current state of the output pin
9  int buttonState;              // the current reading from the input pin
10 int lastButtonState = LOW;    // the previous reading from the input pin
11 unsigned long lastDebounceTime = 0;  // the last time the output pin was toggled
12 unsigned long debounceDelay = 50;    // the debounce time; increase if the output flickers
13
14 void setup() {
15   pinMode(buttonPin, INPUT);
16   pinMode(ledPin, OUTPUT);
17
18   // set initial LED state
19   digitalWrite(ledPin, ledState);
20 }
21
22 void loop() {
23   // read the state of the switch into a local variable:
24   int reading = digitalRead(buttonPin);
25
26   // check to see if you just pressed the button
27   // (i.e. the input went from LOW to HIGH),  and you've waited
28   // long enough since the last press to ignore any noise:
29
30   // If the switch changed, due to noise or pressing:
31   if (reading != lastButtonState) {
32     // reset the debouncing timer
33     lastDebounceTime = millis();
34   }
35
36   if ((millis() - lastDebounceTime) > debounceDelay) {
37     // whatever the reading is at, it's been there for longer
38     // than the debounce delay, so take it as the actual current state:
39
40     // if the button state has changed:
41     if (reading != buttonState) {
42       buttonState = reading;
43
44       // only toggle the LED if the new button state is HIGH
45       if (buttonState == HIGH) {
46         ledState = !ledState;
47       }
48     }
49   }
50
51   // set the LED:
52   digitalWrite(ledPin, ledState);
53
54   // save the reading.  Next time through the loop,
55   // it'll be the lastButtonState:
56   lastButtonState = reading;
57 }
```

21. Interrupts and Interrupt Service Routines (ISRs)

The problem of the buttons being asynchronous to the microcontroller from the previous section has not yet been addressed. This problem is resolved with the use **interrupts**. Interrupts are useful for making things happen automatically in microcontroller programs, and can help solve timing problems. Good tasks for using an interrupt may include reading a rotary encoder, or monitoring user input. In essence, by attaching an interrupt to a pin, whenever that pin moves HIGH, a special type of function is executed, regardless where the program currently is in the code itself. This function is called an **Interrupt Service Routine (ISR)**. An ISR should have no input parameters, nor should it return anything. Typically, an ISR should be as short and as fast as possible to avoid conflicting with the main program. Variables that are changed inside of an ISR may be currently being referenced inside of the main program, and so they should be given a unique data type known as **volatile**. A volatile variable is loaded from **RAM** (Random Access Memory) and not from the storage registry, and so is suitable for when the variable must be changed mid-computation due to an interrupt.

This very simple example displays the syntax of using an interrupt by making it so whenever pin 4 changes, the LED turns on or off:

```
1  // Button Lighting up an LED ISR Example
2
3  // Define pins for both the button and the LED
4  const int red_btn = 4;
5  const int red_led = 5;
6
7  // Define the state of the LED as volatile (because it can change in mid-loop)
8  volatile bool btn_state = 0;
9
10 void setup()
11 {
12   // Specify pin modes
13   pinMode(red_btn, INPUT);
14   pinMode(red_led, INPUT);
15
16   // Attach the interrupt to the pin with the button and activate it upon any change
17   attachInterrupt(digitalPinToInterrupt(red_btn), myISR, CHANGE);
18
19 }
20
21 void loop()
22 {
23   // Disable interrupts while the LED is being lit up or dimmed and then re-enable them
24   noInterrupts();
25   digitalWrite(red_led, btn_state);
26   interrupts();
27 }
28
29 // Interrupt Service Routine
30 void myISR()
31 {
32   btn_state = !btn_state;
33 }
```

There are a number of functions that are introduced here, and their documentation is as follows:

`attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)`

Purpose: Attaches an interrupt to a digital pin, specified which ISR to associate with this interrupt, and what has to happen in order for the interrupt to be activated.

Parameters: *pin*: the pin that has the interrupt attached

ISR: the ISR to call when the interrupt occurs (must be a void function with no inputs)

mode: defines when the interrupt should be triggered.

Four constants are predefined as valid values for *mode*:

- LOW to trigger the interrupt whenever the pin is low,
- CHANGE to trigger the interrupt whenever the pin changes value
- RISING to trigger when the pin goes from low to high,
- FALLING for when the pin goes from high to low.

`detachInterrupt(digitalPinToInterrupt(pin))`

Purpose: Turns off the given interrupt.

Parameters: *pin*: the pin that has the interrupt attached

`noInterrupts()`

Purpose: Disables interrupts (you can re-enable them with `interrupts()`). Interrupts allow certain important tasks to happen in the background and are enabled by default. Some functions will not work while interrupts are disabled, and incoming communication may be ignored. Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical sections of code.

Parameters: *none*

`Interrupts()`

Purpose: Re-enables interrupts (after they've been disabled by `noInterrupts()`). Interrupts allow certain important tasks to happen in the background and are enabled by default. Some functions will not work while interrupts are disabled, and incoming communication may be ignored. Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical sections of code.

Parameters: *none*

22. Analog Sensors (Accelerometers)

An **analog sensor** is a sensor that outputs an analog voltage that is in some way related to the measurement the sensor is supposed to take. Analog Devices sells many of these types of sensors. While analog sensors are for the most part obsolete to their digital successors, they can still frequently find use and tend to be fairly cheap.

An **accelerometer** is a device that measures acceleration in the Cartesian coordinates relative to free-fall ("proper acceleration"). Conceptually, the sensor behaves like a mass on a spring with damping. Upon accelerating, the mass is displaced and it is this displacement that is used to compute the acceleration. (in reality, the piezoelectric effect converts displacement directly into voltage which is then converted into an analog signal). The ADXL327 model, for example, has the following layout:
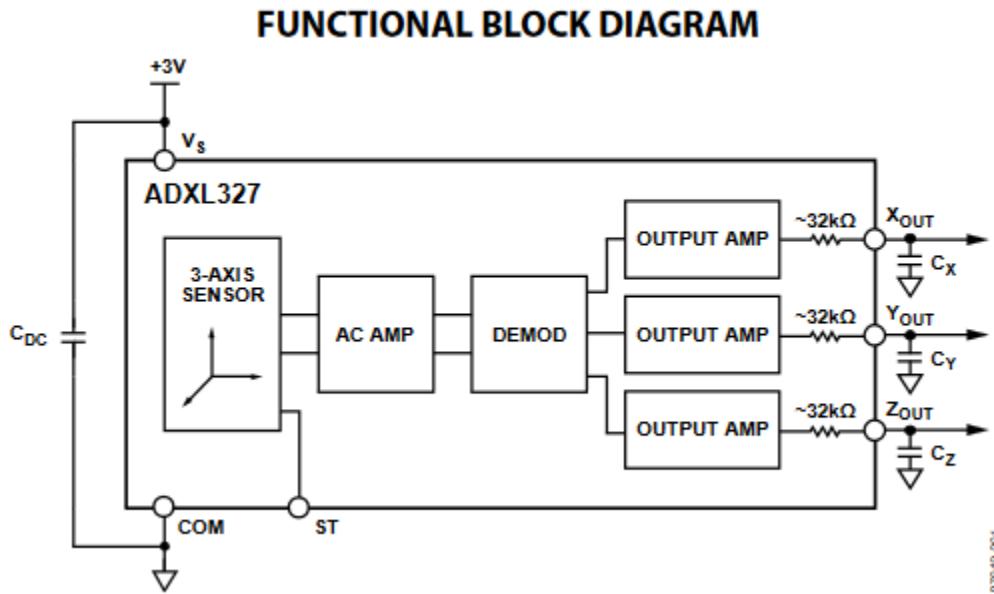

Figure 40. ADXL327 Block Diagram

There is a lot of things to notice in this diagram. First, **COM** denotes a common ground and is represented as a triangle on this diagram. Next, note that the sensor is fed into an AC amplifier which is then **demodulated**. This means that the effects of the measurement of one axis is separated from the effects of the other two axes. These signals are then amplified and passed through a low-pass filter where the user chooses the sizing of the filter's capacitor. The acceleration for each axis is then provided as an analog signal.

The **ST** pin reflects a test voltage that can be used instead of the sensor. The accelerometer itself can be supplied anywhere between 1.8 V and 3.6 V at approximately 350 μA to function. However, for analog sensors it is imperative to note that the output is **ratiometric**, which means it scaled relative to the supply voltage.

The capacitor located on the outside of the chip is what is known as a **decoupling capacitor**. Because the current draw into the sensor varies and the sensor and the wiring that connects to it all have an native inductance, back voltages will impede the proper amounts of power from getting to the sensor. The presence

of the capacitor bypass the effects of the inductance of the wire, because it acts as an energy storage unit that will supply the sensor with power even if a induced back electromotive force is present.

Accelerometers measure acceleration is **g**'s (where $1\ g\ \equiv 9.81\ ^m/_{s^2}$). This specific model claims to have a minimum sensitivity of $378\ \frac{mV}{g}$ and a maximum sensitivity of $462\ \frac{mV}{g}$, depending on the supply voltage (again, it is scaled because the output is ratiometric). The measuring range is given to be ± 2 g.

Table 4. ADXL327 Accelerometer Datasheet Excerpt

Table 1.

Parameter	Conditions	Min	Typ	Max	Unit
SENSOR INPUT	Each axis				
Measurement Range		±2	±2.5		g
Nonlinearity	Percent of full scale		±0.2		%
Package Alignment Error			±1		Degrees
Interaxis Alignment Error			±0.1		Degrees
Cross Axis Sensitivity[1]			±1		%
SENSITIVITY (RATIOMETRIC)[2]	Each axis				
Sensitivity at X_{OUT}, Y_{OUT}, Z_{OUT}	$V_S = 3\,V$	378	420	462	mV/g
Sensitivity Change Due to Temperature[3]	$V_S = 3\,V$		±0.01		%/°C
ZERO g BIAS LEVEL (RATIOMETRIC)					
0 g Voltage at X_{OUT}, Y_{OUT}	$V_S = 3\,V$	1.3	1.5	1.7	V
0 g Voltage at Z_{OUT}	$V_S = 3\,V$	1.2	1.5	1.8	V
0 g Offset vs. Temperature			±1		mg/°C
NOISE PERFORMANCE					
Noise Density X_{OUT}, Y_{OUT}, Z_{OUT}			250		µg/√Hz rms
FREQUENCY RESPONSE[4]					
Bandwidth X_{OUT}, Y_{OUT}[5]	No external filter		1600		Hz
Bandwidth Z_{OUT}[5]	No external filter		550		Hz
R_{FILT} Tolerance			32 ± 15%		kΩ
Sensor Resonant Frequency			5.5		kHz
SELF TEST[6]					
Logic Input Low			+0.6		V
Logic Input High			+2.4		V
ST Actuation Current			+60		µA
Output Change at X_{OUT}	Self test 0 to 1	−210	−450	−850	mV
Output Change at Y_{OUT}	Self test 0 to 1	+210	+450	+850	mV
Output Change at Z_{OUT}	Self test 0 to 1	+210	+770	+1400	mV
OUTPUT AMPLIFIER					
Output Swing Low	No load		0.1		V
Output Swing High	No load		2.8		V
POWER SUPPLY					
Operating Voltage Range		1.8		3.6	V
Supply Current	$V_S = 3\,V$		350		µA
Turn-On Time[7]	No external filter		1		ms
TEMPERATURE					
Operating Temperature Range		−40		+85	°C

This accelerometer, like most analog devices, also has some **zero offset bias**. This means that as the actual acceleration is zero, the output reflects some non-zero voltage. This means that the acceleration in g's could be computed as:

$$a = \left[\left(\frac{V_{ref}}{2^{resolution}}\right)\left(V_{read} - \alpha V_{supply}\gamma\right)\right]\left(\beta V_{supply}\delta\right)$$

α is the zero offset bias' dependence on the supply voltage

β is the signal sensitivity dependence on the supply voltage

γ is the mean offset bias across the range of voltage supplies

δ is the mean sensitivity across the range of voltage supplies

Zero offset bias is a parameter that can also be effected by the temperature of the sensor. This is mostly a factor as the sensor initially turns on and moves from cold to hot. By reading the temperature with a temperature sensor, this bias could be accounted for if it's a problem.

A tutorial on wiring this specific sensor and making code for it is provided on Arduino's website here.

NOTE: As far as accelerometers go, they are not able to measure certain types of angular rotation. Because of this, the inclusion of **gyroscopes** to help with this problem with this type of sensor lead to what is known as an **Inertial Measurement Unit (IMU)**. Analog IMUs are sold and a list of Analog Devices IMUs is provided here.
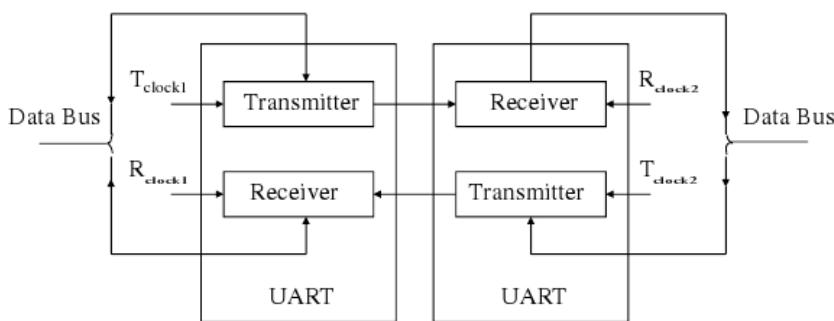
23.   Serial Communication

Serial communication utilizes digital data (HIGH, LOW) and timing to transmit data, compared to analog sensors that change an analog voltage based on their output. Some of the more common examples of serial communication to be explored are:

- **UART (Universal Asynchronous Receiving and Transmitting)**
  - **Examples:** TTL, RS232, RS422
  - Universal in potential usages, but slow (approximately 10 Mhz – 35 MHz)

- **SPI (Serial Peripheral Interface)**
  - Very fast (approximately 500 MHz)
  - Requires a unique "chip select" wire for each chip
  - Functions by allowing a "master" (microcontroller) to handle "slave' ICs (integrated circuits).

- **I^{2}C (Inter – Integrated Circuit)**
  - Fast (slower than SPI, but faster than UART)
  - Only requires two wires for up to approximately 63 slave devices.

- **CAN (Controller Aided Network)**
  - Works like I^{2}C but gives some sensors "priority" over others

- **One Wire**
  - Works similarly to I^{2}C but data and clock information are combined onto one wire.

UART Signals

UART is considered "universal" because its protocol is to send all data as character bytes. For example, the number 123.45, which could be sent as a four byte float, is instead sent as six bytes in the form of a character array. This is a big reason why the protocol is slower than its counterparts, because a lot of space is wasted during the transmission of data. UART is "asynchronous" because each device has a buffer (where the received data is stored) and an internal clock. Data does not have to be sent and received in any specific timeframe, because it is always sent to the buffer and parsed when the chip's internal clock dictates that it should be. Each device requires its own buffer, which can be inconvenient when working with more than two devices. The Teensy 3.2 microcontroller has three buffers available for use in UART communication (**Serial1**, **Serial2**, and **Serial3** ports). These ports comprise of two pins, **TX (transmitter)** and **RX (receiver)**. The interface between two UART devices can be very simply illustrated, as seen below:

UART signals come in a variety of signal types, some examples of which are given below:

Table 5. UART Examples

UART Signal Type	LOW (V)	HIGH (V)	Speed
Transistor-Transistor Logic (TTL)	0 to 0.8	2.0 to 5.0	Approximately 10 MHz
RS232	3.0 to 15.0	-3.0 to -15.0	10 MHz
RS422/EIA422	Positive	Negative	10 MHz
RS485	< -0.2	> 0.2	35 MHz

All types of UART can send information both directions at the same time, which means that the protocol is **full duplex.** This is in comparison to **simplex** protocols which only allow one-way communication, which UART is obviously capable of, and **half duplex**, which allows for two-way communication, just not at the same time.

UART data framing usually takes the following form:
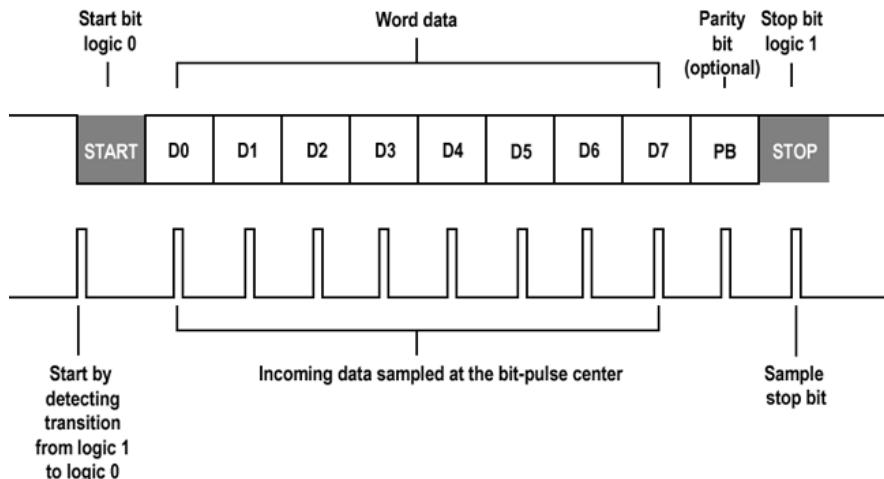
Figure 41. UART Data Structure

The start bit is logical LOW, and the stop bit is logical HIGH. These bits indicate when to start and stop reading the character array sent as the data. This ensures that the receiver checks the data line at a known interval (the **BAUD rate**, measured in bits/second). Two UART devices must be able to agree upon a BAUD rate, or else communication will be impossible. The data itself is sent as a byte with an optional ninth bit in the form of a **parity bit**. The parity bit returns a 1 or a 0 depending on if the number of 1's in the byte are even or odd (*Even parity bits* return 1 for an odd count and 0 for an even count, and *odd parity bits* return 0 for an odd count and 1 for an even count).

All UART hardware operations are controlled by a clock signal which runs at a multiple of the data rate (usually 8 times the BAUD rate). The receiver tests the state of the incoming signal on each clock pulse, looking for the start bit. If the apparent start bit lasts one half of the bit time, it counts as a valid start bit and signals the start of a new character byte to be received.

UART is frequently used in applications such as computer modems, and with **GPS** via the **NMEA (National Marine Electronics Association**) standard. Wireless communication and Bluetooth modules frequently use UART signals as well.

One other aspect about UART to mention is that all microcontroller boards have at least one Serial port (Serial) in which the board can communicate with a computer via the **USB (Universal Serial Bus)** port. This means that a board such as the Teensy 3.2 actually has four Serial ports altogether. The microcontroller can receive 5V power and transmit data via the USB protocol using the Serial port and the Serial object class created in the Arduino.h library. A basic example of demonstrating how to initialize Serial communication via the Arduino IDE is given below:

```
1  void setup(){
2    Serial.begin(9600);
3    Serial1.begin(38400);
4    Serial2.begin(19200);
5    Serial3.begin(4800);
6
7    Serial.println("Hello Computer");
8    Serial1.println("Hello Serial 1");
9    Serial2.println("Hello Serial 2");
10   Serial3.println("Hello Serial 3");
11 }
12
13 void loop() {}
```

There are several important functions to discuss when referring to Serial communication, which are outlined here:

```
Serial.begin(BAUD, optional: config)
```

Purpose: Sets the data rate in bits per second (BAUD) for serial data transmission on the Serial port specified (`Serial`, `Serial1`, `Serial2`, or `Serial3`). For communicating with the computer, use one of these rates: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200. You can, however, specify other rates for interfacing with other devices. An optional second argument configures the data, parity, and stop bits. The default is 8 data bits, no parity, one stop bit.

Parameters: *BAUD:* the data transfer rate (in bits/s) as long type

config: sets data, parity, and stop bits. Valid values are:

- SERIAL_5N1
- SERIAL_6N1
- SERIAL_7N1
- SERIAL_8N1 (the default)
- SERIAL_5N2
- SERIAL_6N2
- SERIAL_7N2
- SERIAL_8N2
- SERIAL_5E1
- SERIAL_6E1
- SERIAL_7E1
- SERIAL_8E1

- SERIAL_5E2
- SERIAL_6E2
- SERIAL_7E2
- SERIAL_8E2
- SERIAL_5O1
- SERIAL_6O1
- SERIAL_7O1
- SERIAL_8O1
- SERIAL_5O2
- SERIAL_6O2
- SERIAL_7O2
- SERIAL_8O2

`Serial.end()`

Purpose: Disables serial communication on the Serial port specified, allowing the RX and TX pins to be used for general input and output. To re-enable serial communication, call `Serial.begin().`

Parameters: none

`Serial.print(val, format)`

Purpose: Prints data to the serial port as human-readable ASCII text. This command can take many forms. Numbers are printed using an ASCII character for each digit. Floats are similarly printed as ASCII digits, defaulting to two decimal places. Bytes are sent as a single character. Characters and strings are sent as is. For example:

- `Serial.print(78)` gives "78"
- `Serial.print(1.23456)` gives "1.23"
- `Serial.print('N')` gives "N"
- `Serial.print("Hello world.")` gives "Hello world."

An optional second parameter specifies the base (format) to use; permitted values are BIN (binary, or base 2), OCT (octal, or base 8), DEC (decimal, or base 10), HEX (hexadecimal, or base 16). For floating point numbers, this parameter specifies the number of decimal places to use. For example:

- `Serial.print(78, BIN)` gives "1001110"
- `Serial.print(78, OCT)` gives "116"
- `Serial.print(78, DEC)` gives "78"
- `Serial.print(78, HEX)` gives "4E"
- `Serial.println(1.23456, 0)` gives "1"
- `Serial.println(1.23456, 2)` gives "1.23"
- `Serial.println(1.23456, 4)` gives "1.2346"

`Serial.println()` behaves the same as `Serial.print()` except that the printed value will be printed preceded by a new line character (so that the printed value is on a new line). You can pass flash-memory based strings to `Serial.print()` by wrapping them with `F()`.For example :

- `Serial.print(F("Hello World"))`

Parameters: *val*: the value to print (any data type)

        format: specifies the number base (for integral data types) or number of decimal places (for floating point types)

`Serial.printf(string, var1, var2, …)`

Purpose: Behaves the same as `Serial.print`() except allows the user to insert placeholders for variables known as **type field characters**. Common examples of these would be %d for a decimal value (integer) and %f for a float. A detailed list a type field characters and how to use them can be found [here](#).

Parameters: *string*: the string to print

> *variables*: the variables whose values will be placed in the string based on the type field characters used.

`Serial.available()`

Purpose: Returns the number of bytes (characters) available for reading from the serial port specified. This is data that's already arrived and stored in the **serial receive buffer** (which holds 64 bytes).

Parameters: none

`Serial.read()`

Purpose: Reads incoming serial data on the specified serial port and returns the first byte of incoming serial data available (or -1 if no data is available) as an integer.
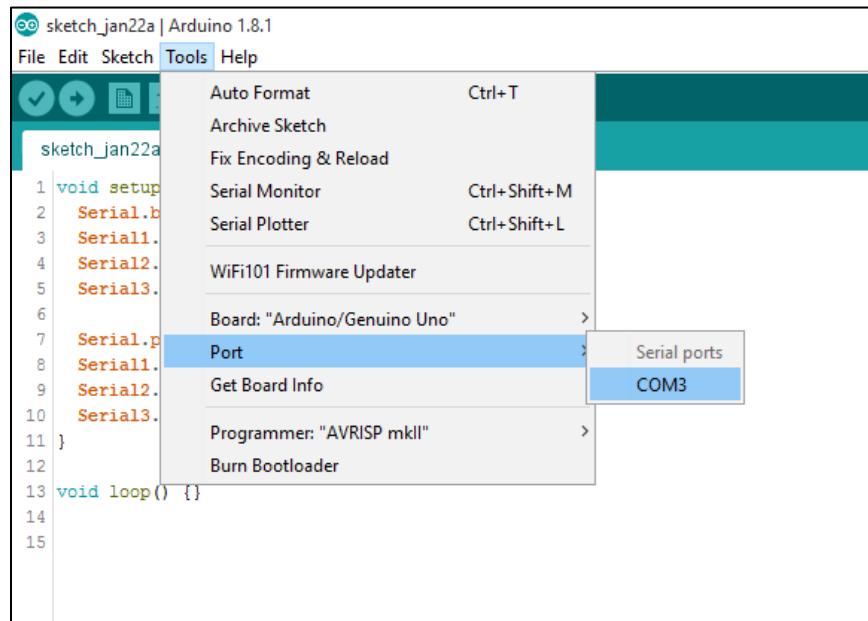
Parameters: none

`Serial.write(val)`
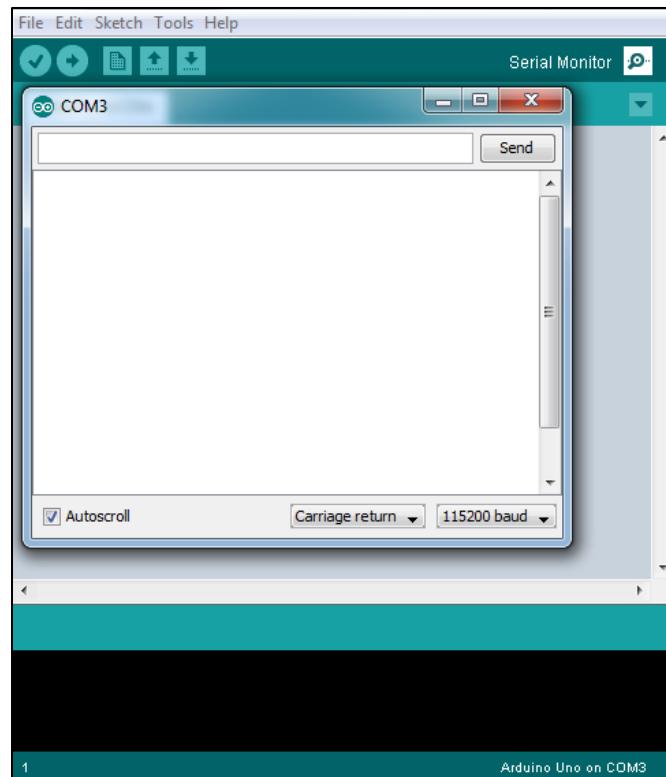
Purpose: Writes binary data to the serial port. This data is sent as a byte or series of bytes; to send the characters representing the digits of a number use the `print`() function instead.

Parameters: *val*: a value to send as a single byte or a string to send as a series of bytes.

To actually interface with the Serial monitor (Serial communication from microcontroller to computer through USB), first specify the COM port in which the USB is connected into the computer from the Arduino IDE (Tools -> Port -> COMxx)

The Serial Monitor can then be brought up from the Tools menu or the keyboard shortcut Ctrl + Shift + M. The Serial monitor will appear as the following window that will display text as its printed:

A very simple example for interfacing with data received from Serial communication is given below:

```
1  // Simple UART Example
2
3  int incomingByte = 0;    // for incoming serial data
4
5  void setup() {
6         Serial.begin(9600);     // opens serial port, sets data rate to 9600 bps
7  }
8
9  void loop() {
10
11         // send data only when you receive data:
12         if (Serial.available() > 0) {
13                // read the incoming byte:
14                incomingByte = Serial.read();
15
16                // say what you got:
17                Serial.print("I received: ");
18                Serial.println(incomingByte, DEC);
19         }
20 }
```

Another simple example showing off writing of Serial data is below:

```
1  // Simple UART Example #2
2
3  void setup(){
4    Serial.begin(9600);
5  }
6
7  void loop(){
8    Serial.write(45); // send a byte with the value 45
9
10     int bytesSent = Serial.write("hello"); //send the string "hello" and return the length of the string.
11 }
```

UART Flow Control

UART Flow Control is a method for slow and fast devices to communicate with each other over UART without the risk of losing data. Consider the case where two units are communicating over UART. A transmitter, *T*, is sending a long stream of bytes to a receiver, *R*. Now assume that *R* is a slower device than *T* and at a certain point, it cannot keep up with the data that is being received. At this point, *R* needs to either do some processing on the data or empty some buffers, before it can keep receiving data. In the meantime, *R* needs to tell *T* to stop transmitting for a while. This is where **flow control** comes in. Flow control provides extra signaling to inform the transmitter that it should stop (pause) or start (resume) the transmission. Several forms of flow control exist. What is referred to as hardware flow control uses extra

wires. The logic level on these wires define whether the transmitter should keep sending data or stop. With software flow control, special characters are sent over the normal data lines to start or stop the transmission.

With hardware flow control (also called **RTS/CTS flow control**), two extra wires are needed in addition to the data lines. They are called **RTS (Request to Send)** and **CTS (Clear to Send).** These wires are cross-coupled between the two devices, so RTS on one device is connected to CTS on the remote device and vice versa, as shown:
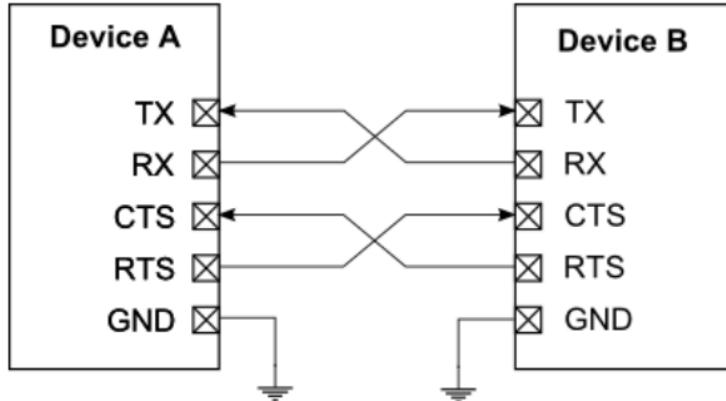

Figure 42. UART Basics with Flow Control

Each device will use its RTS to output if it is ready to accept new data and read CTS to see if it is allowed to send data to the other device. As long as a device is ready to accept more data it will keep the RTS line HIGH. It will take RTS LOW some time *before* its receive buffer is full. There might still be data on the line and in the other device transmit registers which has to be received even after RTS has been taken LOW. The other device is required to respect the flow control signal and pause the transmission until RTS is again brought HIGH.

The flow control is bidirectional, meaning both devices can request a halt in transmission. If one of the devices never have to request a stop in transmission (i.e. it is fast enough to always receive data), the CTS signal on the other device can be tied to the asserted logic level. The RTS pin on the fast device can thus be freed up to other functions.

With some UART protocols such as RS-232, there will be pins such as **Data Terminal Ready (DTR)** and/or **Data Set Ready (DSR)** which are likewise used for flow control. There is very little consistency from device to device on how these pins are to be used. For example, on a modem, the signal may be terminated when the DTR line is driven LOW in some cases, and in others it may do something else entirely.

It is worth noting that for older equipment, the CTS lines between two UART devices would be tied together and the RTS lines would also be tied together and the flow control would only be unidirectional.

Through the use of acknowledgement bytes (ACK) it is possible to regulate the flow of UART data in software as well.

SPI Signals

SPI is much faster than UART, on the order of 500 MHz in speed. SPI functions by allowing a "master" microcontroller to handle multiple "slave" IC's, which is cheap because the microcontroller can use its internal clock (making this interfacing synchronous with the microcontroller) , especially if the "slaves" are just sensors that are outputting information. The general wiring schematic is given by:
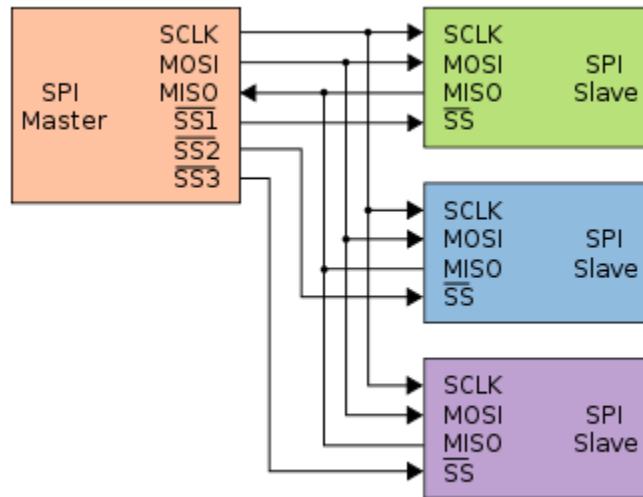

Figure 43. SPI Basic Wiring Diagram

SCLK ≡ Serial Clock

MOSI ≡ Master Out Slave In (**DO** ≡ Data Out)

MISO ≡ Master In Slave Out (**DI** ≡ Data In)

SS ≡ Slave Select (**CS** ≡ Chip Select)

SPI communication follows the following basic steps:

1. Master configured clock using frequency supported by slave (up to a few MHz).
2. Master selects slave device by sending logical LOW on the SS/CS pin.
3. Master waits while ADC works on slave to record measurement.
4. Full duplex communication occurs on MOSI/MISO pins using master clock.
5. Master selects next slave after data transmission is completed and process repeats.

The master must configure the **clock polarity (CPOL)** and the **clock phase (CPHA)** with respect to when to take the data. Polarity determines whether the line level is HIGH or LOW when the sensor is inactive, and Phase determines when the data is read (as the line falls or rises). This means:
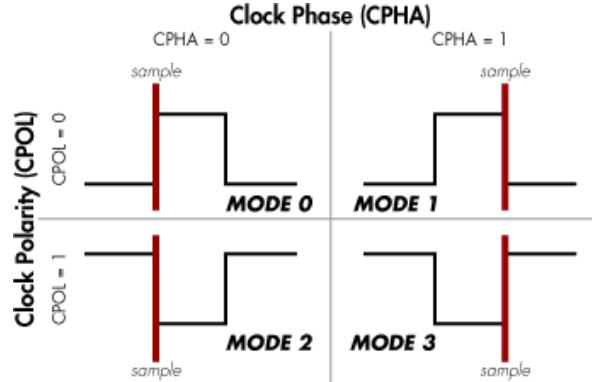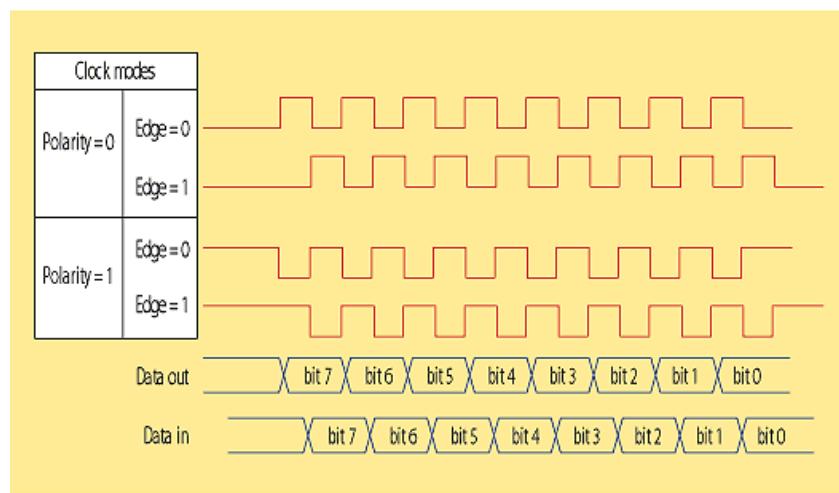

Figure 44. SPI Modes

Given that there are four distinct signals to be expected that tells the slave when the read the data and when to change, these combinations of clock phase and clock polarity have been summarized into an **SPI Mode** convention:

SPI MODE	SPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Comparing the modes and the resultant data in/out is possible with a **timing diagram**, as shown:

2. SPI can have four different clock configurations. Polarity determines the clock line's level when inactive, and the phase determines if data is clocked on the rising or falling edge.

Figure 45. SPI Timing Diagram

This type of system ensures that data comes in a stream of bits, instead of waiting on a buffer to be read. This data is often "raw" and will need to be converted into something useful, which is found on the data sheet of the slave.

The **SPI.h** library is used to communicate via SPI protocol and must be included using a compiler directive into any sketch that tries to interface with SPI devices. SPI protocol can be used to communicate between microcontrollers.

Some important functions used to interface with SPI are given below:

SPI.begin()

Purpose: Initializes the SPI bus by setting SCK, MOSI, and SS to outputs, pulling SCK and MOSI low, and SS high.

Parameters: none

SPI.end()

Purpose: Disables the SPI bus (leaving pin modes unchanged).

Parameters: none

SPI.transfer(val)

Purpose: SPI transfer is based on a simultaneous send and receive: the received data is returned in receivedVal (or receivedVal16). In case of buffer transfers the received data is stored in the buffer in-place (the old data is replaced with the data received).

Parameters: *val*: the byte to send out over the bus (or two bytes)

SPI.beginTransaction(SPISettings(speedMaximum, dataOrder, dataMode))

Purpose: Initializes the SPI bus using the defined SPISettings. The **SPISettings** object is used to configure the SPI port for your SPI device. All 3 parameters are combined to a single SPISettings object, which is given to SPI.beginTransaction(). When all of your settings are constants, SPISettings should be used directly in SPI.beginTransaction(). For constants, this syntax results in smaller and faster code. If any of your settings are variables, you may create a SPISettings object to hold the 3 settings. Then you can give the object name to SPI.beginTransaction(). Creating a named SPISettings object may be more efficient when your settings are not constants, especially if the maximum speed is a variable computed or configured, rather than a number you type directly into your sketch.

Parameters: *speedMaximum*: the maximum speed of communication. For a SPI chip rated up to 20 MHz, use 20000000.

dataOrder: MSBFIRST or LSBFIRST (specifies whether the Most Significant Bit (MSB) or Least Significant Bit) comes first in data transmission.

> *dataMode:* specifies the SPI Mode (SPI_MODE0, SPI_MODE1, SPI_MODE2, or SPI_MODE3)

`SPI.endTransaction()`

Purpose: Stop using the SPI bus. Normally this is called after de-asserting the chip select, to allow other libraries to use the SPI bus.
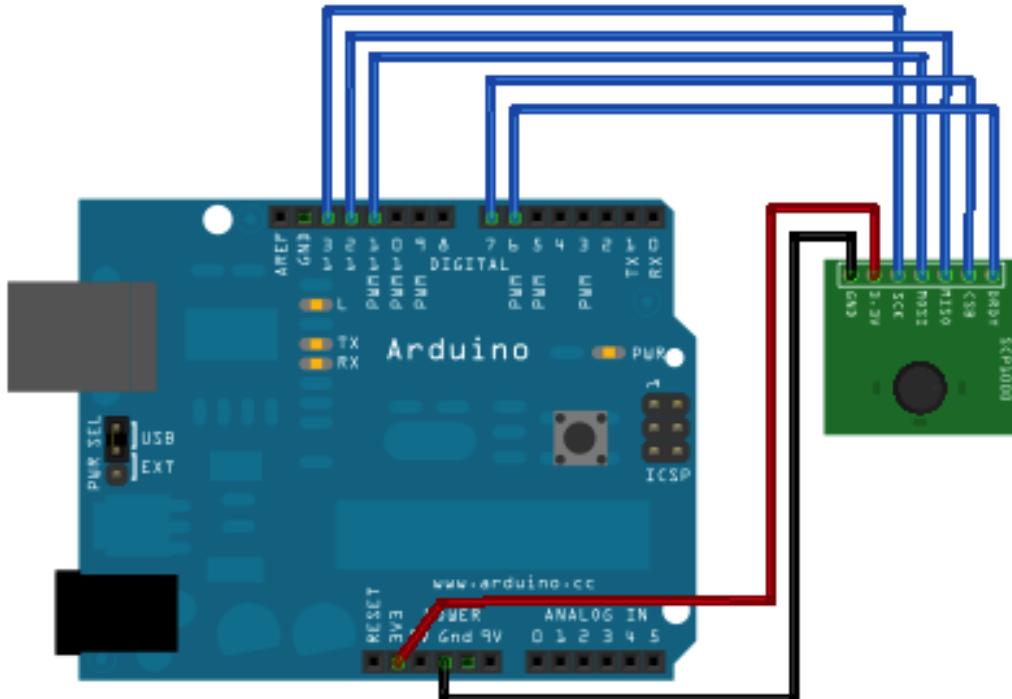
**Parameters:** none

`SPI.usingInterrupt(interruptNumber)`
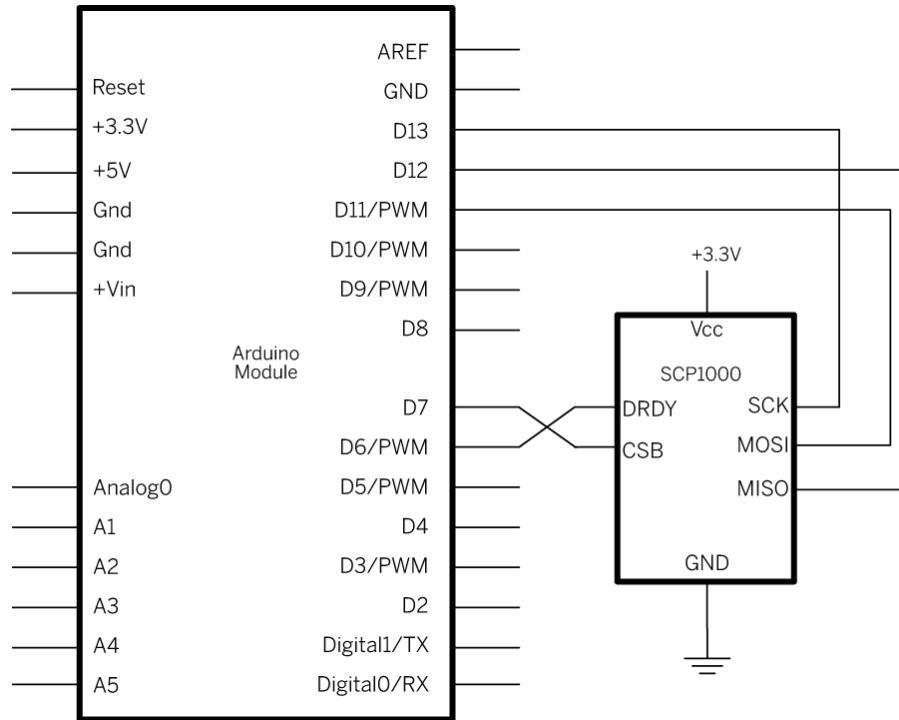
Purpose: If your program will perform SPI transactions within an interrupt, call this function to register the interrupt number or name with the SPI library. This allows `SPI.beginTransaction()` to prevent usage conflicts. Note that the interrupt specified in the call to `usingInterrupt()` will be disabled on a call to `beginTransaction()` and re-enabled in `endTransaction().`

Parameters: *interruptNumber:* the associated interrupt number.

An example for how to wire up and use a SCP1000 Barometric Sensor which uses SPI protocol on an Arduino is provided below. First, the circuit is provided showing how to wire up the barometric sensor's **breakout board**:

The actual schematic for what is happening in this circuit is shown below:

The code below starts out by setting the SCP1000's configuration registers in the `setup()`. In the main loop, it sets the sensor to read in high resolution mode, meaning that it will return a 19-bit value, for the pressure reading, and 16 bits for the temperature. The actual reading in degrees Celsius is the 16-bit result divided by 20.

Then it reads the temperature's two bytes. Once it's got the temperature, it reads the pressure in two parts. First it reads the highest three bits, then the lower 16 bits. It combines these two into one single long integer by bit shifting the high bits then using a bitwise OR to combine them with the lower 16 bits. The actual pressure in Pascal is the 19-bit result divide by 4.

```
1  /*
2  SCP1000 Barometric Pressure Sensor Display
3
4  Circuit:
5  SCP1000 sensor attached to pins 6, 7, 10 - 13:
6  DRDY: pin 6
7  CSB: pin 7
8  MOSI: pin 11
9  MISO: pin 12
10 SCK: pin 13
11 */
12
13 // the sensor communicates using SPI, so include the library:
14 #include <SPI.h>
15
16 //Sensor's memory register addresses:
17 const int PRESSURE = 0x1F;        //3 most significant bits of pressure
18 const int PRESSURE_LSB = 0x20;    //16 least significant bits of pressure
19 const int TEMPERATURE = 0x21;     //16 bit temperature reading
20 const byte READ = 0b11111100;     // SCP1000's read command
21 const byte WRITE = 0b00000010;    // SCP1000's write command
22
23 // pins used for the connection with the sensor
24 // the other you need are controlled by the SPI library):
25 const int dataReadyPin = 6;
26 const int chipSelectPin = 7;
27
28 void setup() {
29   Serial.begin(9600);
30
31   // start the SPI library:
32   SPI.begin();
33
34   // initalize the  data ready and chip select pins:
35   pinMode(dataReadyPin, INPUT);
36   pinMode(chipSelectPin, OUTPUT);
37
38   //Configure SCP1000 for low noise configuration:
39   writeRegister(0x02, 0x2D);
40   writeRegister(0x01, 0x03);
41   writeRegister(0x03, 0x02);
42   // give the sensor time to set up:
43   delay(100);
44 }
45
46 void loop() {
47   //Select High Resolution Mode
48   writeRegister(0x03, 0x0A);
49
50   // don't do anything until the data ready pin is high:
51   if (digitalRead(dataReadyPin) == HIGH) {
52     //Read the temperature data
53     int tempData = readRegister(0x21, 2);
54
55     // convert the temperature to celsius and display it:
56     float realTemp = (float)tempData / 20.0;
57     Serial.print("Temp[C]=");
58     Serial.print(realTemp);
59
60
61     //Read the pressure data highest 3 bits:
62     byte  pressure_data_high = readRegister(0x1F, 1);
63     pressure_data_high &= 0b00000111; //you only needs bits 2 to 0
64
65     //Read the pressure data lower 16 bits:
66     unsigned int pressure_data_low = readRegister(0x20, 2);
67     //combine the two parts into one 19-bit number:
68     long pressure = ((pressure_data_high << 16) | pressure_data_low) / 4;
69
70     // display the temperature:
71     Serial.println("\tPressure [Pa]=" + String(pressure));
72   }
73 }
74
```

```
75  //Read from or write to register from the SCP1000:
76  unsigned int readRegister(byte thisRegister, int bytesToRead) {
77    byte inByte = 0;            // incoming byte from the SPI
78    unsigned int result = 0;   // result to return
79    Serial.print(thisRegister, BIN);
80    Serial.print("\t");
81    // SCP1000 expects the register name in the upper 6 bits
82    // of the byte. So shift the bits left by two bits:
83    thisRegister = thisRegister << 2;
84    // now combine the address and the command into one byte
85    byte dataToSend = thisRegister & READ;
86    Serial.println(thisRegister, BIN);
87    // take the chip select low to select the device:
88    digitalWrite(chipSelectPin, LOW);
89    // send the device the register you want to read:
90    SPI.transfer(dataToSend);
91    // send a value of 0 to read the first byte returned:
92    result = SPI.transfer(0x00);
93    // decrement the number of bytes left to read:
94    bytesToRead--;
95    // if you still have another byte to read:
96    if (bytesToRead > 0) {
97      // shift the first byte left, then get the second byte:
98      result = result << 8;
99      inByte = SPI.transfer(0x00);
100     // combine the byte you just got with the previous one:
101     result = result | inByte;
102     // decrement the number of bytes left to read:
103     bytesToRead--;
104   }
105   // take the chip select high to de-select:
106   digitalWrite(chipSelectPin, HIGH);
107   // return the result:
108   return (result);
109 }
110
111
112 //Sends a write command to SCP1000
113
114 void writeRegister(byte thisRegister, byte thisValue) {
115
116   // SCP1000 expects the register address in the upper 6 bits
117   // of the byte. So shift the bits left by two bits:
118   thisRegister = thisRegister << 2;
119   // now combine the register address and the command into one byte:
120   byte dataToSend = thisRegister | WRITE;
121
122   // take the chip select low to select the device:
123   digitalWrite(chipSelectPin, LOW);
124
125   SPI.transfer(dataToSend); //Send register location
126   SPI.transfer(thisValue);  //Send value to record into register
127
128   // take the chip select high to de-select:
129   digitalWrite(chipSelectPin, HIGH);
130 }
```

I²C Signals

The main inconvenience with SPI communication is that it requires four wires for each slave device. As a user starts to incorporate multiple sensors into an embedded program, the hardware can become tedious to keep up with. This is the main motivation behind the development of **I²C** communication, which requires only two wires for interfacing with any number of sensors. I²C is not as fast as SPI, but is still much faster than typical UART data transmission, usually finding data exchange rates between 100 kHz and 50 MHz. Communication, as stated, is possible through two lines: the **Serial Clock (SCL)** and **Serial Data (SDA)**. Both of these lines are connected to both the microcontroller and the sensor and require pull-up resistors, because I²C devices can only pull voltage down to LOW. The general schematic for this protocol is demonstrated below:
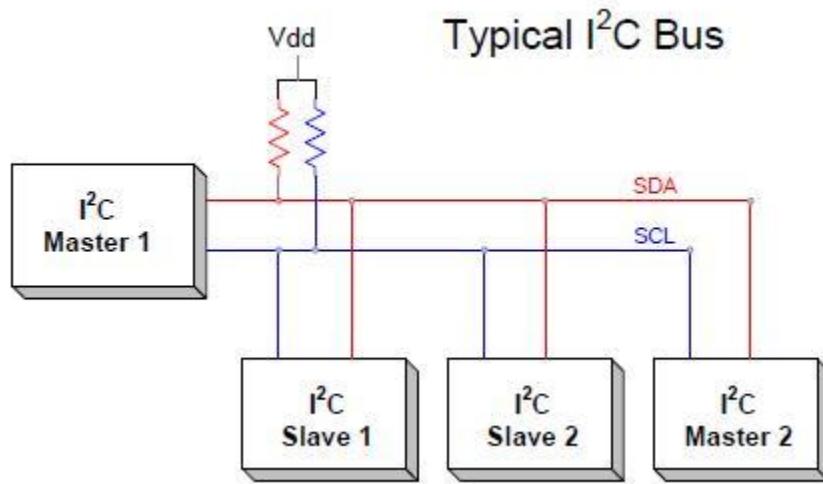

Figure 46. I2C Basic Wiring Diagram

Like with SPI, two sensors cannot communicate at the same time and thus require some method of talking Unlike in SPI, which hardwires a Chip Select pin and wire to select which sensor is being referenced at any point in time, I²C references the slave chips by sending a unique 7-bit address down the SDA line in order to determine which sensor is "speaking". These addressed can be reserved (for a cost), so for convenience these integrated circuits usually have a hardwired adjustable address (with 2 – 8 unique address numbers).

To send a message from the master to the slave, the following procedure is undergone:

1. Master starts: sends address + **W (Write Bit)**
2. Waits for slave with indicated address **ACK (Acknowledge Bit)**
3. Master sends which register(s) to change to slave with the indicated address
4. Master sends data that is wanted into the register

To accept a message from a slave (read it into the microcontroller):

1. Master starts: sends address + **R (Read Bit)**
2. Waits for slave with indicated address ACK
3. Master sends register that is to be read to slave with indicated address
4. Slave returns the data to master

The general timing diagram will take the following form:
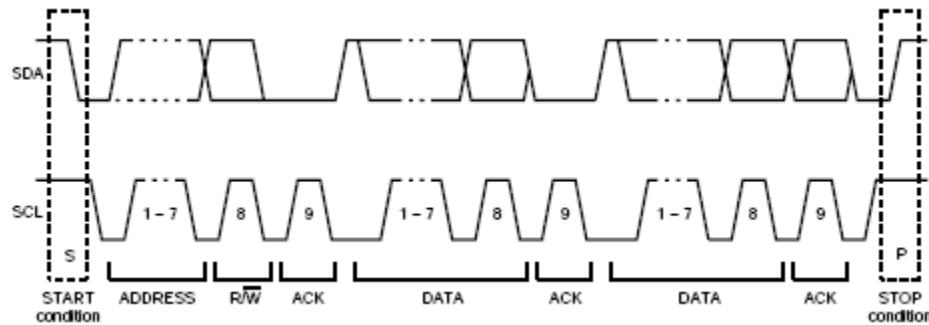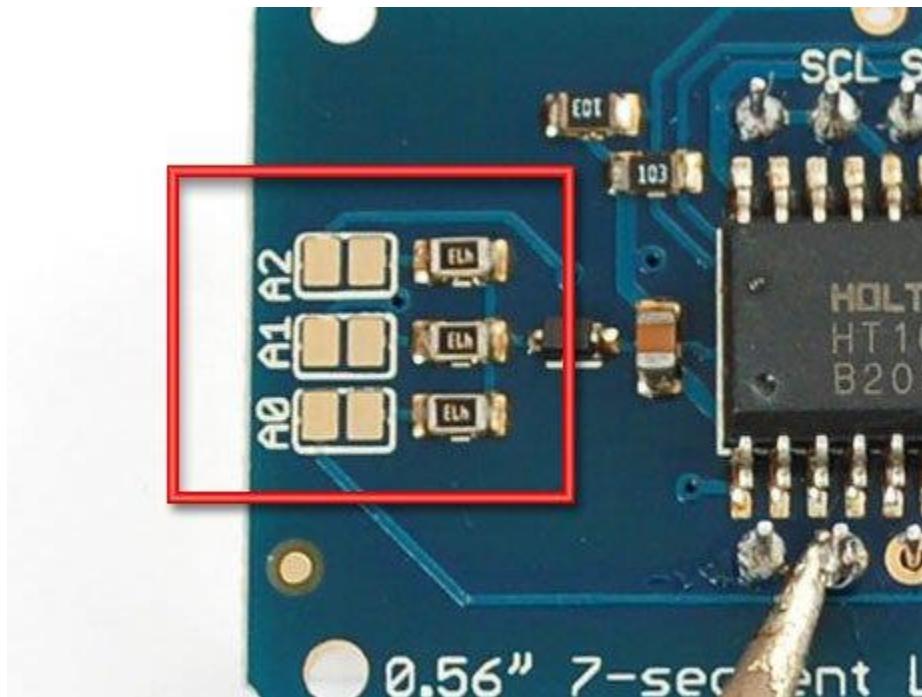
Figure 47. I2C Timing Diagram

To set a chip's adjustable address, some knowledge of the chip is required. Often, specific pins labelled A0, A1, … will be present on the chip and depending on whether those pins are held HIGH or LOW, the address changes. For example, consider the HT16K33 driver chip with the following solder pads:

Should solder connect the two pads, the input will be considered HIGH, and if not, the pin will be considered LOW. The default address of this device is 0x70 (addresses are frequently represented in hexadecimal to save space). Changing the address of this device is fairly simple. Look on the back to find the two or three A0, A1 or A2 solder jumpers. Each one of these is used to hardcode in the address. If a jumper is shorted with solder, that sets the address. A0 sets the lowest bit with a value of 1, A1 sets the middle bit with a

value of 2 and A2 sets the high bit with a value of 4. The final address is 0x70 + A2 + A1 + A0. So for example if A2 is shorted and A0 is shorted, the address is 0x70 + 4 + 1 = 0x75. If only A1 is shorted, the address is 0x70 + 2 = 0x72.

I^{2}C protocol is easily implemented in microcontrollers through the **Wire.h** library. Several helpful functions exist within this library (much like with the SPI library), some of the more important ones are talked about here:

`Wire.begin(address)`

Purpose: Initiate the Wire library and join the I2C bus as a master or slave. This should normally be called only once.

Parameters: *address:* the 7-bit slave address (optional); if not specified, join the bus as a master.

`Wire.requestFrom(address, quantity, stop)`

Purpose: Used by the master to request bytes from a slave device. The bytes may then be retrieved with the `available`() and `read`() functions. As of Arduino 1.0.1, `requestFrom`() accepts a boolean argument changing its behavior for compatibility with certain I2C devices. If true, `requestFrom`() sends a stop message after the request, releasing the I2C bus. If false, `requestFrom`() sends a restart message after the request. The bus will not be released, which prevents another master device from requesting between messages. This allows one master device to send multiple requests while in control. The default value is true.

Parameters: *address*: the 7-bit address of the device to request bytes from

quantity: the number of bytes to request

stop: boolean value, true will send a stop message after the request, releasing the bus. false will continually send a restart after the request, keeping the connection active.

`Wire.read()`

Purpose: Returns the number of bytes available for retrieval with `read()`. This should be called on a master device after a call to `requestFrom()` or on a slave inside the `onReceive()` handler.

Parameters: none

`Wire.write(val, length)`

Purpose: Writes data from a slave device in response to a request from a master, or queues bytes for transmission from a master to slave device (in-between calls to `beginTransmission()` and `endTransmission()`).

Parameters: *val*: value to send as a single byte, string, or an array of data to send as several bytes

length: the number of bytes to transmit if an array is being written (no argument needed otherwise)

`Wire.beginTransmission(address)`

Purpose: Begin a transmission to the I²C slave device with the given address. Subsequently, queue bytes for transmission with the `write()` function and transmit them by calling `endTransmission()`.

Parameters: *address:* the 7-bit slave address to transmit to

`Wire.endTransmission(stop)`
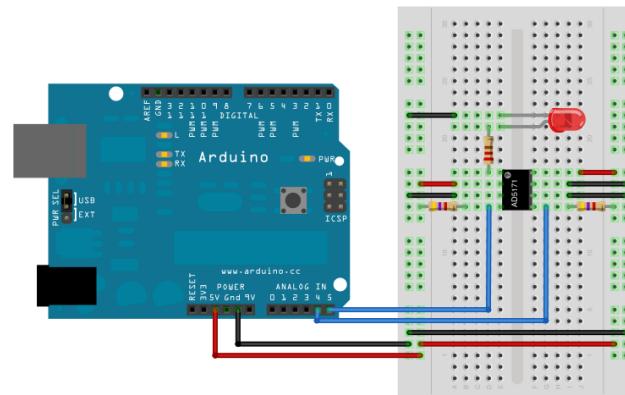
Purpose: Ends a transmission to a slave device that was begun by `beginTransmission()` and transmits the bytes that were queued by `write()`. As of Arduino 1.0.1, `endTransmission()` accepts a boolean argument changing its behavior for compatibility with certain I2C devices. If true, `endTransmission()` sends a stop message after transmission, releasing the I2C bus. If false, `endTransmission()` sends a restart message after transmission. The bus will not be released, which prevents another master device from transmitting between messages. This allows one master device to send multiple transmissions while in control. The default value is true.

The returned byte will be:
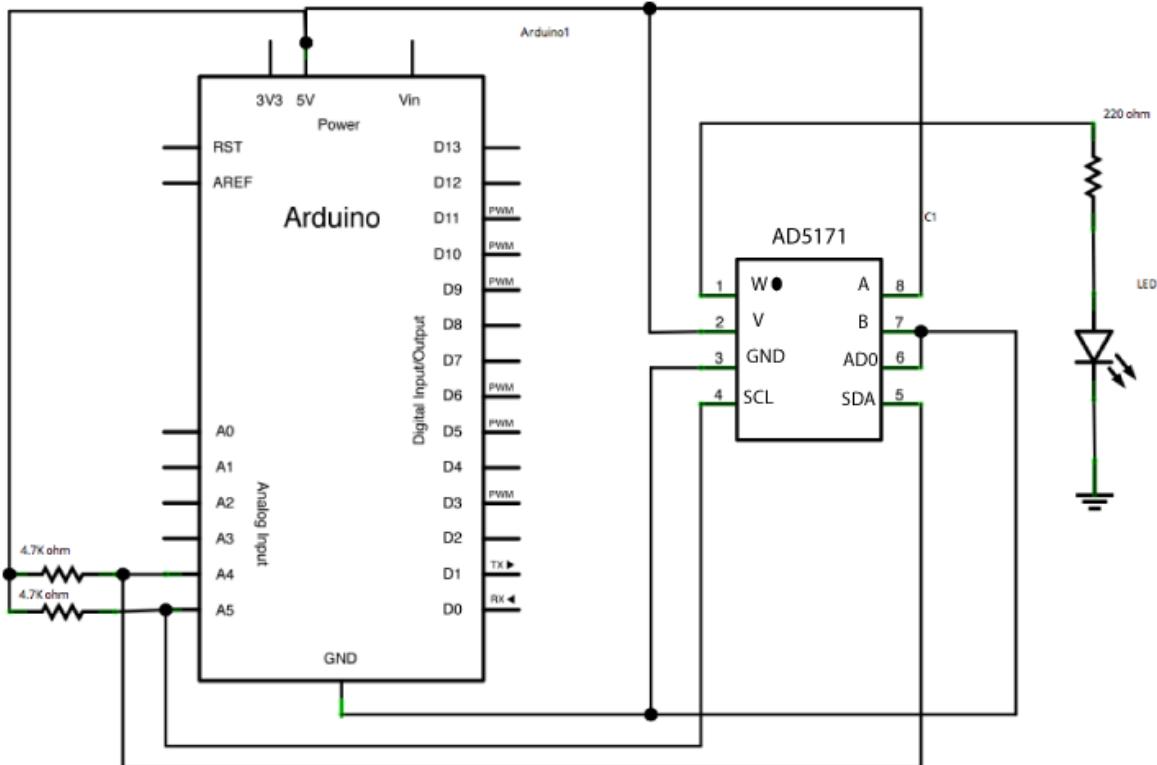
- 0: success
- 1: data too long to fit in transmit buffer
- 2: received NACK on transmit of address
- 3: received NACK on transmit of data
- 4: other error

Parameters: *stop:* boolean. true will send a stop message, releasing the bus after transmission. false will send a restart, keeping the connection active.

A simple example showing how to use I2C protocol is the reading of a digital potentiometer. The AD5171 Digital Potentiometer can be wired in the following configuration so that the potentiometer reading impacts the brightness of some LEDs:

The schematic of this circuit is shown below:

When the AD5171's pin 6, ADO, is connected to ground, it's address is 44. To add another digital pot to the same SDA bus, connect the second pot's ADO pin to +5V, changing it's address to 45. To interface with the potentiometer and light up the LED, the following code could be used:

```
#include <Wire.h>

void setup() {
  Wire.begin(); // join i2c bus (address optional for master)
}

byte val = 0;

void loop() {
  Wire.beginTransmission(44); // transmit to device #44 (0x2c)
  // device address is specified in datasheet
  Wire.write(byte(0x00));           // sends instruction byte
  Wire.write(val);             // sends potentiometer value byte
  Wire.endTransmission();     // stop transmitting

  val++;        // increment value
  if (val == 64) { // if reached 64th position (max)
    val = 0;    // start over from lowest value
  }
  delay(500);
}
```

CAN Bus

The **CAN (controller area network) bus** is a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other within a vehicle without a host computer. This bus standard includes its own messaging protocol for communications between nodes on the network. This standard was invented for the interfacing of sensors in an automobile, where there are several systems that need to interact with each other, and each have a certain priority over others for how fast they need information.
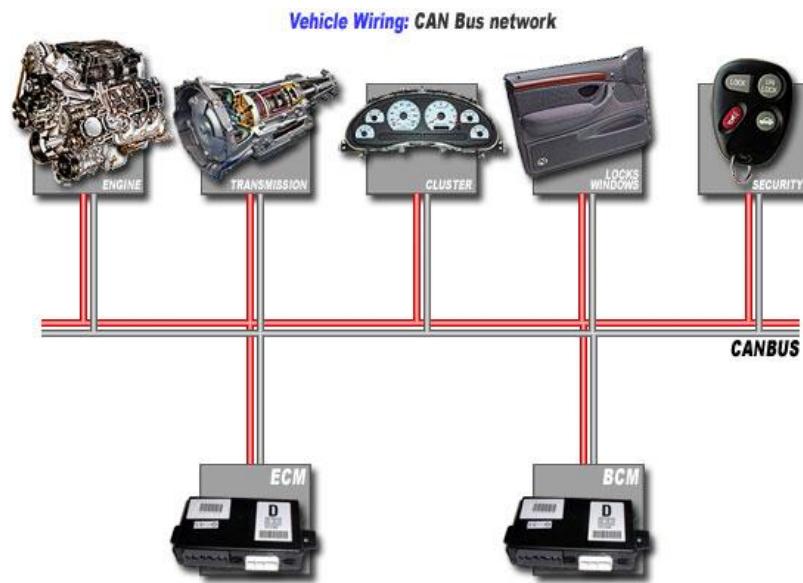

Figure 48. Example CAN Bus

The CAN bus (Controller Area Networking) was defined in the late 1980 by Bosch, initially for use in automotive applications (CAN 2.0). It has been found to be very useful in a wide variety of distributed industrial systems. A 2014 enhancement to the specifications (CAN FD) improves the performance of CAN. Generally speaking, CAN has the following characteristics:

- Uses a single terminated twisted pair cable to transmit data
- Multiple masters can exist on a single CAN Bus
- Highly reliable with extensive error checking
- Maximum Signal frequency used is 1 Mbit/sec (CAN 2.0) , 15 Mbits/sec (CAN FD)
- Typical maximum data rate achievable is 320 kBits/sec for CAN 2.0 and 3.7 MBits/sec for CAN FD
- Maximum latency of high priority message <120 µsec at 1Mbit/sec
- The faster the data transmission, the shorter the available conduit.
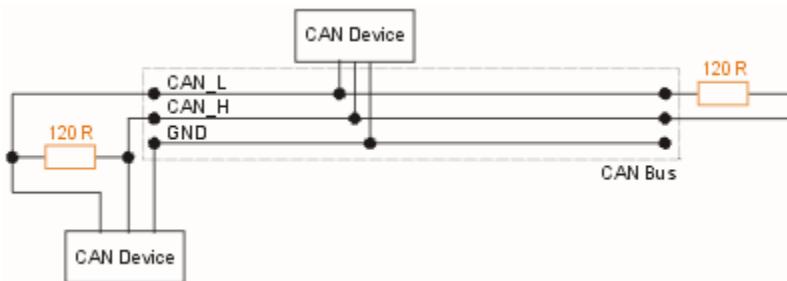
CAN is unusual in that the entities on the network, called nodes, are not given specific addresses. Instead, it is the messages themselves that have an identifier which also determines the messages' priority. Nodes then depending on their function transmit specific messages and look for specific message. For this reason there is no theoretical limit to the number of nodes although in practice it is ~64. Note that no two nodes can transmit the same message ID as this violates the priority rules.

Three specifications are in use:

- **CAN 2.0A** sometimes known as **Basic or Standard CAN** with 11 bit message identifiers which was originally specified to operated at a maximum frequency of 250Kbit/sec and is ISO11519.
- **CAN 2.0B** known as **Full CAN** or extended frame CAN with 29 bit message identifier which can be used at up to 1Mbit/sec and is ISO 11898.
- **CAN FD** increases the max data throughput to ~ 3.7 Mbits/sec. It does this by retaining much of the 2.0 packet structure (which it is compatible with) but using one reserved bit to indicate that the data part of the packet is using the new standard. Once an FD enabled device or interface detects this it can do two things..... Transmits/receives the data part at a secondary frequency of up to 12 Mbits/sec (v 1Mbits/sec for CAN 2.0) and also it allows the data part of the package to consist of up to 64 bytes (v 8 bytes for CAN 2.0). For more details see **CAN FD**.

CAN may be implemented over a number of physical media so long as the drivers are open-collector and each node can hear itself and others while transmitting (this is necessary for its message priority and error handling mechanisms). The most common media is a twisted pair 5V differential signal which will allow operations in high noise environments, and with the right drivers will work even if one of the wires is open circuit. A number of transceiver chips are available the most popular probably being the Philips 82C251 as well as the TJA1040.

When running Full CAN (ISO 11898-2) and CAN FD at its higher speeds it is necessary to terminate the bus at both ends with 120 Ohms. The resistors are not only there to prevent reflections but also to unload the open collector transceiver drivers. The bus must always be terminated correctly:

The Teensy 3.1 and 3.2 can join a CAN Bus through their designated pins and use of the FlexCan.h library. The relevant functions and descriptions can be found at the bottom of that page.

OneWire Bus

An additional data protocol comes in the form of **OneWire (1-Wire)** which actually uses just one wire for both data transmission and the clock. Dallas Semiconductor (now Maxim) produces a family of devices that are controlled through a proprietary 1-wire protocol. There are no fees for programmers using the Dallas 1-Wire (trademark) drivers.

On a 1-Wire network, which Dallas has dubbed a *MicroLan* a single master device communicates with one or more 1-Wire slave devices over a single data line (the 1-Wire Bus), which can also be used to provide power to the slave devices (meaning that the only two lines fed to a sensor are data and ground in what is called *parasitic power mode*). An example of such a device is the MAX31850 Thermocouple Breakout Board for reading K type thermocouples. For longer stretches of wire between the microcontroller and the sensor, parasitic power mode is less likely to give reliable results.

When operating in parasitic power mode, only two wires are required: one data wire, and ground. In this mode, the power line must be connected to ground, per the datasheet. **At the master, a 4.7k pull-up resistor must be connected to the 1-wire bus.** When the line is in a HIGH state, the device pulls current to charge an internal capacitor. This current is usually very small, but may go as high as 1.5 mA when doing a temperature conversion or writing EEPROM. When a slave device is performing one these operations, the bus master must keep the bus pulled HIGH to provide power until the operation completes; a delay of 750ms is required for a DS18S20 temperature conversion. The master cannot do anything during this time, like issuing commands to other devices, or polling for the slave's operation to be completed. To support this, the OneWire.h library makes it possible to have the bus held HIGH after the data is written.

With an external supply, three wires are required: the bus wire, ground, and power. **The 4.7k pull-up resistor is still required on the bus wire.** As the bus is free for data transfer, the microcontroller can continually poll the state of a device doing a conversion. This way, a conversion request can finish as soon as the device reports being done, as opposed to having to wait for conversion time (dependent on device function and resolution) in parasitic power mode.

The functions included with OneWire.h are:

`OneWire myWire(pin)`

Purpose: Create the `OneWire` object, using a specific pin. Even though you can connect many 1 wire devices to the same pin, if you have a large number, smaller groups each on their own pin can help isolate wiring problems. You can create multiple `OneWire` objects, one for each pin.

Parameters: *pin:* the pin for the `OneWire` Bus

`myWire.search(addrArray)`

Purpose: Search for the next device. The `addrArray` is an 8 byte array. If a device is found, `addrArray` is filled with the device's address and true is returned. If no more devices are found, false is returned.

Parameters: *addrArray:* 8 byte array representing the OneWire address of the current device.

`myWire.`reset_search`()`

Purpose: Begin a new search. The next use of search will begin at the first device.

Parameters: *none*

`myWire.`reset`()`

Purpose: Reset the 1-wire bus. Usually this is needed before communicating with any device.

Parameters: *none*

`myWire.`select`(addrArray)`

Purpose: Select a device based on its address. After a reset, this is needed to choose which device you will use, and then all communication will be with that device, until another reset.

Parameters: *addrArray:* 8 byte array representing the OneWire address of the selected device.

`myWire.`skip`()`

Purpose: Skip the device selection. This only works if you have a single device, but you can avoid searching and use this to immediatly access your device.

Parameters: *none*

`myWire.`write`(num,1)`

Purpose: Writes a byte of data to the selected device. The second argument is optional but if entered as "1", the OneWire bus will leave power applied to it after writing.

Parameters: *num:* the byte of data to be written

`myWire.`read`()`

Purpose: Reads a byte of data from the selected device

Parameters: *none*

`myWire.`crc8`(dataArray, length)`

Purpose: Compute a [CRC (Cyclic Redundancy Check)](#) on an array of data to detect data corruption.

Parameters: *dataArray:* the data to be checked

   *length*: the length bit for the CRC check
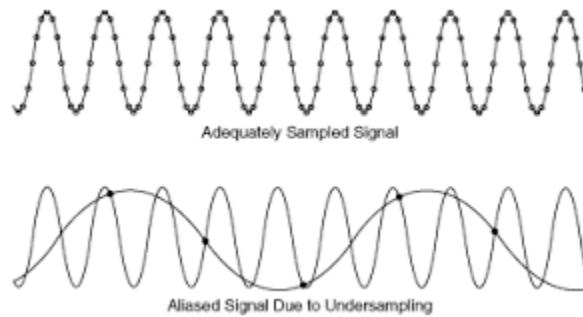
24.   Digital Filters

All filters have a digital equivalent that can be implanted in code without the hardware required of a RC Filter or RLC Filter. The general idea behind the construction of a filter is to eliminate unwanted aspects of an input signal, which is generally noise. Noise is typically represented as uniformly represented, randomly distributed data (**Gaussian distribution**). Analog filters focus on elimination of frequencies, whereas digital filters focus on the variance of a signal from the mean.

The ideal filter will eliminate noise from the system by following some basic principles:

1. Data will fall inside of a range
2. Data has a maximum speed in which it can change
3. Data comes in at the right time (no phase lag or phase lead)
4. Data is based on the physics that are expected.

A microcontroller, when sampling data and discretizing it via an ADC, generates an approximate waveform to the initial signal. It is important to note that the speed at which a signal is sampled should be fast enough so that the signal is not **aliased**. An aliased signal is one that because it is not sampled frequently enough, looks like a signal completely different than the one that was intended to be sampled.

Adequately Sampled Signal

Aliased Signal Due to Undersampling

To prevent aliasing, the signal should be sampled at least twenty times the signal frequency. (**Nyquist theory** dictates that an absolute minimum is sampling twice as fast as the signal's frequency).

Discrete Low Pass Filter

A simple first order low pass filter can be obtained for recorded measurement y_k given new raw sensor measurement x_k by assuming the complex system can be modelled as a simple one with first order dynamics:

$$\dot{y} + \lambda y = \lambda x$$

The transfer function of such a simple system would appear as:

$$\frac{Y(s)}{X(s)} = \frac{\lambda}{s + \lambda}$$

λ being placed on the right hand side of the equation makes it so that the "DC Gain" (or frequency response as the frequency of the input signal moves towards 0) is unity (one). Another way to view this is that the step response magnitude will be unity. λ denotes the pole of the transfer function, and in this way is the reciprocal of the simple system's time constant (τ).

In the digital microcontroller, assume the sample rate (T_s) is much greater than the system bandwidth (ω_b) and then simulate the simple model with the raw measurement as the input. In other words, find the relationship $y_k = f(x_k, y_{k-1})$ from $\dot{y} = \lambda x - \lambda y$. It is found that Euler's method of integration provides the basis of this relationship such that:

$$y_k = y_{k-1} + T_s \dot{y}_{k-1}$$

$$y_k = (T_s \lambda) x_k + (1 - T_s \lambda) y_{k-1}$$

$T_s \lambda$ is required to be a number greater than 0 but less than 1 in order for this scheme to be stable. Denoting $\alpha = T_s \lambda$ (sample time multiplied by cutoff frequency of the filter), we arrive at:

$$y_k = \alpha x_k + (1 - \alpha) y_{k-1}$$

A low pass filter is an example of an **IIR Filter (Infinite Impulse Response).** This means that such a filter exponentially approaches the true value, but can never quite reach the true value. Other examples include **Butterworth filters** (which have maximally flat frequency response in the passband), **Bessel filters** (which have maximally linear phase response), and **Chebyshev filters** (minimal error between ideal and actual filter characteristics in the passband). To move these filters into the discrete domain is often mathematically rigorous and often not worth the time. An alternative to these types of filters are the nonlinear **FIR (Finite Impulse Response)** filters, which are fairly intuitive and easy to use:

Moving Average Filter

A moving average filter simply takes n samples and averages them for the recorded measurement y_k.

$$y_k = \frac{x_k + x_{k-1} + \cdots + x_{k-n}}{n}$$

This type of filter tends to attenuate some frequencies better than others.

Over-Sampling

If the ADC speed is much greater than the sample time and the signal noise is greater than the discretization error, then a helpful approach is to quickly sample the measurement several times and use a single recorded measurement as the average of these rapid samples. This would be given by:

$$y_k = \frac{x_k^{(1)} + x_k^{(2)} + \cdots + x_k^{(n)}}{n}$$

The **ADC.h** library includes a series of functions to be called in the `setup()` loop, among them is `setAveraging(num_avg)`, which gives the ability to easily over-sample data. The ADC is also by default setup to oversample, with the number of samples taken being 4 for the Teensy 3.2.

This filters noise and effectively increases the resolution of the ADC is there is enough signal noise. In some cases, noise can be intentionally added to ensure that this works ("**dithering**"). One way to add uniformly distributed noise is to simply heat up the circuit (see Nyquist-Johnson noise which states that noise is proportional to resistance and temperature). Another approach is to use random numbers as noise in software (see this page on details on the `random()` function in Arduino)

Median Filter

A median filter simply rejects noise by taking the median of n samples. This simple filter can work quite well and is given by:

$$y_k = \text{median}(x_k, x_{k-1}, \ldots, x_{k-n})$$

This method rejects large excursions with a single sample delay. A median filter can be constructed by storing measurements in an array and using a bubblesort algorithm to sort the values in order from highest to lowest, then taking the middle value of that array.

Velocity Filters

A velocity filter restricts how quickly parameters can change in time, typically based on some physical knowledge of the actual model being measured. One way to represent this would be to use the maximum increment of change as *dy* and then using max() and min() as bounds:

$$y_k = \max(y_k - dy, \min(y_k + dy, x_k))$$

NOTE: There exist several complex filtering schemes not covered here, notably Kalman filters. This is because these filters begin to become more advanced and require knowledge and mathematics beyond the scope of this document. To read up more on Kalman filters in particular, An Introduction to Kalman Filters by Greg Welch and Gary Bishop is a good place to start. For the sake of this document, it should be noted that unless the noise in the system is known in a statistical sense and can be included in a robust system model, then a Kalman filter is not the filter to use.

25. Creating Your Own C++ Libraries

A **C++ class** is a user made library that gives an Arduino sketch a "hidden toolbox" that is easy to access. Classes provide two main benefits to the user:

1. Protect **methods** (functions) from user (functions in a class are termed methods)

2. Make generalized methods instead of writing new functions (i.e. a class can be created that debounces a button instead of writing a new function for each button that needs to be debounced)

A class consists of two portions: the header file (.h) and the C++ source (.cpp). These files must have the same name and be in the same directory. The header file is typically used for initializing the class and prototyping all methods and variables to be used within the class. These methods and variables can either be **public** (can be referenced in the main sketch) or **private** (only referenced from within the class itself). As a matter of consistency, private parameters are usually prefaced with an underscore ("_"). Using the simple example of using the LED to blink Morse code, the syntax for the header file would be:

```cpp
#include "Arduino.h"

class Morse
{
  public:
    Morse(int pin);
    void dot();
    void dash();
  private:
    int _pin;
};
```

A class is simply a collection of functions and variables that are all kept together in one place. These functions and variables can be public, meaning that they can be accessed by people using your library, or private, meaning they can only be accessed from within the class itself. Each class has a special function known as a constructor, which is used to create an instance of the class. The constructor has the same name as the class, and no return type.

Now, the source file could look like:

```cpp
#include "Arduino.h"
#include "Morse.h"

Morse::Morse(int pin)
{
  pinMode(pin, OUTPUT);
  _pin = pin;
}

void Morse::dot()
{
  digitalWrite(_pin, HIGH);
  delay(250);
  digitalWrite(_pin, LOW);
  delay(250);
}

void Morse::dash()
{
  digitalWrite(_pin, HIGH);
  delay(1000);
  digitalWrite(_pin, LOW);
  delay(250);
}
```

First comes a couple of #include statements. These give the rest of the code access to the standard Arduino functions, and to the definitions in your header file. Then comes the constructor. Again, this explains what should happen when someone creates an instance of your class. In this case, the user specifies which pin they would like to use. The pin is configured as an output save it into a private variable for use in the other functions. There are a couple of strange things in this code. First is the **Morse::** before the name of the function. This says that the function is part of the Morse class. You'll see this again in the other methods in the class. The second unusual thing is the underscore in the name of our private variable, _pin. This variable can actually have any name you want, as long as it matches the definition in the header file. Adding an underscore to the start of the name is a common convention to make it clear which variables are private, and also to distinguish the name from that of the argument to the function (pin in this case).

To incorporate this into a sketch, just be sure to include the newly created library:

```
#include <Morse.h>

Morse morse(13);

void setup()
{
}

void loop()
{
  morse.dot(); morse.dot(); morse.dot();
  morse.dash(); morse.dash(); morse.dash();
  morse.dot(); morse.dot(); morse.dot();
  delay(3000);
}
```

26.   Rotary Encoders

A **rotary encoder** is one of the most common means to measure the position and rotational velocity of a shaft. There are two main types of encoders: absolute encoders and incremental encoders. An absolute encoder will remember the position of the shaft after losing power and measures the position to some common reference point on every run, whereas an incremental encoder will simply keep track of the motion once the program begins and will not store the information for later use.

Figure 49. Incremental Rotary Encoder

An incremental rotary encoder provides cyclical outputs (only) when the encoder is rotated. They can be either mechanical, optical or magnetic. The mechanical type requires debouncing and is typically used as digital potentiometers on equipment including consumer devices. Most modern home and car stereos use mechanical rotary encoders for volume control. Due to the fact the mechanical switches require debouncing, the mechanical type are limited in the rotational speeds they can handle. The incremental rotary encoder is the most widely used of all rotary encoders due to its low cost and ability to provide signals that can be easily interpreted to provide motion related information such as velocity.

The fact that incremental encoders use only two sensors does not compromise their resolution. One can find in the market incremental encoders with up to 10,000 counts per revolution, or more.
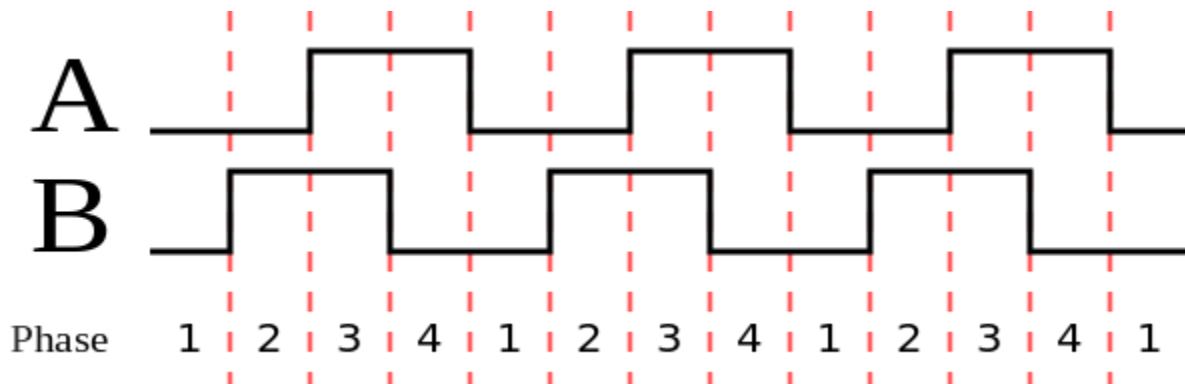
There can be an optional third output: reference or "index", which happens once every turn. This is used when there is the need of an absolute reference, such as positioning systems. The index output is usually labeled Z. The optical type is used when higher speeds are encountered or a higher degree of precision is required. Incremental encoders are used to track motion and can be used to determine position and velocity. This can be either linear or rotary motion. Because the direction can be determined, very accurate

measurements can be made. They employ two outputs called **A** and **B**, which are called **quadrature outputs**, as they are 90 degrees out of phase. The state diagram for these outputs is given by:

Coding for clockwise rotation				Coding for counter-clockwise rotation		
Phase	A	B		Phase	A	B
1	0	0		1	1	0
2	0	1		2	1	1
3	1	1		3	0	1
4	1	0		4	0	0

The two output wave forms are 90 degrees out of phase, which is what quadrature means. These signals are decoded to produce a count up pulse or a count down pulse. For decoding in software, the A and B outputs are read by software, either via an interrupt on any edge or polling, and the above table is used to decode the direction. For example, if the last value was 00 and the current value is 01, the device has moved one half step in the clockwise direction. The resolution of the encoder would then reveal how far this half-step is in degrees of rotation. The mechanical types would be debounced first by requiring that the same (valid) value be read a certain number of times before recognizing a state change.

An example encoder output for clockwise rotation is given by:

A

B

Phase   1   2   3   4   1   2   3   4   1   2   3   4   1

The **Encoder.h** library is extremely useful for reading data from a rotary encoder. For best performance with this library, both outputs A and B should be connected to interrupt pins (and on the Teensy 3.2 all digital pins have interrupt capability). It can be used (albeit with some sacrifice in performance) if only one pin has interrupt ability, or if neither of them do.

The basic usage of the library is covered with the following expressions:

`Encoder myEnc(pin1, pin2)`

Purpose: Create an Encoder object `myEnc`, using 2 pins. You may create multiple Encoder objects, where each uses its own 2 pins. The first pin should be capable of interrupts. If both pins have interrupt capability, both will be used for best performance. Encoder will also work in low performance polling mode if neither pin has interrupts.

Parameters: *pin1:* Channel A output

　　　　　pin2: Channel B output

`myEnc.read()`

Purpose: Returns the accumulated position as an integer. This number can be positive or negative.

Parameters: *none*

`myEnc.write(newPosition)`

Purpose: Set the new accumulated position as an integer. This is useful for zeroing out the encoder reading at any stage in the code.

Parameters: *newPosition:* new `myEnc` position as an integer

Some example code is provided below that returns the position of two knobs connected to the microcontroller:

```
/* Encoder Library - TwoKnobs Example
 * http://www.pjrc.com/teensy/td_libs_Encoder.html
 *
 * This example code is in the public domain.
 */

#include <Encoder.h>

// Change these pin numbers to the pins connected to your encoder.
//   Best Performance: both pins have interrupt capability
//   Good Performance: only the first pin has interrupt capability
//   Low Performance:  neither pin has interrupt capability
Encoder knobLeft(5, 6);
Encoder knobRight(7, 8);
//   avoid using pins with LEDs attached

void setup() {
  Serial.begin(9600);
  Serial.println("TwoKnobs Encoder Test:");
}

long positionLeft  = -999;
long positionRight = -999;

void loop() {
  long newLeft, newRight;
  newLeft = knobLeft.read();
  newRight = knobRight.read();
  if (newLeft != positionLeft || newRight != positionRight) {
    Serial.print("Left = ");
    Serial.print(newLeft);
    Serial.print(", Right = ");
    Serial.print(newRight);
    Serial.println();
    positionLeft = newLeft;
    positionRight = newRight;
  }
  // if a character is sent from the serial monitor,
  // reset both back to zero.
  if (Serial.available()) {
    Serial.read();
    Serial.println("Reset both knobs to zero");
    knobLeft.write(0);
    knobRight.write(0);
  }
}
```

27.   Load Cells

A **load cell** is a device that can measure the tensile or compressive forces on it. Often this is accomplished via the use of a **Wheatstone Bridge**. A Wheatstone bridge is an electrical circuit used to measure an unknown electrical resistance by balancing two legs of a bridge circuit, one leg of which includes the unknown component as shown (Figure 50).
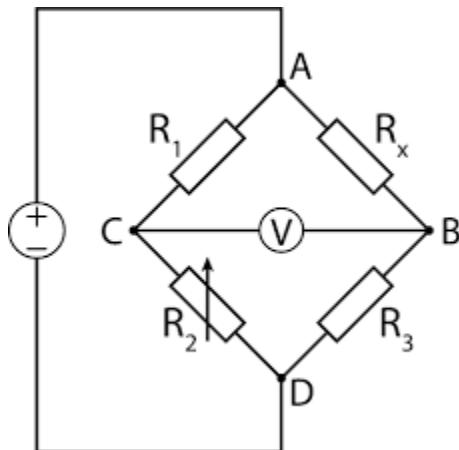
Figure 50. Wheatstone Bridge Schematic

The main idea is that if the two resistances on the left side of the circuit are equivalent to the two resistances on the right side of the circuit, then no current will pass through the center of the circuit (as current will only want to pass through the path of least resistance). Therefore, by adjustment of the resistor R_2 the resistance of R_x can be found. The primary benefit of a Wheatstone bridge is its ability to provide extremely accurate measurements.

With a load cell, this unknown resistance varies with the load linearly, and the measurement of this resistance reveals the load after calibration. Load cells come in a variety of different forms, and can be set up to be mounted in several ways and be set up for significantly different loads. A very small example of this wide variety are displayed below (Figure 51):
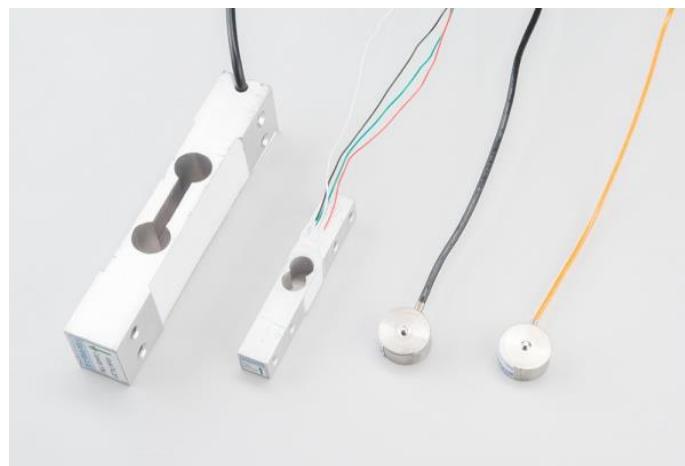
Figure 51. Example of Load Cell Appearances

A **load cell amplifier**, such as the HX711 is compatible with any such load cell that utilizes a bridge and uses SPI protocol for communication. The attached link has links to the library, which has functions for taring the load cell and calibrating the output. The amplifier takes in four inputs from the load cell (the two excitation voltages and the load cell output).
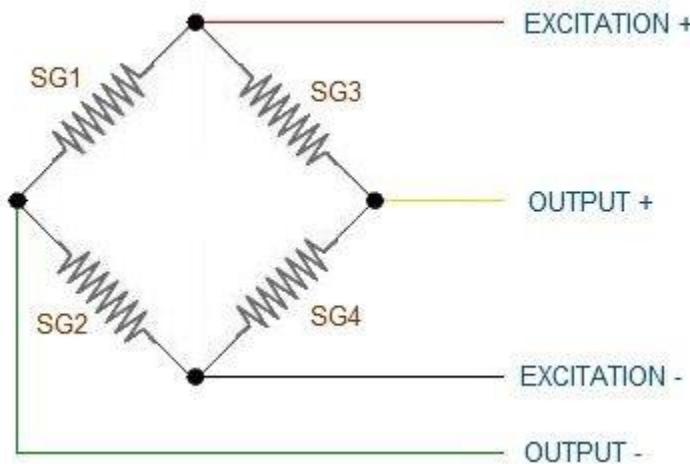
Figure 52. Load Cell Wiring

Because this is a load cell, each of the resistors in the Wheatstone Bridge is actually a **strain gauge**. Knowledge of the voltage of all four locations reveals everything about the loading of the load cell because the resistance of each strain gauge is known. The convention for wire coloring coming off a load cell can vary greatly, but generally follows a format:

- Excitation+ (E+) or VCC is red
- Excitation- (E-) or ground is black.
- Output+ (O+), Signal+ (S+)+ or Amplifier+ (A+) is white
- O-, S-, or A- is green or blue

The amplifier requires only two pins to be connected to the microcontroller, Data Out (DOUT) and Serial Clock (CLK). Oddly enough, these can be connected to any GPIO (General Purpose Input/Output) pin and still function.

The library has some of the following useful functions and expressions:

`HX711 scale(DOUT, CLK)`

Purpose: Create a scale object to interact with the load cell with the two pins specified.

Parameters: *DOUT pin:* pin for the output of the load cell amplifier

   CLK pin: pin for the serial clock

`scale.set_scale(calibration_factor)`

Purpose: Sets the scale object's calibration factor. Run the code at least once with a scaling factor of 1 and test with a known weight. Then divide the number 10.0 by the output value gotten to calibrate the reading.

Parameters: *calibration_factor:* factor that transforms resistance measurements to load reading

`scale.tare(num_samples)`

Purpose: Tare (zero) the scale object. Can be used without an input, but if an input is chosen, the number of samples specified will be taken and averaged to find the zero position.

Parameters: *num_samples:* number of samples to average for load cell reading

`scale.get_units(num_samples)`

Purpose: Returns a floating point number corresponding to the computed load. If an input is given, then the number of samples specified is averaged to return the load.

Parameters: *num_samples:* number of samples to average for load cell reading

28.   Electric Motors

Stepper Motors

A **stepper motor** is a brushless DC electric motor that divides a full rotation into a number of equal steps. The motor's position can then be commanded to move and hold at one of these steps without any feedback sensor (an open-loop controller), as long as the motor is carefully sized to the application in respect to torque and speed. The common appearance of a stepper motor is something as is shown below:
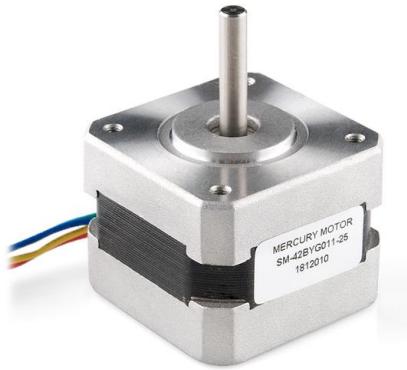
Figure 53. Example Stepper Motor

A stepper motor is a motor controlled by a series of electromagnetic coils. The center shaft has a series of magnets mounted on it, and the coils surrounding the shaft are alternately given current or not, creating magnetic fields which repulse or attract the magnets on the shaft, causing the motor to rotate.

This design allows for very precise control of the motor: by proper pulsing, it can be turned in very accurate steps of set degree increments (for example, two-degree increments, half-degree increments, etc.). They are used in printers, disk drives, and other devices where precise positioning of the motor is necessary.

There are two basic types of stepper motors, **unipolar steppers** and **bipolar steppers**.

Unipolar Stepper Motors

The unipolar stepper motor has five, six, or eight wires and four coils (actually two coils divided by center connections on each coil). The center connections of the coils are used as the power connection, and for the five wire configuration they are tied together. These stepper motors are termed unipolar steppers because power always comes in on this one pole. The other two wires on each coil are then switches HIGH or LOW in order to reverse the polarity of the current across the coil and switch stepper motor direction.

The inherent advantage of the unipolar stepper motor is ease of use. However, it takes twice as much wire per coil when compared to a bipolar stepper motor. Because only half of the coil is used, a unipolar stepper motor will generate less torque than a bipolar stepper motor.
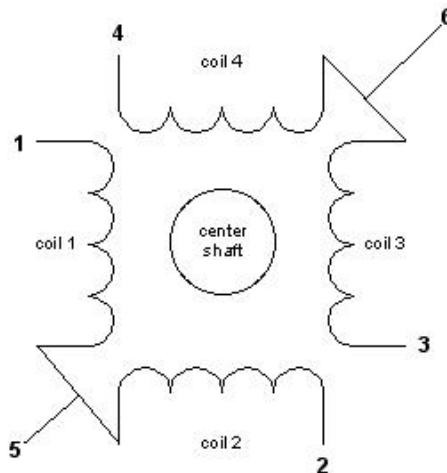
Figure 54. Unipolar Stepper Motor Schematic (5 Wire)
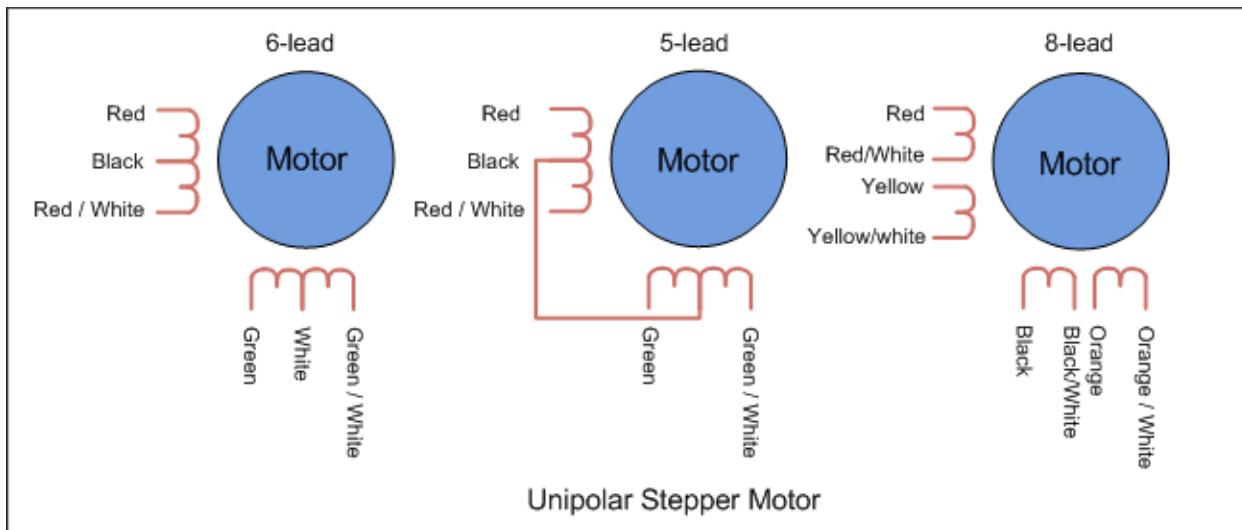
Figure 55. Unipolar Stepper Motors (Various Lead Types)

Bipolar stepper motors

The bipolar stepper motor usually has four wires coming out of it. Unlike unipolar steppers, bipolar steppers have no common center connection. They have two independent sets of coils instead. Because there is only a single winding per phase, a more complicated circuit is needed to switch the polarity. This is usually accomplished externally with an H-Bridge attached to each coil.

Bipolar stepper motors can be distinguished from unipolar steppers by measuring the resistance between the wires. You should find two pairs of wires with equal resistance. If the leads of a multimeter connected to two wires that are not electrically connected (i.e. not attached to the same coil), you should see infinite resistance (or no continuity).

Like other motors, stepper motors require more power than a microcontroller can give them, so a separate power supply is required for it. Ideally, the voltage will be known, but if not, get a variable DC power supply, apply the minimum voltage (hopefully 3V or so), apply voltage across two wires of a coil (e.g. 1 to 2 or 3 to 4) and slowly raise the voltage until the motor is difficult to turn. It is possible to damage a motor this way, so do not go too far. Typical voltages for a stepper might be 5V, 9V, 12V, 24V. Higher than 24V is less common for small steppers.
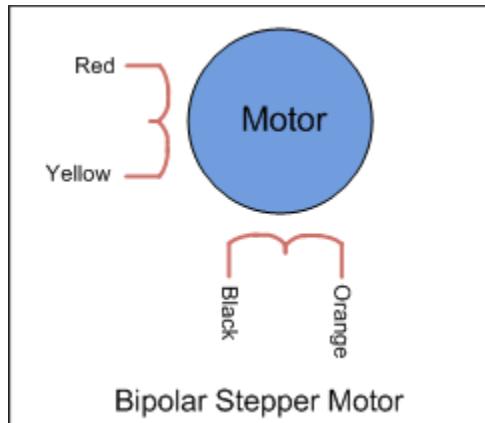
Figure 56. Bipolar Stepper Motor Schematic

To control the stepper, apply voltage to each of the coils in a specific sequence. The sequence would go like this:

Step	wire 1	wire 2	wire 3	wire 4
1	High	low	high	low
2	low	high	high	low
3	low	high	low	high
4	high	low	low	high

A comparison between the two types of stepper motors is provided below:
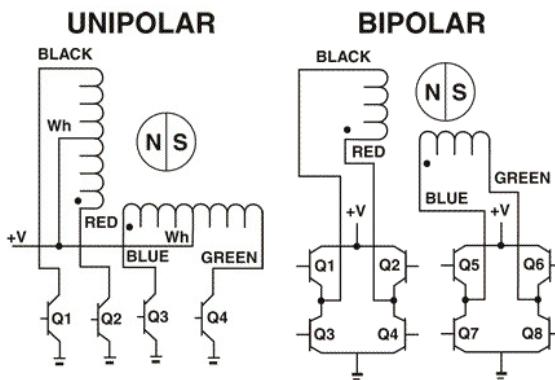
Figure 57. Comparison of Unipolar to Bipolar Stepper Motors

For convenience, instead of making your own circuit of transistors and diodes to power a stepper motor, a user can take advantage of premade boards known as **stepper motor drivers** such as the EasyDriver developed by SparkFun, shown below:
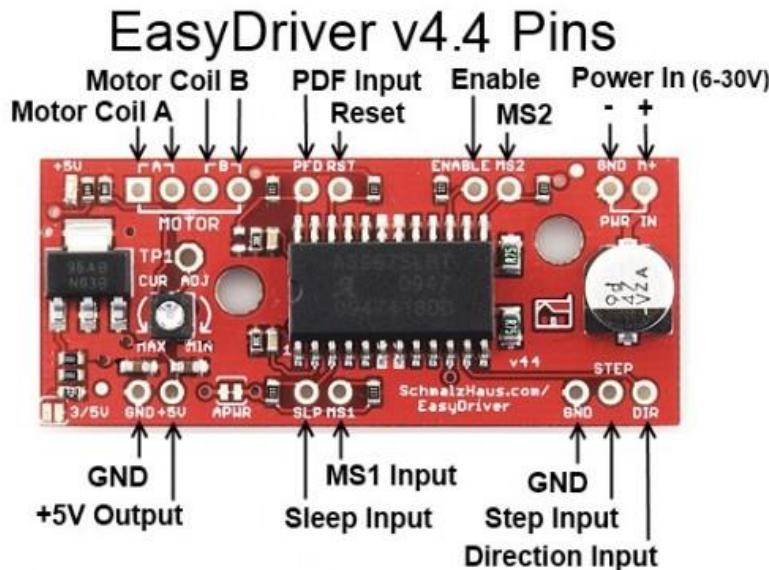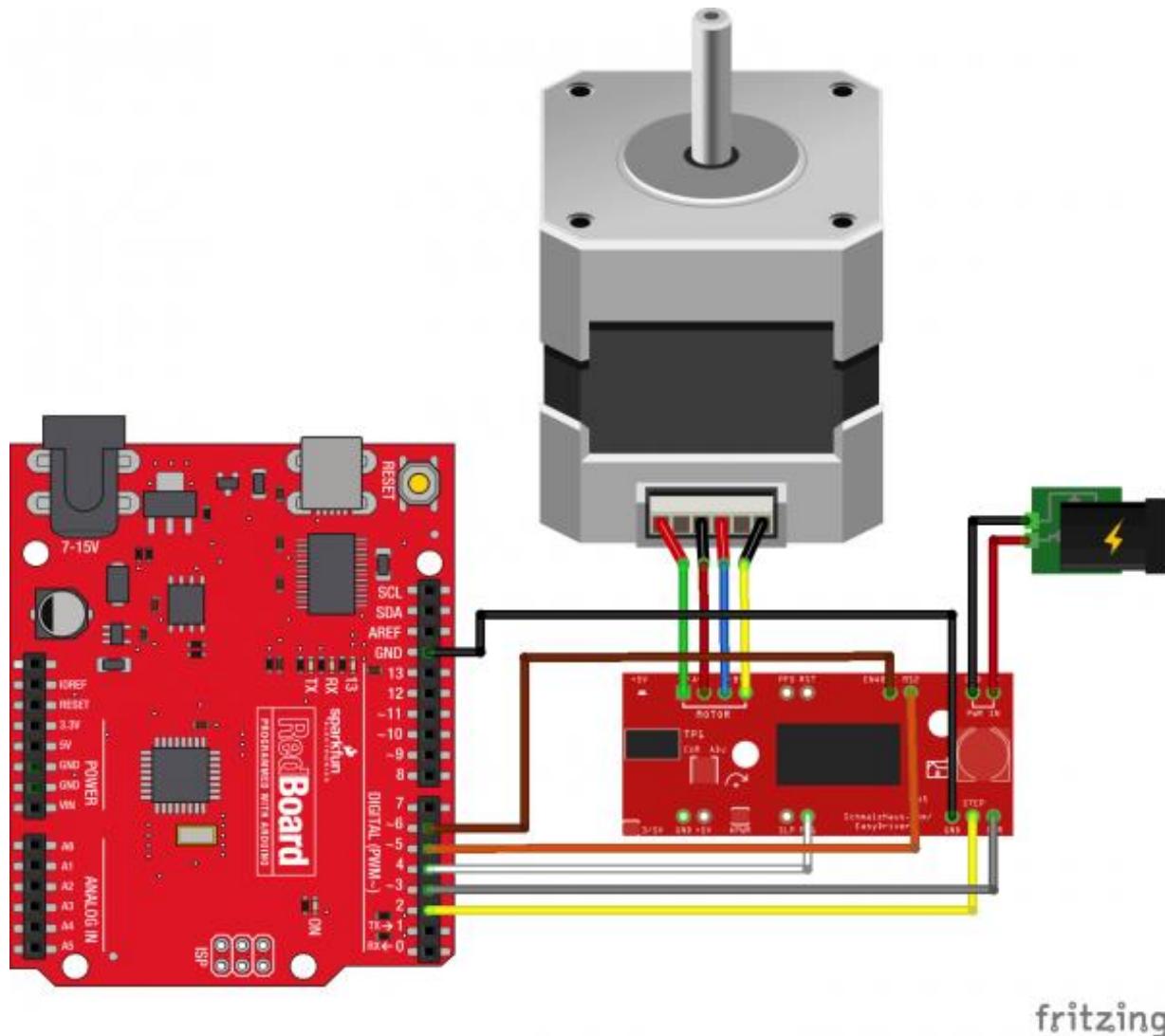

Figure 58. SparkFun EasyDriver

This provided four inputs for the four wires connected to the coils, and two additional inputs for **microstepping** (MS1 and MS2). Microstepping uses sinusoidal waveforms applied to the coils to increase angular resolution of the motor. After hooking up the motor, the circuit would look something like the image on the following page:

The **Stepper.h** library is useful for controlling stepper motors and includes the following functions:

```
Stepper myStepper = Stepper(steps, pin1, pin2, pin3, pin4)
```

Purpose: This function creates a new instance of the Stepper class that represents a particular stepper motor attached to your Arduino board. Use it at the top of your sketch, before the setup loop. The number of parameters depends on how you've wired your motor - either using two or four pins of the Arduino board.

Parameters: *steps:* the number of steps in one revolution of your motor. If your motor gives the number of degrees per step, divide that number into 360 to get the number of steps as an integer

pin1, pin2, pin3 (optional), pin4 (optional): pins that connect to stepper motor

```
myStepper.step(steps)
```

Purpose: Turns the motor a specific number of steps, at a speed determined by the most recent call to `setSpeed()`. This function is blocking; that is, it will wait until the motor has finished moving to pass control to the next line in your sketch. For example, if you set the speed to, say, 1 RPM and called `step(100)` on a 100-step motor, this function would take a full minute to run. For better control, keep the speed high and only go a few steps with each call to `step()`.

Parameters: *steps:* the number of steps to turn the motor - positive to turn one direction, negative to turn the other as an integer

```
myStepper.setSpeed(rpms)
```

Purpose: Sets the motor speed in rotations per minute (RPMs). This function doesn't make the motor turn, just sets the speed at which it will when you call `step()`.

Parameters: *rpms:* a positive long type number that reflects the speed of the motor in rotations per minute.

DC Motors

A **DC motor (Direct Current Motor)** is any of a class of rotary electrical machines that converts direct current electrical energy into mechanical energy. The most common types rely on the forces produced by magnetic fields. Nearly all types of DC motors have some internal mechanism, either electromechanical or electronic, to periodically change the direction of current flow in part of the motor. DC motors come in two major varieties: **brushed DC motors** and **brushless DC motors**.

A simple brushed electric motor will have six major components:

- **Armature or rotor**
- **Commutator**
- **Brushes**
- **Axle**
- **Field magnet**
- **DC power supply of some sort**
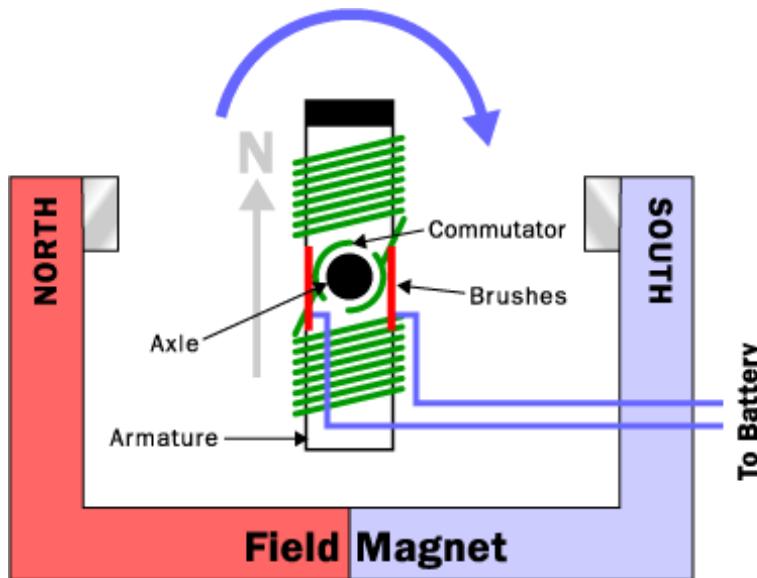
Figure 59. Brushed DC Motor Schematic

The axle holds the armature and the commutator. The armature is a set of electromagnets. The armature in a small DC motor like one would find in a toy could be a simple as thin metal plates stacked together, with thin copper wire coiled around each of the three poles of the armature. The two ends of each wire (one wire for each pole) are soldered onto a terminal, and then each of the terminals is wired to one plate of the commutator. The final piece of any DC electric motor is the field magnet. The field magnet in this motor is formed by the motor container itself plus two curved permanent magnets.

The motor turns because as electricity passes through the coils, the coils become magnetized and rotate in accordance with the magnetic field generated by the field magnet. Because the motor would stop once the coil north pole is oriented towards the field magnet's south pole and vice versa, the brushes serve the purpose or reversing the polarity of the current passing through the coils as the motor rotates.

Brushed DC motors are simple and quite easy to manufacture, but they have a litany of problems that plague them, including:

- The brushes eventually wear out.
- Because the brushes are making/breaking connections, there is sparking and electrical noise.
- The brushes limit the maximum speed of the motor.
- Convection is limited due to presence of electromagnet in the center of the motor, leading to overheating.
- The use of brushes puts a limit on how many poles the armature can have.

In a brushless DC motor, the permanent magnets are moved to the rotor and the electromagnets are moved to the stator. External transistors can then be used to power the electromagnets as the motor turns. While this operation required additional hardware and is more expensive than a brushed DC motor, it is much more reliable and can more accurately turn the motor.
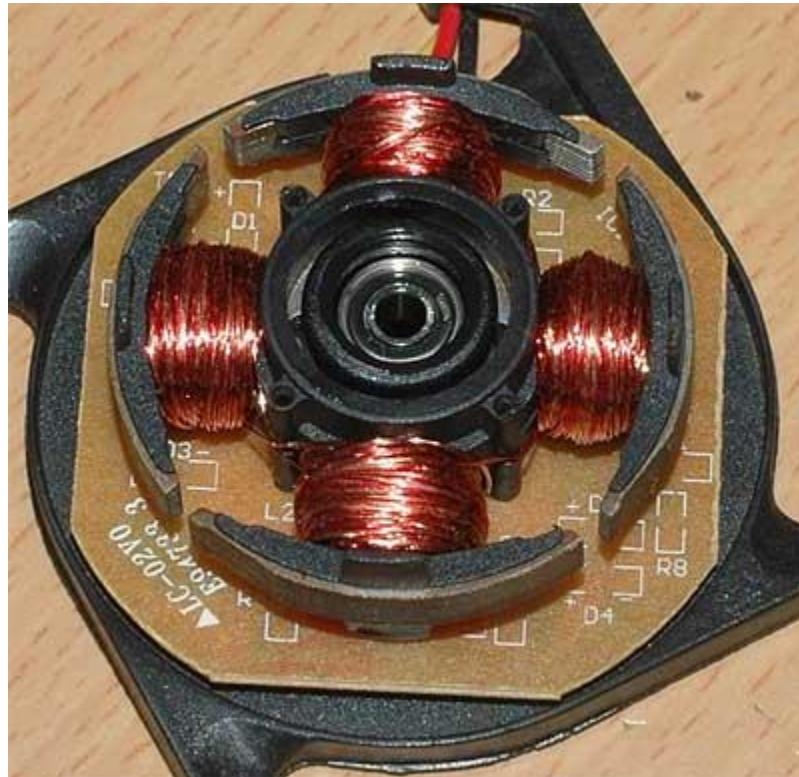
Figure 60. Brushless DC Motor

Worth noting is that method of operation is pretty similar to that of **AC motors**, except that the transistors are replaced with the naturally oscillating signal of the AC power supply that drives the motor.

DC motors can be easily included in a circuit analysis of a system. The DC motor's counter emf (E_b) is proportional to the product of the machine's total flux strength (Φ) and armature speed in rpm (n):

$$E_b = k_b \Phi n$$

where k_b is the counter emf constant of proportionality. The DC motor's input voltage must overcome the counter emf as well as the voltage drop created by the armature current across the motor resistance, that is, the combined resistance across the brushes, armature winding and series field winding (R_m), if any:

$$V_m = E_b + R_m I_a$$

The DC motor's torque (T) is proportional to the product of the armature current and the machine's total flux strength:

$$T = \frac{1}{2\pi} k_b I_a \Phi = k_T I_a \Phi$$

where k_T is the torque constant, which is itself directly proportional to the counter emf constant. The motor speed is given by the first equation to be:

$$n = \frac{E_b}{k_b \Phi}$$

Knowing the expression for V_m, it can then be seen that:

$$n = \frac{V_m - R_m I_a}{k_b \Phi} = k_n \frac{V_m - R_m I_a}{\Phi}$$

where k_n is the speed constant.

Because the total resistance of a DC motor is often less than one Ohm, there is large current draw as the motor begins to rotate and there is no inductance creating a back emf. When DC motors were first invented, this was accommodated by manually adjusting the input power as the motor would start up. In modern technology, a protection circuit known as a *four-point-starter* is frequently used to accommodate for this.
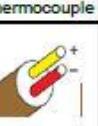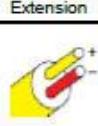
29.   Thermocouples

A **thermocouple** is a type of temperature sensor. Unlike semiconductor temperature sensors such as the TMP36 , thermocouples have no electronics inside them. They are simply made by welding together two metal wires (although if necessary, tightly twisting the wires together can also suffice). Because of a physical effect of two joined metals known as the **Seebeck effect**, there is a slight but measurable voltage across the wires that increases with temperature. This is usually expressed as a linear relationship such that:

$$V_{TC} = S\nabla T$$
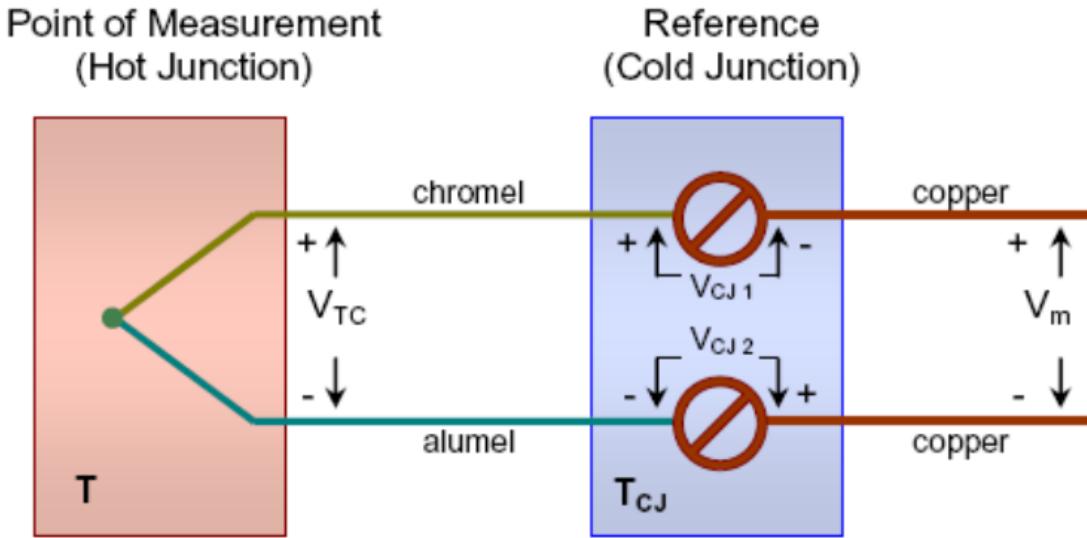
where S is the Seebeck coefficient and ∇T is the temperature gradient. The type of metals used affect the voltage range, cost, and sensitivity, which is why there are multiple types of thermocouples. The main improvement of using a thermocouple over a semiconductor sensor or thermistor is that the temperature range is very much increased. For example, the TMP36 can go from -50 to 150°C, after that the chip itself can be damaged. Common thermocouples on the other hand, can go from -200°C to 1350°C (K type) and there are ones that can go above 2300°C. The different types of thermocouples and their color codes are provided below:

Table 6. Thermocouple Types

ANSI Code	ANSI MC 96.1 Color Coding		Alloy Combination		Maximum T/C Grande temp. range	EMF(mv)Over Max.temp.range	IEC 584-3 Color Coding	IEC Code
	Thermocouple	Extension	+ Lead	- Lead				
K			NICKEL-CHROMIUM Ni-Cr	NICKEL-ALUMINUM Ni-Al	-270 to 1372°C -454 to 2501°F	-6.458 to 54.886		K
J			IRON Fe (magnetic)	CONTANTAN COOPER-NICKEL Cu-Ni	-210 to 1200°C -346 to 2193°F	-8.095 to 69.553		J
T			COPPER Cu	CONTANTAN COOPER-NICKEL Cu-Ni	-270 to 400°C -454 to 752°F	-6.258 to 20.872		T
E			NICKEL-CHROMIUM Ni-Cr	CONTANTAN COOPER-NICKEL Cu-Ni	-270 to 1000°C -454 to 1832°F	-9.835 to 76.373		E
N			NICROSIL Ni-Cr-Si	NISIL Ni-Si-Mg	-270 to 1300°C -450 to 2372°F	-4.345 to 47.513		N
S	NONE ESTABLISHED		PLATINUM-10% RHODIUM Pt-10%Rh	PLATINUM Pt	-50 to 1768°C -58 to 3214°F	-0.236 to 18.693		S
R	NONE ESTABLISHED		PLATINUM-13% RHODIUM Pt-13%Rh	PLATINUM Pt	-50 to 1768°C -58 to 3214°F	-0.226 to 21.101		R
B	NONE ESTABLISHED		PLATINUM-30% RHODIUM Pt-30%Rh	PLATINUM-6% RHODIUM Pt-6%Rh	0 to 1820°C 32 to 3308°F	0 to 13.820		B

One end of a thermocouple sensor comprises the junction which is placed on the object whose temperature is to be measured. At the other end of the wire pair, the thermocouple voltage is measured. However, additional thermocouple junctions are formed when connecting to the wire ends, so the total potential measured depends on the temperatures of those junctions as well as the junction at the measurement end of the thermocouple. Consequently, thermocouple voltages are relative values which must be measured with respect to a junction at a known temperature, called the **cold junction** or **reference junction**.

The following diagram shows a K-type thermocouple used to measure the temperature at its hot junction, and the influence of the cold junction on the measurement.

$$V_m = V_{TC} - (V_{CJ\,1} + V_{CJ\,2}) = V_{TC} - V_{CJ}$$

Figure 61. Thermocouple Method of Operation

This voltage is a roughly linear function of the hot junction temperature inside of the range of temperatures the thermocouple is rated for. It is worth noting that depending on the mechanical structure of the probe, the time constant of the temperature response of the hot junction will change. It is also worth noting that grounding the hot junction (which can happen accidentally) will cause the thermocouple data to be useless.

This read voltage is typically very small, and so a **thermocouple amplifier** is required to interface with something like a microcontroller. As mentioned before, these amplifiers frequently use OneWire as a data transmission protocol, but they exist in all forms (the MAX31855, for example, uses SPI protocol). The MAX31850K is a very useful breakout board for reading K type thermocouples with OneWire protocol (and can be used with parasitic power mode thanks to an on-board capacitor so only two wires total are needed). The use of this breakout board also requires the use of the **DallasTemp.h** library as well as the OneWire.h library. This library creates some very useful and easy to use functions for use in interacting with the thermocouple over the OneWire Bus (including the `getTempC()` and `getTempF()` functions that explicitly return the temperature with no math required by the user except for filtering). A tutorial demonstrating how to use both libraries is available on Adafruit's website.

30.   Telemetry and Wireless Data Transmission

Telemetry is an automated communications process by which measurements and other data are collected at remote or inaccessible points and transmitted to receiving equipment for monitoring. In other words, telemetry is the wireless transmission of data. The simplest devices capable of telemetry are radio frequency (RF) modules consisting of a dedicated **transmitter** and **receiver**. The transmitter (such as the RF Link Transmitter – 434 MHz) broadcasts data and the receiver (such as the RF Link Receiver – 434 MHz) can receive the data. The benefit of using such a pair is that compared to the alternatives, they are very cheap (the linked products cost less than ten dollars for the pair). The downsides, however, is that they are only capable of one-way communication, and can only send digital data which is subject to non-negligible noise, and using multiple transmitters and receivers can be difficult and may require multiple transmitter receiver pairs operating on different frequencies.

Commonly used transmitter/receiver pairs come in the form of radio modules such as the **Digi XBee** which transmits data using UART using **Zigbee wireless protocols**. An example of some XBee modules is provided below:
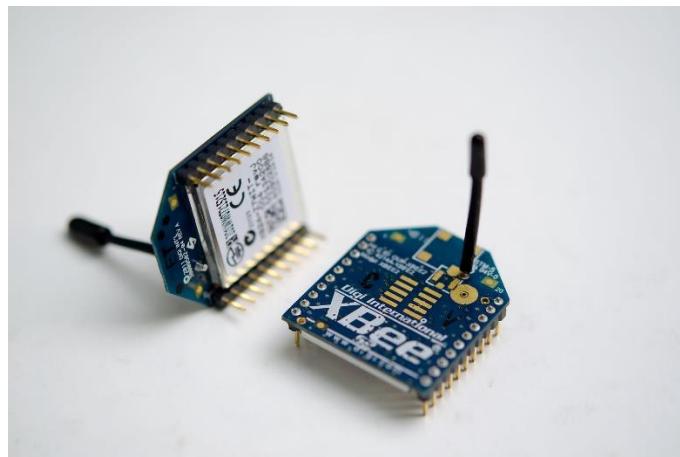

Figure 62. XBEE Modules with External Antennas

This type of module has one distinct advantage in that it is easy to use and can transmit data between two microcontrollers fairly easily. There exist several XBee modules that cover a range of form factors, antenna types, and radio frequencies. Unfortunately, they tend to be more expensive and often they do not easily fit into a breadboard, so adapters are available for this purpose and for interfacing with a microcontroller easily (for example, the Sparkfun XBee Explorer Regulated).

Bluetooth (BT) is a wireless technology standard for exchanging data over short distances with frequencies in the range of 2.4-2.485 GHz at a maximum range of about 30 feet. An easy way to view BT is the radio frequency (RF) equivalent to Serial communication.

Bluetooth is a packet-based protocol with a master-slave structure. One master may communicate with up to seven slaves in what is called a **piconet**. All devices share the master's clock. Packet exchange is based on the basic clock, defined by the master, which ticks at 312.5 µs intervals. Two clock ticks make up a slot of 625 µs, and two slots make up a slot pair of 1250 µs. In the simple case of single-slot packets the master transmits in even slots and receives in odd slots. The slave, conversely, receives in even slots and transmits

in odd slots. Packets may be 1, 3 or 5 slots long, but in all cases the master's transmission begins in even slots and the slave's in odd slots.

Every single Bluetooth device has a unique 48-bit address, commonly abbreviated BD_ADDR. This will usually be presented in the form of a 12-digit hexadecimal value. The most-significant half (24 bits) of the address is an **organization unique identifier** (OUI), which identifies the manufacturer. The lower 24-bits are the unique part of the address.

Creating a Bluetooth connection between two devices is a multi-step process involving three progressive states:

1. **Inquiry** – If two Bluetooth devices know absolutely nothing about each other, one must run an inquiry to try to **discover** the other. One device sends out the inquiry request, and any device listening for such a request will respond with its address, and possibly its name and other information.
2. **Paging (Connecting)** – Paging is the process of forming a connection between two Bluetooth devices. Before this connection can be initiated, each device needs to know the address of the other (found in the inquiry process).
3. **Connection** – After a device has completed the paging process, it enters the connection state. While connected, a device can either be actively participating or it can be put into a low power sleep mode.
   o **Active Mode** – This is the regular connected mode, where the device is actively transmitting or receiving data.
   o **Sniff Mode** – This is a power-saving mode, where the device is less active. It will sleep and only listen for transmissions at a set interval (e.g. every 100ms).
   o **Hold Mode** – Hold mode is a temporary, power-saving mode where a device sleeps for a defined period and then returns back to active mode when that interval has passed. The master can command a slave device to hold.
   o **Park Mode** – Park is the deepest of sleep modes. A master can command a slave to "park", and that slave will become inactive until the master tells it to wake back up.

When two Bluetooth devices share a special affinity for each other, they can be **bonded** together. Bonded devices automatically establish a connection whenever they are close enough. For example, a phone can be synced up to a car's BT system such that whenever the car starts up, the phone immediately syncs up without any user input required.

Bonds are created through one-time a process called **pairing**. When devices pair up, they share their addresses, names, and profiles, and usually store them in memory. The also share a common secret **key**, which allows them to bond whenever they are together in the future.

Pairing usually requires an authentication process where a user must validate the connection between devices. The flow of the authentication process varies and usually depends on the interface capabilities of one device or the other. Sometimes pairing is a simple operation, where the click of a button is all it takes to pair (this is common for devices with no UI, like headsets). Other times pairing involves matching six digit numeric codes. Older, legacy (v2.0 and earlier), pairing processes involve the entering of a common PIN code on each device. The PIN code can range in length and complexity from four numbers (e.g. "0000" or "1234") to a 16-character alphanumeric string.

The transmit power, and therefore **range**, of a Bluetooth module is defined by its power class. There are three defined classes of power:

Table 7. Bluetooth Power Classes

Class Number	Max Output Power (dBm)	Max Output Power (mW)	Max Range
Class 1	20 dBm	100 mW	100 m
Class 2	4 dBm	2.5 mW	10 m
Class 3	0 dBm	1 mW	10 cm

Some modules are only able to operate in one power class, while others can vary their transmit power.

Bluetooth profiles are additional protocols that build upon the basic Bluetooth standard to more clearly define what kind of data a Bluetooth module is transmitting. While Bluetooth specifications define how the technology works, profiles define how it is used.

The profile(s) a Bluetooth device supports determine(s) what application it's geared towards. A hands-free Bluetooth headset, for example, would use headset profile (HSP), while a Nintendo Wii Controller would implement the human interface device (HID) profile. For two Bluetooth devices to be compatible, they must support the same profiles.

For replacing a serial communication interface (like RS-232 or a UART) with Bluetooth, **Serial Port Profile (SPP)** is the suggested profile. SPP is great for sending bursts of data between two devices. It is one of the more fundamental Bluetooth profiles (Bluetooth's original purpose was to replace RS-232 cables after all).

Using SPP, each connected device can send and receive data just as if there were RX and TX lines connected between them. Two Arduinos, for example, could converse with each other from across rooms, instead of from across the desk.

Several Bluetooth modules are available for interfacing with microcontrollers. Among them are the HC-05 Transceiver which can act as either the master or the slave. An identical module that can only act as the slave is the HC-06. These small (~3 cm long) modules run on 3.3V power with 3.3V signal levels. They have no pins and usually solder to a larger board. Often, larger boards are sold with pins that incorporate these modules, such as the board shown on the following page:
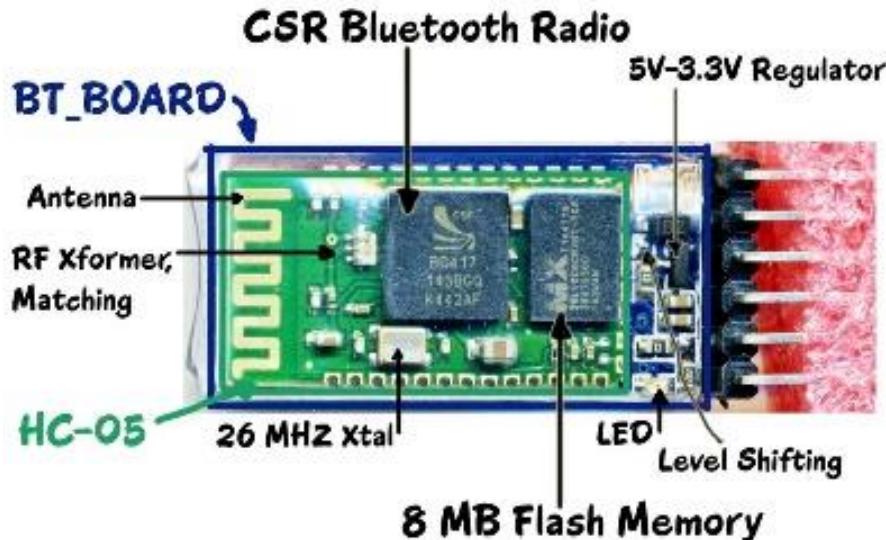
Figure 63. HC05 Bluetooth Module

Note that the green HC-05 sub-module is soldered on top of the blue BT Board. The HC-05 module includes the Radio and Memory chips, 26 MHz crystal, antenna and RF matching network. The right section of the BT Board has connection pins for power and signals as well as a 5V to 3.3V Regulator, LED, and level shifting.

The pin-out of this board is shown below:

- **KEY:** If brought HIGH **before** power is applied, forces AT Command Setup Mode. LED blinks slowly (2 seconds)
- **VCC:** Power Supply (+5 V)
- **GND:** Microcontroller Ground
- **TXD:** Transmit Serial Data from HC-05 to microcontroller Serial Receive. (NOTE: 3.3V HIGH level)
- **RXD:** Receive Serial Data from microcontroller Serial Transmit
- **STATE:** Reports if connected or not

The module has two modes of operation, *Command Mode* where we can send **AT commands** (an AT Command is short for Attention Command, which are used to initiate the beginning of a command prompt) to it and *Data Mode* where it transmits and receives data to another bluetooth module.

The default mode is Data Mode, and the default configuration is given by:

- **BAUD Rate:** 9600 bps
- **Data:** 8 bits
- **Stop Bits:** 1 bit
- **Parity Bits:** None
- **Handshake:** None
- **Passkey:** 1234
- **Device Name:** HC-05

In some cases you may want to change some of the configuration setup values. There are two ways to get into Command Mode:

Connect the KEY pin high before applying power to the module. This will put the module into command mode at 38400 baud. This is commonly used, and needed if you don't know the baud rate the module is set to. You can use the BlueToothCommandUtility for this.

Apply power to the module then pull the KEY pin high. This will enter command mode at the currently configured baud rate. This is useful if you want to send AT commands from a microcontroller as the KEY pin can be controlled from one of the microcontroller pins, but you need to know the currently configured Baud Rate.

Commands are sent to the module in UPPERCASE and are terminated with a CR/LF pair. The format of commands is:

Always starts with "AT", then "+" followed by <ParameterName>.

Then either:

- ? (returns current value of parameter)
- = (New Value of parameter)

A few examples:

- **AT** (AT Test command. Should respond with OK)
- **AT+VERSION?** (show the firmware version)
- **AT+UART=9600,0,0** (Set baud rate to 9600, 1 stop bit, no parity)

Bluetooth Master Mode:

To configure the module as Bluetooth Master and to pair with another bluetooth module follow these steps. The module must first be put command mode as above by pulling the CMD pin high before power on.

Enter these commands in order:

- **AT+RMAAD** Clear any paired devices
- **AT+ROLE=1** Set mode to Master
- **AT+RESET** After changing role, reset is required
- **AT+CMODE=1** Allow connection to any address
- **AT+INQM=0,5,5** Inquire mode - Standard, stop after 5 devices found or after 5 seconds
- **AT+PSWD=1234** Set PIN. Should be same as slave device
- **AT+INIT** Start Serial Port Profile (SPP) (If Error(17) returned - ignore as profile already loaded)
- **AT+INQ** Start searching for devices

A list of devices found will be displayed, one of which is the slave module. The format of the output is:

+INQ:address,type,signal

The address of the module is what we need and is in the format 0123:4:567890 (**NOTE:** The colons must be replaced with commas when we use the address with the following commands.)

If you get more than one device listed and don't know which one is the slave module, you can query the module for it's name using:

AT+RNAME? <address>

Once the correct slave address if found, the microcontroller needs to pair with it, which requires the next set of commands.

- **AT+PAIR=<address>,<timeout>** The timeout is in seconds and if you need to type in the pin on the slave device you need to give enough time to do this.
- **AT+BIND=<address>** Set bind address to the slave address
- **AT+CMODE=0** Allow master to ONLY connect to bound address (slave). This allows the master to automatically connect to the slave when switched on
- **AT+LINK=<address>** Connect to slave.

Slave Mode:

The HC-05 bluetooth module can also act as a slave. There are fewer commands to set this up:

- **AT+ORGL** Reset to defaults
- **AT+RMAAD** Clear any paired devices
- **AT+ROLE=0** Set mode to SLAVE
- **AT+ADDR** Display SLAVE address

31.   Data Storage

Data that is taken in by a microcontroller can be displayed onto the Serial monitor or it can use a program such as the streaming serial plotter to display data in real time and store it in a program such as MATLAB listed under Additional Resources. Microcontrollers can store non-volatile data by themselves, however, using what is known as **EEPROM (*El*ectrically *E*rasable *P*rogrammable *R*ead-*o*nly *M*emory)**. EEPROMs are organized as arrays of floating-gate transistors. EEPROMs can be programmed and erased in-circuit, by applying special programming signals. Originally, EEPROMs were limited to single byte operations which made them slower, but modern EEPROMs allow multi-byte page operations. It also has a limited life for erasing and reprogramming, now reaching a million operations in modern EEPROMs. In an EEPROM that is frequently reprogrammed while the computer is in use, the life of the EEPROM is an important design consideration.

Essentially, EEPROM allows you to permanently store small amounts of data, which is very useful for saving settings, collecting small data sets, or any other use where you need to retain data even if the power is turned off. EEPROM is included with the Arduino IDE and is built in to most microcontrollers, so no additional hardware is required. For example, every Teensy microcontroller has a different amount of EEPROM available for use:

Board	EEPROM Size
Teensy 3.6	4096 bytes
Teensy 3.5	4096 bytes
Teensy 3.2	2048 bytes
Teensy 3.1	2048 bytes
Teensy 3.0	2048 bytes
Teensy LC	128 bytes
Teensy++ 2.0	4096 bytes
Teensy 2.0	1024 bytes
Teensy++ 1.0	2048 bytes
Teensy 1.0	512 bytes

The supported micro-controllers on the various Arduino and Genuino boards have different amounts of EEPROM: 1024 bytes on the ATmega328, 512 bytes on the ATmega168 and ATmega8, 4 KB (4096 bytes) on the ATmega1280 and ATmega2560. The Arduino and Genuino 101 boards have an emulated EEPROM space of 1024 bytes.

Worth noting is that the Teensy 3.6 can not write to EEPROM memory when running faster than 120 MHz. Also, for all Teensy microcontrollers, the EEPROM library will automatically reduce the processor's speed during the time EEPROM data is written. If using Serial1 or Serial2, communication may be disrupted due to BAUD rate changes. Other serial ports are not affected by the temporary speed change during EEPROM writing.

The data is stored as bytes located at the EEPROM address. Address can range from 0 to the EEPROM size minus 1. For a Teensy 2.0, Address can be 0 to 1023, for a total of 1024 unique bytes that can be stored in the EEPROM.

The EEPROM is specified with a write endurance of 100,000 cycles. Each time you write, the memory is stressed, and eventually it will become less reliable. Each EEPROM address is guaranteed to work for at least 100,000 writes, and will very likely work for many more. Normally this limit is not an issue if you write to the EEPROM infrequently. Reading data from an address does not stress the EEPROM, only writes count for the write endurance.

The functions for usage of EEPROM are simple enough and are listed below:

EEPROM.read(address)

Purpose: Read a byte (0 to 255) from the EEPROM.

Parameters: *address:* the location within the EEPROM.

EEPROM.write(address, data)

Purpose: Write a byte (0 to 255) from the EEPROM.

Parameters: *address:* the location within the EEPROM to store the byte.

data: the number to store

SD Cards

A very popular way to permanently store data is through the use of **SD Cards** (Secure Digital Cards). SD Cards are data storage circuits that interface using SPI, but there exist adapters and libraries for streamlining this process when interacting with a microcontroller. PJRC directly sells SD card adapters here on their website, but there exist several vendors for such adapters and an SD shield exists for Arduino that serves the same purpose. The SD library allows for reading from and writing to SD cards, e.g. on the Arduino Ethernet Shield. This library is built on sdfatlib developed by William Greiman. The library supports FAT16 and FAT32 file systems on standard SD cards and SDHC cards. It uses short 8.3 names for files. The file names passed to the SD library functions can include paths separated by forward-slashes, /, e.g. "directory/filename.txt". Because the working directory is always the root of the SD card, a name refers to the same file whether or not it includes a leading slash (e.g. "/file.txt" is equivalent to "file.txt"). As of version 1.0, the library supports opening multiple files.

The communication between the microcontroller and the SD card uses SPI, which takes place on digital pins 11, 12, and 13 (on most Arduino boards) or 50, 51, and 52 (Arduino Mega). Additionally, another pin must be used to select the SD card. This can be the hardware SS pin - pin 10 (on most Arduino boards) or pin 53 (on the Mega) - or another pin specified in the call to SD.begin(). Note that even if you do not use the hardware SS pin, it must be left as an output or the SD library will not work.

32. Oscilloscopes

Oftentimes when it comes to troubleshooting and sifting through data, the use of an **oscilloscope** can be very helpful. An oscilloscope (sometimes colloquially just called scope) is an electronic testing instrument that allows observation of constantly varying voltage signals, usually as a function of time. For projects involving the measurement of vibrations, these instruments are particularly useful because the vibrations can be turned into voltages through the use of an accelerometer and then read. An example oscilloscope output is provided below:
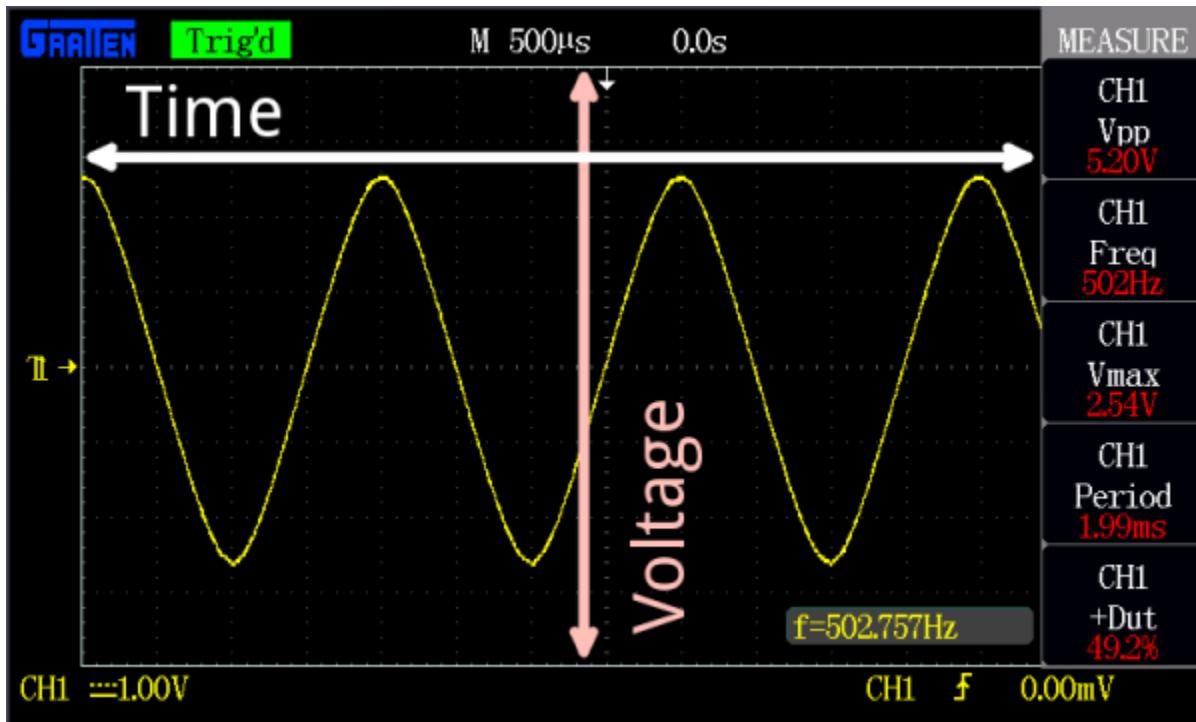
Figure 64. Oscilloscope Display

There are several key aspects to selecting an oscilloscope that quantify its performance:

- **Bandwidth** – Oscilloscopes are most commonly used to measure waveforms which have a defined frequency. No scope is perfect though: they all have limits as to how fast they can see a signal change. The bandwidth of a scope specifies the range of frequencies it can reliably measure.
- **Digital vs. Analog** – As with most everything electronic, o-scopes can either be analog or digital. Analog scopes use an electron beam to directly map the input voltage to a display. Digital scopes incorporate microcontrollers, which sample the input signal with an analog-to-digital converter and map that reading to the display. Generally analog scopes are older, have a lower bandwidth, and less features, but they may have a faster response (and look much cooler).
- **Channel Amount** – Many scopes can read more than one signal at a time, displaying them all on the screen simultaneously. Each signal read by a scope is fed into a separate channel. Two to four channel scopes are very common.
- **Sampling Rate** – This characteristic is unique to digital scopes, it defines how many times per second a signal is read. For scopes that have more than one channel, this value may decrease if multiple channels are in use.

- **Rise Time** – The specified rise time of a scope defines the fastest rising pulse it can measure. The rise time of a scope is very closely related to the bandwidth. It can be approximated with $t_r = \frac{0.35}{BW}$.

- **Maximum Input Voltage** – Every piece of electronics has its limits when it comes to high voltage. Scopes should all be rated with a maximum input voltage. If your signal exceeds that voltage, there's a good chance the scope will be damaged.

- **Resolution** – The resolution of a scope represents how precisely it can measure the input voltage. This value can change as the vertical scale is adjusted.

- **Vertical Sensitivity** – This value represents the minimum and maximum values of your vertical, voltage scale. This value is listed in volts per div.

- **Time Base** – Time base usually indicates the range of sensitivities on the horizontal, time axis. This value is listed in seconds per div.

- **Input Impedance** – When signal frequencies get very high, even a small impedance (resistance, capacitance, or inductance) added to a circuit can affect the signal. Every oscilloscope will add a certain impedance to a circuit it's reading, called the input impedance. Input impedances are generally represented as a large resistive impedance (>1 MΩ) in parallel (∥) with small capacitance (in the pF range). The impact of input impedance is more apparent when measuring very high frequency signals, and the probe you use may have to help compensate for it.

Oscilloscopes offer you the ability (through the use of knobs traditionally) to adjust the number of divisions on the time scale (horizontal) or the voltage scale (vertical) by adjusting the value of Δt or ΔV for those divisions. They also allow for the horizontal and vertical axis to be offset by some bias amount specified by the user.

One of the more useful features of the oscilloscope comes in the form of **triggers**. A trigger is an indication set by the user that tells the oscilloscope when to start measuring the data. This is very useful for getting a still-shot image of the waveform on the oscilloscope display. Triggers come in several trigger types, such as:

- An **edge trigger** is the most basic form of the trigger. It will key the oscilloscope to start measuring when the signal voltage passes a certain level. An edge trigger can be set to catch on a rising or falling edge (or both).

- A **pulse trigger** tells the scope to key in on a specified "pulse" of voltage. You can specify the duration and direction of the pulse. For example, it can be a tiny blip of 0V -> 5V -> 0V, or it can be a seconds-long dip from 5V to 0V, back to 5V.

- A **slope trigger** can be set to trigger the scope on a positive or negative slope over a specified amount of time.

You can also usually select a triggering mode, which, in effect, tells the scope how strongly you feel about your trigger. In automatic trigger mode, the scope can attempt to draw your waveform even if it doesn't trigger. Normal mode will only draw your wave if it sees the specified trigger. And single mode looks for your specified trigger, when it sees it will draw your wave then stop.

The last note on oscilloscopes is that the probes that collect the data may appear simple, but they are actually specially calibrated circuits that enable the inductance present in the wires to be negated by the presence of a resistor and capacitor. Courtesy of SparkFun, an example circuit showing the probe of a oscilloscope is given below:
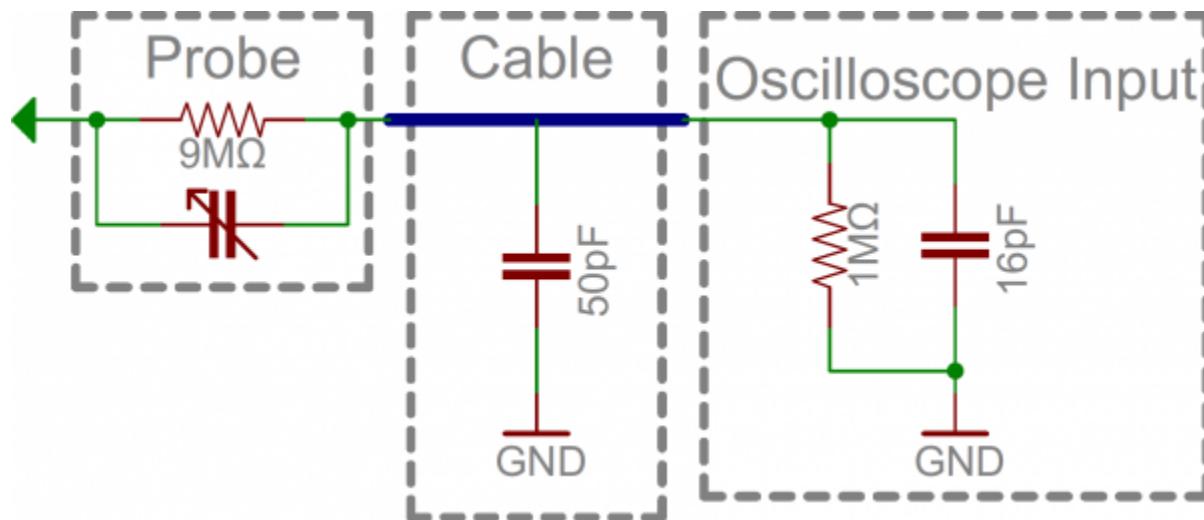
Figure 65. Oscilloscope Probe Circuit

In general, oscilloscopes are bulky instruments that can cost several thousand dollars. However, the Digilent Analog Discovery 2 is a device that includes not only an oscilloscope, but a waveform generator and several digital I/O's and can interface directly with a computer via a micro USB cable. This device is small enough to be easily transportable and only cost a few hundred dollars.

33. Soldering

Solder is a fusible metal alloy used to create permanent bonds between electrical components. The most commonly available solders are tin-lead solders, although due to health concerns lead free solders are becoming more prominent, especially in Europe. Solder comes in several forms, but can be commonly bought on a spool, as shown:

Figure 66. Spool of 1.6 mm Tin-Lead Solder

Solder is often sold with a **flux core** which has three primary purposes:

1. It removes any oxidized metal from the bonding surface

2. It seals out air from the bonding surface to prevent oxidation while the metals are hot.

3. It aids in the solders ability to maintain a contact surface (improves **wettability**)

The amount of solder types that exist can be somewhat overwhelming, but a general rule of thumb is that the greater the tin content, the higher the yield strength of the joint and the lower the melting point. Other metals can be alloyed to change properties such as the thermal conductivity, melting point, etc.

The tool used to melt the solder is typically a **soldering iron**. An example soldering iron is shown below:
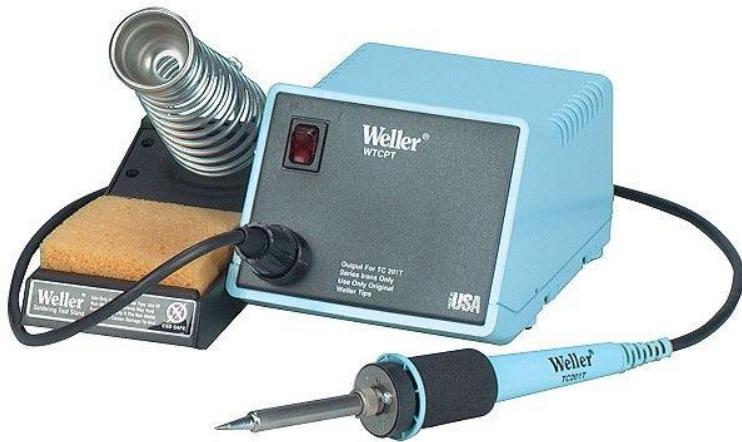
Figure 67. Example of a Soldering Iron

In order to solder something like pin headers into a PCB, a breadboard can be used to ensure everything is aligned and levelled, and a soldering iron and solder can be used to create the permanent bonds between the pin headers and the board. In order to properly solder in this fashion, there are some basic steps:

1. The tip of the soldering iron and whatever is being soldered must be **tinned** (some solder must be melted and deposited onto the surfaces)
2. The tip of the soldering iron must simultaneously be touching the copper plating of the through hole as well as the pin header so that both components heat up (in the case where pin headers are being soldered into a PCB).
3. The solder must be brought in so that it touches both the copper plating and the pin header.
4. Move the solder close to the soldering iron tip until it begins to melt. As it gets hot enough, capillary action will "suck" the solder into the through hole.
5. Clean the tip of the soldering iron using a wet sponge and move on to the next thing to be soldered.

The following image illustrates this basic idea:
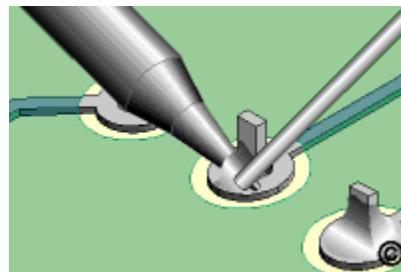
Figure 68. Illustration of Soldering Pin Headers to PCB

Upon completion, the solder joint should have a distinct look about it, illustrated below:

Figure 69. Anatomy of a Good Solder Joint

If the pin gets significantly hotter than the pad, then the solder tends to "bubble up" and not move through the through hole.

Soldering wires together is also a fairly rudimentary task. Simply fray the wires and wrap the two frayed portions together to maximize electrical contact and then apply both the solder and the soldering iron to the contact point.

34. Electronic Packaging

Electronic packaging is a major discipline within the field of electronic engineering, and includes a wide variety of technologies. It refers to enclosures and protective features built into the product itself, and not to shipping containers. It applies both to end products and to components. Packaging of an electronic system must consider protection from mechanical damage, cooling, radio frequency noise emission, protection from electrostatic discharge, maintenance, operator convenience, and cost. Prototypes and industrial equipment made in small quantities may use standardized commercially available enclosures such as card cages or prefabricated boxes. Mass-market consumer devices may have highly specialized packaging to increase consumer appeal. The same electronic system may be packaged as a portable device or adapted for fixed mounting in an instrument rack or permanent installation.

The primary functions of electronic packaging are:

- Signal Distribution
- Power Distribution
- Heat Dissipation
- Protection of Components

The current trends among electronic packages in the market are that they are:

- Smaller
- Lighter
- Faster
- Increased Circuit Density
- Higher Power Density
- More Complicated Functionalities
- More Inputs/Outputs
- More Reliable
- Less Expensive

These current trends provide a litany of challenges for mechanical and electrical engineers to face when designing circuits.
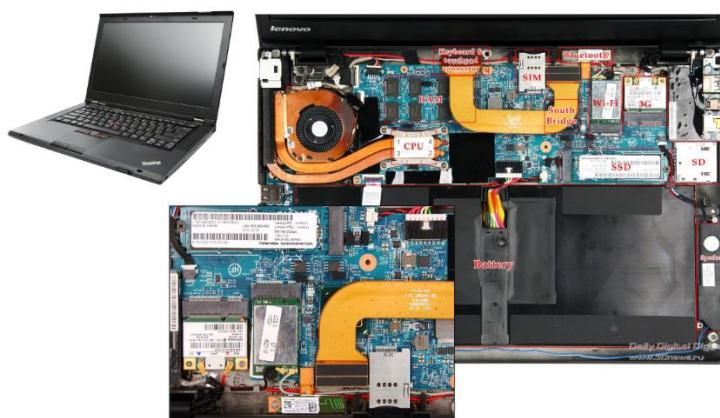
Figure 70. Circuitry inside of a Lenovo Thinkpad

Integrated circuits such as microcontrollers are what are termed **electronic packages**. Electronic packages consist of one or more **chips** (also called **dies**) that perform very specific tasks for the microcontroller. These chips are usually made of silicon.

A first level electronic package, or **chip carrier**, serves the purpose of protecting the chip and providing these interconnection points. It accomplishes this by attaching the chip to the leads on the package through some chip interconnection and covering the chip with an epoxy for protection. This epoxy cover is called **overmolding,** or simply an **encapsulant** for plastic packages.
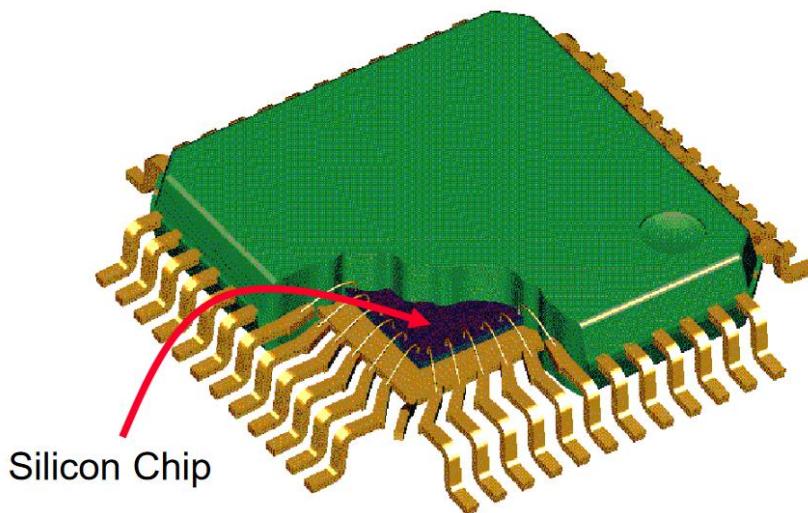
Figure 71. Silicon Chip attached to Copper Leads (Overmolding shown in Green)

Ceramic packages exist as an alternative to plastic packages. These packages provide hermeticity (protection from moisture) and are much more expensive than their plastic counterparts. For this reason, they are typically only used in military or high performance applications. Although plastic packages outnumber ceramic packages, dollar for dollar the ceramic package controls two thirds of the electronic package market. The predominant ceramic material is alumina, and most ceramic processing formulations and procedures are extremely proprietary. There are two varieties of ceramic packages: **Cofired Multilayer Ceramic Packages** and **Pressed Ceramic Packages**. Cofired packages are formed by casting a slurry of alumina powder, glass powder, and organic binders which are sintered together. Pressed packages sinter the top and bottom portions of the ceramic package separately and press them together under heating conditions to get a seal.

First level electronic packages come in several form factors. A few of these and their names are provided below:
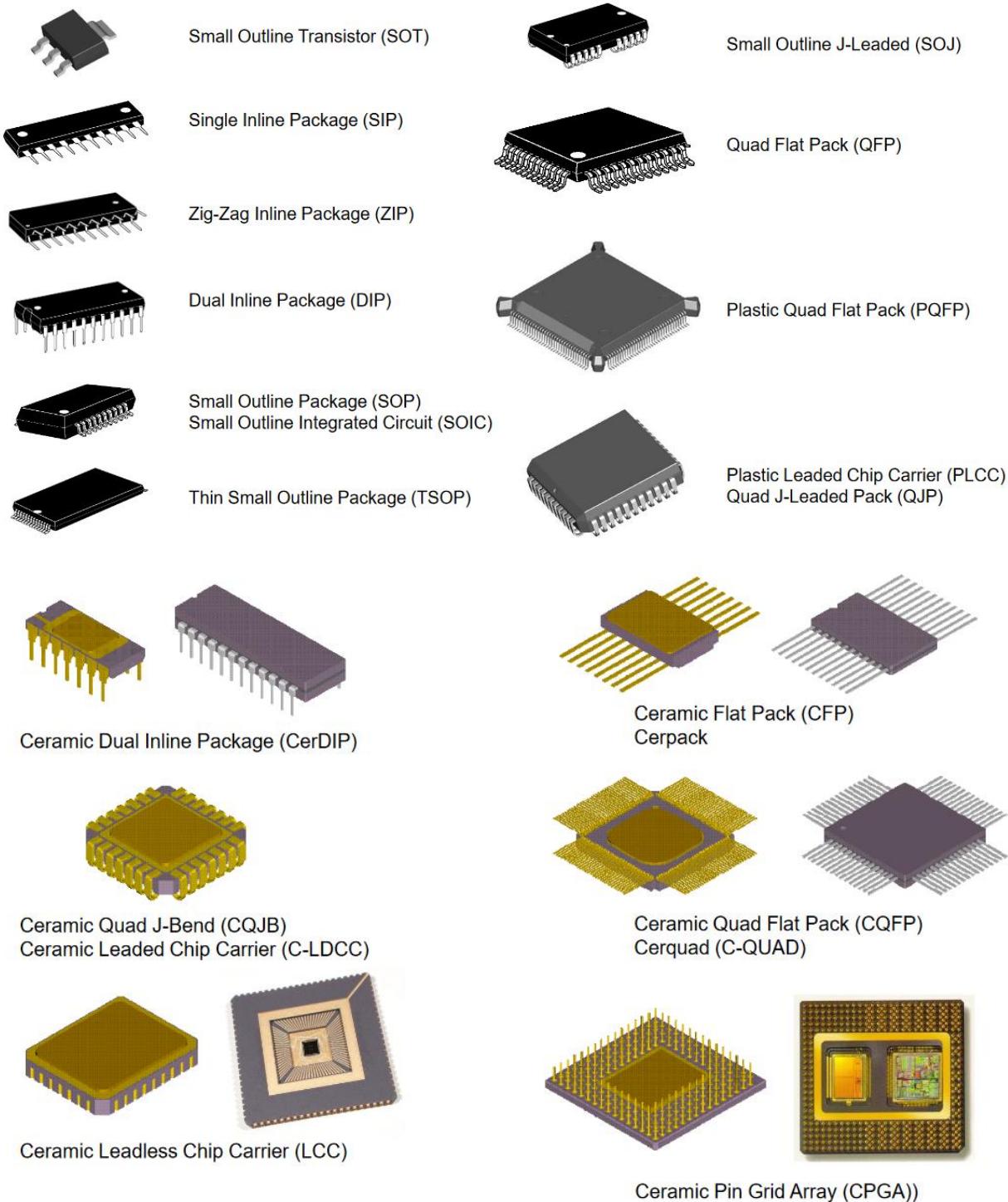
Figure 72. Form Factors of First Level Electronic Packages

From this diagram, it is easily seen that packages such as the Teensy microcontroller qualifies as a Dual Inline Package (DIP). Packages such as these mount to a PCB with **plated through-holes (PTH)**. Packages that incorporate J-Leads or Gull-Leads (which bend underneath the package or outward from it, respectively) are called **Surface Mount Technology (SMT)**, because they mount directly to solder pads on the surface of the PCB. A first level electronic package does not need to only have one chip, because so called **multi-chip modules (MCM)** do exist.

A second level package would a PCB or a card. Chips that directly solder to a second level package are called a **Chip-on-Board (COB)**. The solder that connects chip carriers to the second level package is usually connected to a copper pad and protected with an epoxy known as **underfill**. Underfill helps protect the solder and also allows for a higher level of thermal conductivity between the chip and the substrate than air would. **Solder mask** is a typically green lacquer that is also present between solder pads that makes it very difficult for solder to flow between solder pads, preventing any unwanted bridging between solder pads that are geometrically close together. A silicon passivation layer also exists on the chip that acts as an additional protective layer. The underfill provides a strong bond with this passivation layer.

A series of these PCBs are cards connected together would be called a third level package, and an example of this would be a motherboard.
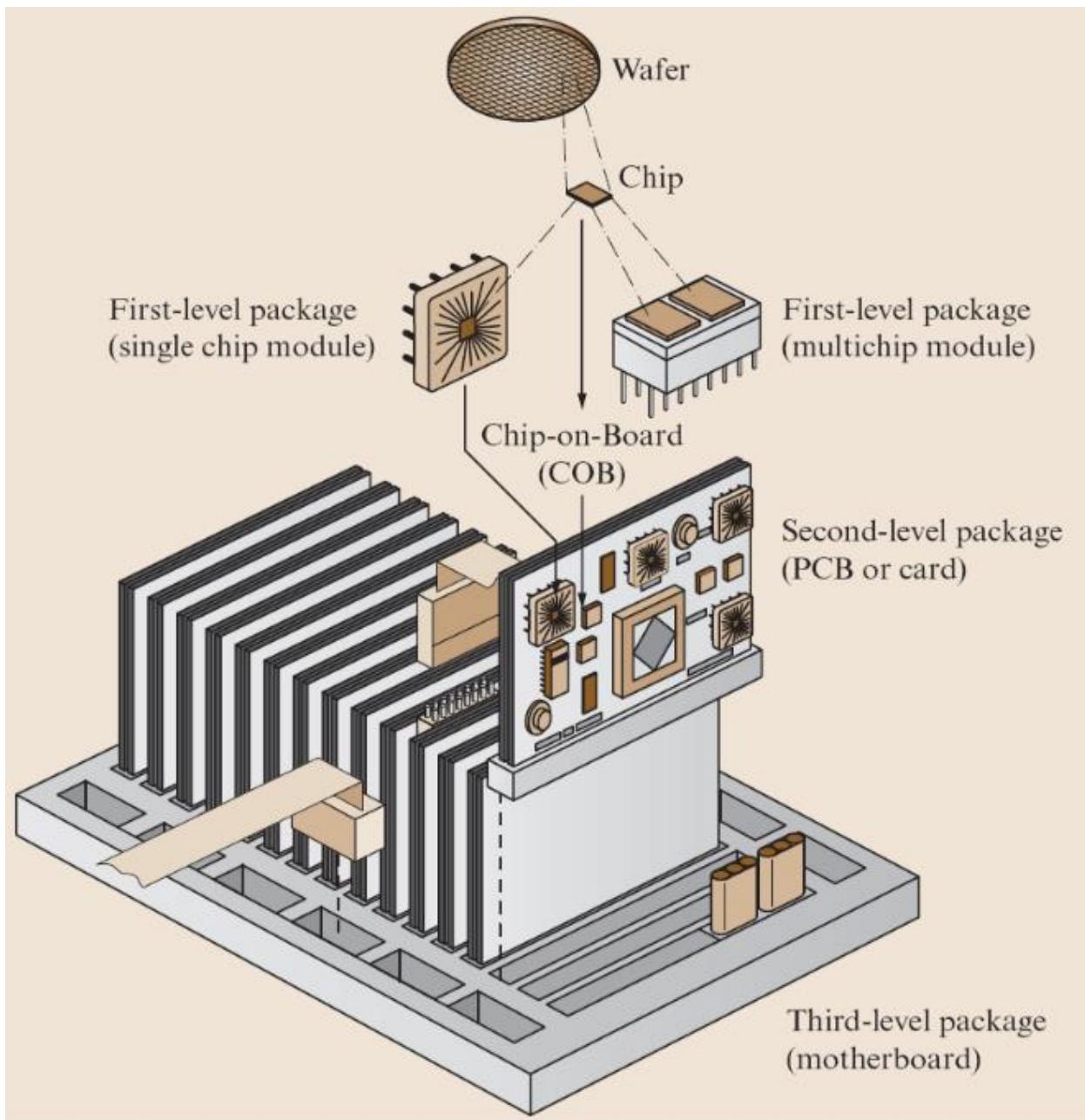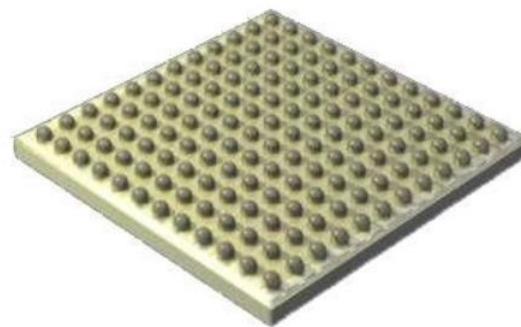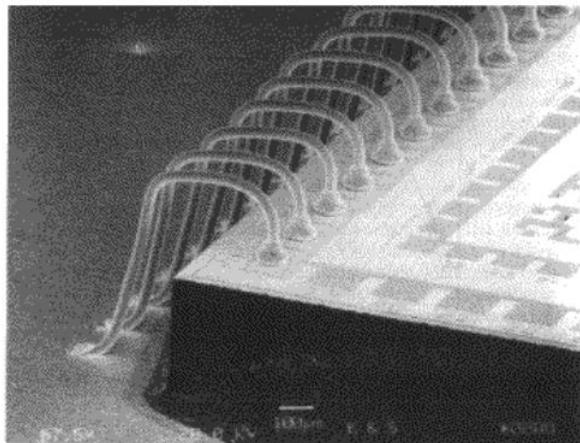
Figure 73. Hierarchy of Electronic Packages

Chips are connected to packages through chip interconnection technologies, of which the primary methods are

- **Wirebonding** (uses a perimeter array of pads on the chip)
- **Flip Chip Solder Bumps** (uses an area or perimeter array of "bumped" pads)
- **Tape Automated Bonding (TAB)** (uses a perimeter array of pads on the chip)
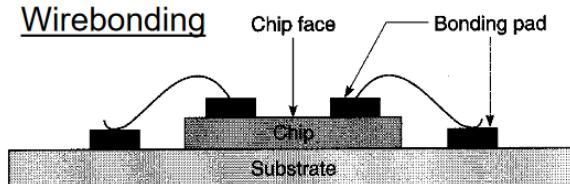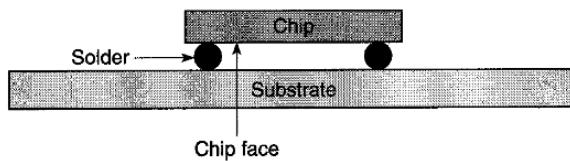
Figure 74. Chip Interconnection Technologies
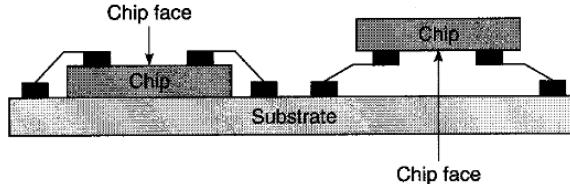
Chip-Level Interconnections Provide Electrical Paths to and from a Substrate for Power and Signal Distribution
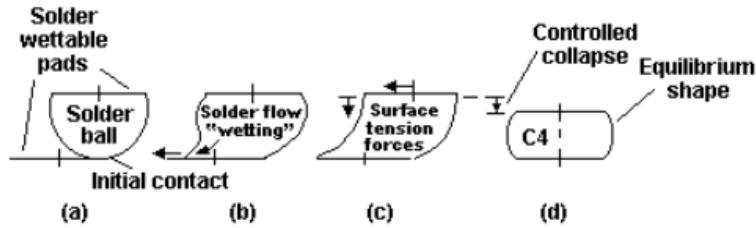
Flip Chip and TAB Require Bump Formation on the Surface of the Chip

Wirebonding is the dominant chip connection technology of today. Unlike TAB and Flip Chip bonding, wire bonds are formed one at a time rather than in bulk. Wirebonding has the benefit of being a flexible process, because wiring changes are easily done without retooling or die changes. There are two primary wire bonding techniques: Ball Bonding and Wedge Bonding. **Ball Bonding** accounts for over 90% of wire bonds seen today, and involves the use of a gold wire and required an operating temperature of about 150-200 degrees Celsius. Wedge Bonding is mostly used with aluminum wires, but can also be used with gold wires, and can be done at room temperature. Both procedures require significant amounts of force and oscillatory motion between 60-120 Hz.

Aluminum wires provide a more reliable bond to the aluminum pads that are located on the chip, but when exposed to humidity it is a risk of corrosion so gold wires are preferred, despite the increase in cost.

Flip chips have connects placed throughout the face of the chip (but could be used only as perimeter connects). They are typically regarded as a strong technology for the future because of their spatial efficiency. The solder bumps are placed onto the die through a process known as solder reflow and controlled collapse:

Flip chips are different than Ball Grid Arrays (BGA) because BGA packages are chips that are bonded to a substrate such as FR-4, which carries the solder ball on its bottom face. A flip chip has the solder balls placed directly onto the chip itself. The solder balls on a flip chip range in height between 15-50 mills, which is about 10% of the height of BGA solder balls.

Typical failure modes of an electronic package include solder joint fatigue, die fracture, severing of interconnections, wire bond failure, delamination of material interfaces, encapsulant fractures, etc. Pictures of these failures on a standard flip chip electronic package are provided below:
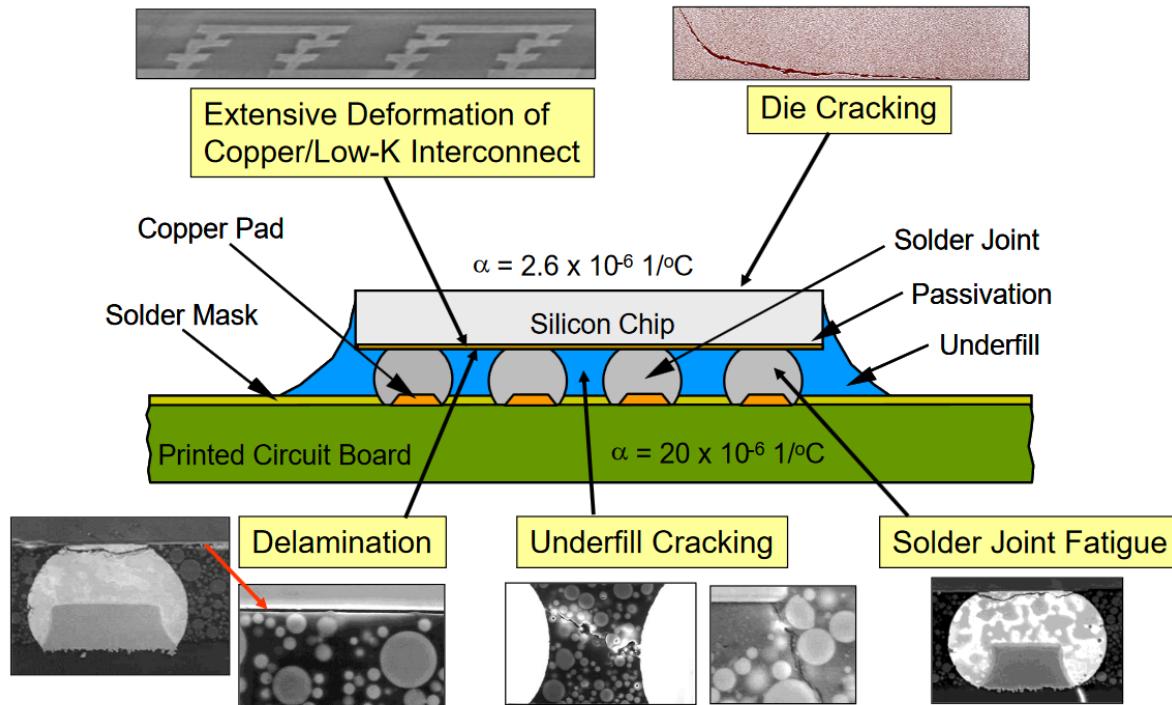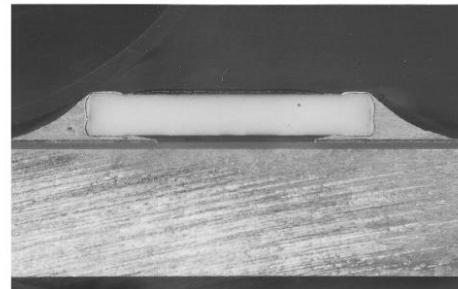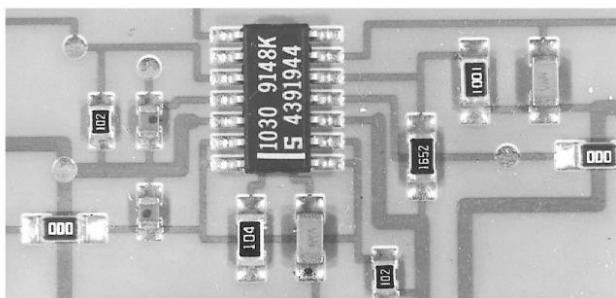

Figure 75. Failure Modes on an Electronic Package
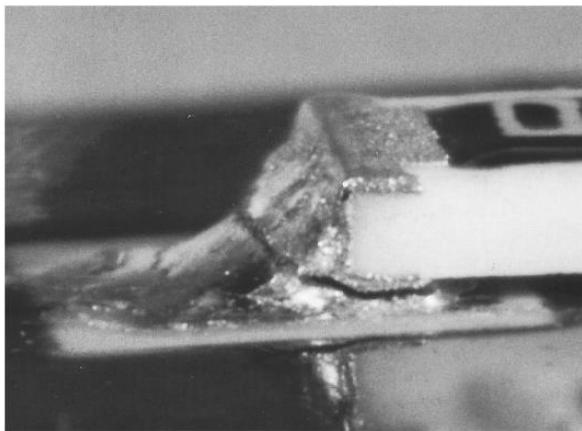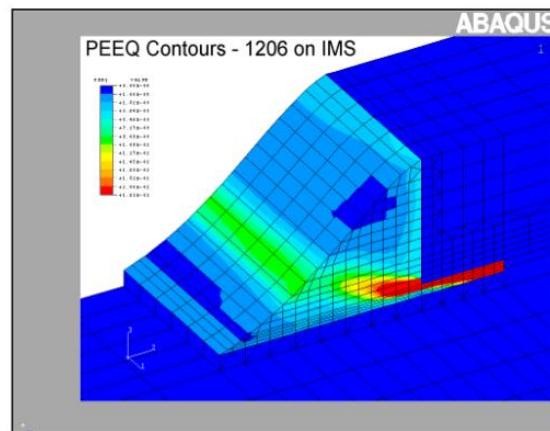
One of the biggest problems with solder joint fatigue comes from a big mismatch in thermal expansion coefficients during the thermal cycling of a component. Take the following SMT resistor:

Chip Resistor Cross-Section

| Material Name	α (CTE) /°C	E (MOE) (psi)
Aluminum Substrate	23.2×10^{-6}	10.1×10^{6}
FR-4 Substrate	20×10^{-6}	1.5×10^{6}
Copper	17.0×10^{-6}	19.2×10^{6}
Solder (63%-Sn/37%-Pb)	24.0×10^{-6}	5.0×10^{6}
Dielectric (Aluminum Substrate)	30.0×10^{-6}	1.16×10^{6}
Ceramic Al_2O_3 (Chip Resistor)	6.2×10^{-6}	44.0×10^{6}

Note the Large Mismatch in the Coefficients of Thermal Expansion of the Component and Substrate
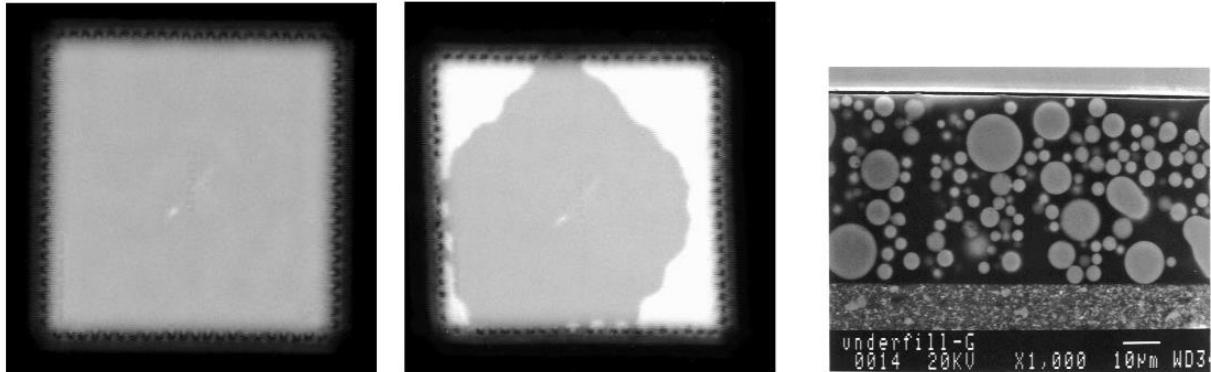
Stresses will occur as the resistor heats up and the substrate will try to expand significantly more than the resistor itself. Over time, the cycling of this can lead to a failure due to fatigue, as shown:

## Thermal Cycling Sample

Finite Element Simulation

For chips, which are very stiff and brittle, the great difference between expansion coefficients between the chip and substrate can lead to die cracking.

For chips with underfill, delamination of that underfill from the passivation layer will lead to heating issues on the chip. Delamination can be seen through the use of **C-Mode Scanning Acoustic Microscopy (C-SAM).** C-SAM provides a nondestructive test for finding air gaps in electronic packages. Using this technology, delamination can be seen as:

# Delamination of the Underfill to Die Interface
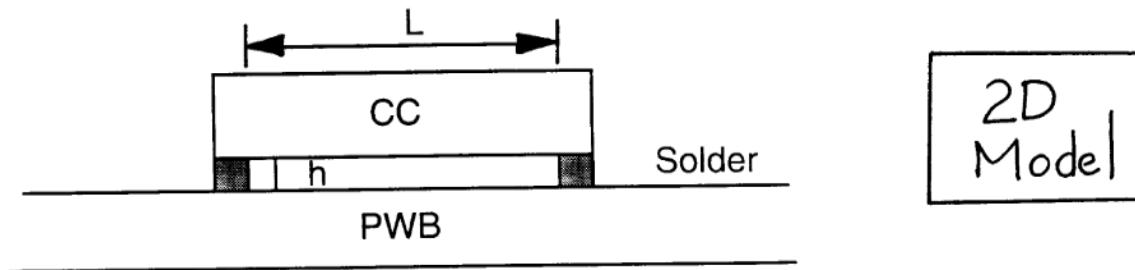
CSAM Images of Chip
Before and After Thermal Cycling

Another type of failure is called **popcorn failure**, which occurs when humidity causes water to become trapped in a component of the package which rapidly expands upon heating during operation.

One analytical method for analyzing solder joints that exists is called the **Distance to Neutral Point (DNP)** approach. This approach makes three key assumptions:

1. Solder joints can be represented as cubes
2. Solder offers no resistance to deformation (the cube has an elastic modulus of zero).
3. The chip and the printed wiring board are considered to be freely expanding.
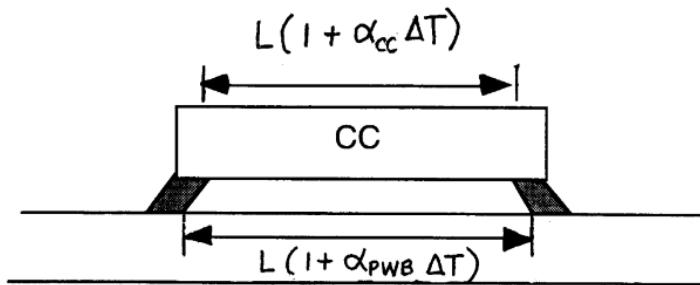
Free expansion implies that the strain is only related to the temperature difference using some reference temperature such that:


$$T = T_0 \quad (\text{Relaxed} / \underline{\text{Unstressed}})$$

$$\Delta T = T - T_o$$

$$\alpha_{cc} \ll \alpha_{PWB}$$

$$L(1 + \alpha_{cc} \Delta T)$$

| CC |

$$\boxed{T > T_o}$$

$$\Delta T > 0$$

$$L(1 + \alpha_{PWB} \Delta T)$$

$$L(1 + \alpha_{cc} \Delta T)$$

| CC |

$$\boxed{T < T_o}$$

$$\Delta T < 0$$
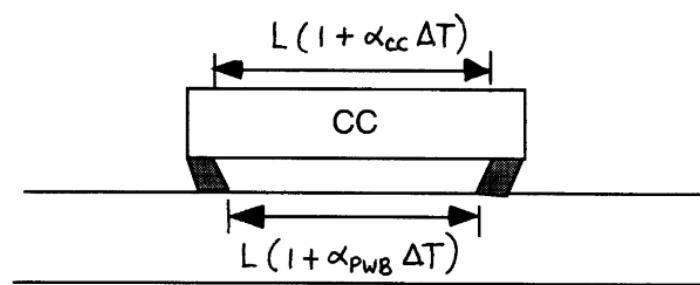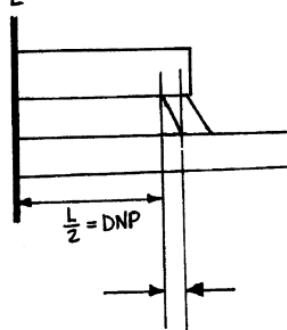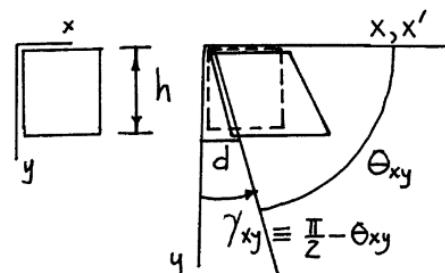
$$L(1 + \alpha_{PWB} \Delta T)$$

¢ — Centerline or "Neutral Point"

$$\frac{L}{2} = DNP$$

$$x$$
$$h$$
$$y$$

$$x, x'$$

$$\theta_{xy}$$

$$\gamma_{xy} \equiv \frac{\pi}{2} - \theta_{xy}$$

$$y'$$

$$\tan \gamma_{xy} = \frac{d}{h}$$

$$d = \frac{L(1 + \alpha_{PWB} \Delta T) - L(1 + \alpha_{cc} \Delta T)}{2}$$

$$d = \frac{L(\alpha_{PWB} - \alpha_{cc}) \Delta T}{2}$$

$$\gamma_{xy} \approx \frac{d}{h}$$

$$\boxed{\gamma_{xy} \approx \frac{L(\alpha_{PWB} - \alpha_{cc}) \Delta T}{2h}}$$

Linearly Dependent on $\frac{L}{2} = DNP$
(Distance to the Neutral Point)

(Part Elastic, Part Plastic)
↳ Fatigue

This analysis brings forth three key observations:

1. A bigger chip results in higher strains.
2. Taller solder joints result in lower strains (indicates that leadless chips have higher strains).
3. Bigger differences between expansion coefficients give bigger strains.

The DNP Formula explicitly states:
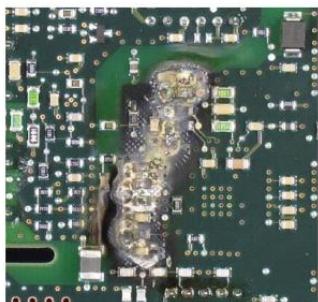
$$\gamma_{xy} = \frac{(\alpha_{PWB} - \alpha_{CC})\Delta T}{h}$$

Typically, the number of cycles until failure from thermal cycling is given as a function of plastic shear strain per cycle. Perhaps the most useful of these equations is the **Coffin-Manson relationship**:

$$N_f = 1.29(\Delta\gamma_p)^{-1.96}$$

As a general note, for compliant leads the leg will absorb some of the mismatch between thermal coefficients and the DNP solution will overestimate the strain. When no underfill is present for something like a flip chip, the DNP is actually very good at predicting trends. However, the presence of underfill causes the predictions to be inaccurate. The DNP provides good solutions for passive chip components like SMT resistors and capacitors and Leadless Chip Carriers as well as Ceramic Ball Grid Arrays.
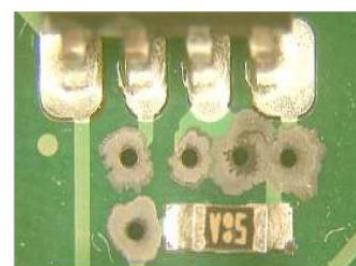
Other failure modes also exist that are not of a mechanical nature, some of which are shown below:

Electrical Shorting Failures   Power Transistor Failure   Corrosion

Figure 76. Non-Mechanical Failures of Electrical Components

The substrate that a die will bond to is often called a Printed Circuit Board (PCB) or Printed Wiring Board (PWB). These typically have the following composition:
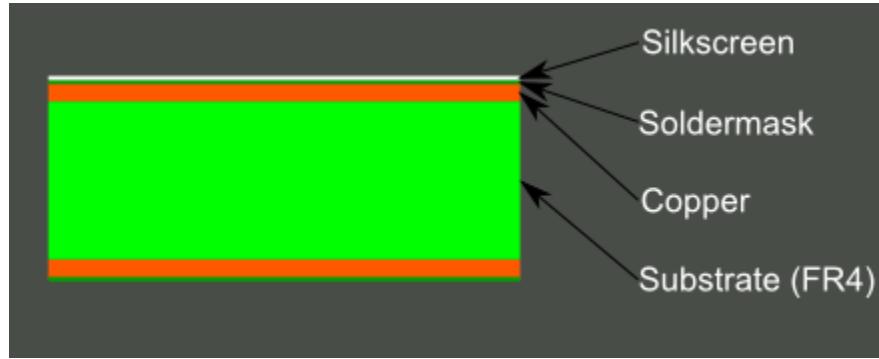

Figure 77. PCB Diagram

The base material, or substrate, is usually fiberglass. Historically, the most common designator for this fiberglass is **FR4 (Flame Retardant 4)**. This solid core gives the PCB its rigidity and thickness. There are also flexible PCBs built on flexible high-temperature plastic (Kapton or the equivalent). The next layer is a thin copper foil, which is laminated to the board with heat and adhesive. On common, double sided PCBs, copper is applied to both sides of the substrate. In lower cost electronic gadgets, the PCB may have copper on only one side. Soldermask is placed on top of the copper, and is most commonly green in color, but nearly any color is possible. The white silkscreen layer is applied on top of the soldermask layer. The silkscreen adds letters, numbers, and symbols to the PCB that allow for easier assembly and indicators for humans to better understand the board.

Usually, a PCB will have several copper layers that are connected to through holes and other layers through points called **vias**. *Tented vias* are covered by soldermask to protect them from being soldered to. Vias where connectors and components are to be attached are often *untented* (uncovered) so that they can be easily soldered.

Conclusions

This document is by no means a complete overview of every single aspect of microcontrollers, but it does provide a very firm introduction that will allow a new user to utilize almost all features of a microcontroller. A more complex understanding of microcontrollers can be obtained by searching through the code that comprises some of these more commonly used libraries and seeing exactly how they function.

As the author of this document, I do hope that it finds you as a senior design student at Auburn University and offers some assistance to you. If you aren't a senior design student, or even a student at Auburn University, I sincerely hope that you also find some value within these pages.

Additional Resources

Adafruit Website – **sells sensors with associated code and libraries needed to make them work (all open source, including the circuitry on the boards)**

SparkFun Website – **sells a variety of useful electronic components**

Omega Website – **great for load cells and pressure transducers**

GitHub Website – **location where many upload user-made libraries for public use**

EAGLE Board Designer Tutorial – **tutorial on the program EAGLE, which enables the user to prototype their own PCBs (Printed Circuit Boards)**

Arduino Shields – **page on Arduino's website dedicated to Arduino shields (PCBs that plug on top of the Arduino board to provide additional functionality)**

Tutorial on Oscilloscopes – **tutorial on how to use an oscilloscope, which is a device that is useful for seeing fast changes in voltages over time that cannot be found with a multimeter**

MATLAB Streaming Plotter – **tool for real-time plotting of data to a computer monitor from a microcontroller using a GUI created in MATLAB**