More Arduino Functions

- More functions:
  - Math functions
  - Trigonometry functions
  - Random numbers
  - Interrupts
    - Examples – Button push and an 8 bit counter.
- Addendum:
  - Characters
  - Bits & Bytes
  - Serial calls
    - Example - Absolute Orientation Sensor

Math Functions: abs()

- abs(x)
- Returns:
  - x if x >= 0
  - e.g. x=5, then abs(x)=5
  - -x if x is < 0
    - e.g. (x=-4, then abs(x)=4
**max()...**

- **max(x, y)**
  - Returns the larger of x vs. y
  - e.g. sensVal = max(sensVal, 20);
    - Assigns sensVal to the larger of sensVal or 20.
    - In this example effectively ensuring that it is at least 20.

**min()...**

- **min(x, y)**
  - Returns the smaller of two numbers
  - e.g. sensVal = min(sensVal, 100);
    - Assigns sensVal to the smaller of sensVal or 100.
    - In this example, ensuring that it never gets above 100.

**Constrain()...**

- **constrain(x, a, b)** - Constrains a number to be within a range.
  - x: the number to constrain, all data types
  - a: the lower end of the range, all data types
  - b: the upper end of the range, all data types
  - Returns
    - x: if x is between a and b
    - a: if x is less than a
    - b: if x is greater than b
  - For example, if `sensVal` was read as 140 from a sensor, the values assigned to `sensVal` in each case would be:
    - `sensVal = constrain(sensVal, 10, 150), answer 140`
    - `sensVal = constrain(sensVal, 50, 100), answer 100`
    - `sensVal = constrain(sensVal, 150, 250), answer 150`
**map()…**

- map(value, fromLow, fromHigh, toLow, toHigh)
  - Re-maps a number from one range to another. That is, a value of fromLow would get mapped to toLow, a value of fromHigh to toHigh, values in-between to values in-between, etc.
  - Use integer math (fractions are truncated).
  - Example:
    ```
    void setup() {}
    void loop()
    {
      int val = analogRead(0);
      val = map(val, 0, 1023, 0, 255);
      analogWrite(9, val);
    }
    ```

**pow()…**

- pow(base, exponent) - Value of a number raised to a power.
  - base: the number (float), exponent: the power to which the base is raised.
  - Example (serial print a binary sequence, 2 to the power of 0…49)
    ```
    float i = 0;
    long result;
    void setup()
    {
      Serial.begin(9600L);
    }
    void loop()
    {
      Serial.println("Base 2: ");
      for(i=0; i<50; i++)
      {
        result=pow(2, i);
        Serial.println(result);
      }
    }
    ```

**sq() and sqrt()…**

- sq(x)
  - The square of the number: \( x^2 \)
    - Returns a double

- sqrt(x)
  - Calculates the square root of a number.
    - Returns a double
Trigonometry

- **sin(rad)** - Calculates the sine of an angle (in radians). The result will be between -1 and 1.
  - `rad`: the angle in radians (float)
  - Returns the sine of the angle (double)

- **cos(rad)** - Calculates the cosine of an angle (in radians). The result will be between -1 and 1.
  - `rad`: the angle in radians (float)
  - Returns the cosine of the angle (double)

- **tan(rad)** - Calculates the tangent of an angle (in radians). The result will be between negative infinity and infinity.
  - `rad`: the angle in radians (float)
  - Returns the tangent of the angle (double)

Random Numbers

- **randomSeed(seed)**
  - Initializes the pseudo-random number generator, causing it to start at an arbitrary point in its random sequence.
  - This sequence is always the same.
  - For a better random seed, initialize with `analogRead()` on an unconnected pin.
- **random(max)** and **random(min, max)** are variations.

Interrupts

- Responsible for making the processor respond to important events.
  - This could be a button push, or other external event that you need react to immediately.
  - With the appropriate signal, the processor turns its attention to executing specific code related to the event.
  - The goal of an external interrupt is to not miss the event!
- Two external interrupt pins: INT0 and INT1 which map to Arduino Uno pins 2 and 3.
**ISR...**

- **Interrupt Service Routines (ISR)**
  - Fast, short code that is executed with an interrupt.
  - Only one can run at a time.
  - Takes no parameters and returns none.
  - Use global variables to pass information between the main program and the ISR.

**Implementing an ISR...**

- **Required code:**
  - `attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)`
  (Pin number, name of ISR, and mode - when should the interrupt be triggered.)
- **Interrupt trigger “modes.”**
  - **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.
  - **HIGH** to trigger the interrupt whenever the pin is high. (Due, Zero and MKR1000 boards)

**Interrupt Example...**

```
const byte LED = 13;
const byte BUTTON = 2;

// Interrupt Service Routine (ISR)
void switchPressed ()
{
  if (digitalRead (BUTTON) == HIGH)
      digitalWrite (LED, HIGH);
  else
      digitalWrite (LED, LOW);
}

void setup ()
{
  pinMode (LED, OUTPUT);  // so we can update the LED
  digitalWrite (BUTTON, HIGH);  // internal pull-up resistor
  attachInterrupt (digitalPinToInterrupt (BUTTON), switchPressed, CHANGE); // attach interrupt handler
}

void loop ()
{
  // loop doing nothing
}
```
**Example: 8-Bit Binary Counter**

**Binary Counter...**

```cpp
int button = 2;      // pin to connect the button
int presses = 0;    // variable to store number of presses
long time = 0;      // used for debounce
long debounce = 100;  // how many ms to "debounce"
const byte numPins = 8; // how many LEDs
int state;        // used for HIGH or LOW
byte pins[] = {5, 6, 7, 8, 9, 10, 11, 12};           // LED Pins

void count() // function count the button presses
{
    if(millis() - time > debounce)  presses++; // debouce pushbutton
    time = millis();
}

void setup()
{
    for(int i = 0; i < numPins; i++)          // set LED pins to outputs
    {
        pinMode(pins[i], OUTPUT);
    }
    pinMode(button, INPUT);
    attachInterrupt(0, count, LOW);     // pin 2 is interrupt 0 on UNO
}

void loop()
{
    /* convert presses to binary and store it as a string */
    String binNumber = String(presses, BIN);
    int binLength = binNumber.length(); //get length of string
    if(presses <= 255) // if we have less or equal to 255 presses
    {
        for(int i = 0, x = 1; i < binLength; i++, x+=2)
        {
            if(binNumber[i] == '0') state = LOW;
            if(binNumber[i] == '1') state = HIGH;
            digitalWrite(pins[i] + binLength - x, state);
        }
    }
}
```

Marian, P. Arduino 8 bit Binary LED Counter. www.electroschematics.com

**Summary**

- More functions:
  - Math functions
  - Trigonometry functions
  - Random numbers
  - Interrupts
  - Examples – Button push and an 8 bit counter.

- Addendum:
  - Characters
  - Bits & Bytes
  - Serial calls
  - Example - Absolute Orientation Sensor.
**Characters**

<table>
<thead>
<tr>
<th>Boolean tests of (thisChar):</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. isAlphaNumeric()</td>
</tr>
<tr>
<td>2. isAlpha()</td>
</tr>
<tr>
<td>3. isAscii()</td>
</tr>
<tr>
<td>4. isWhitespace()</td>
</tr>
<tr>
<td>5. isControl()</td>
</tr>
<tr>
<td>6. isDigit()</td>
</tr>
<tr>
<td>7. isGraph() [printable]</td>
</tr>
<tr>
<td>8. isLowerCase()</td>
</tr>
<tr>
<td>9. isPrintable()</td>
</tr>
<tr>
<td>10. isPunct()</td>
</tr>
<tr>
<td>11. isSpace()</td>
</tr>
<tr>
<td>12. isUpperCase()</td>
</tr>
<tr>
<td>13. isHexadecimalDigit()</td>
</tr>
</tbody>
</table>

**Bits and Bytes**

1. `lowByte(x)` - Extracts the low-order (rightmost) byte of a variable (e.g., a word).
2. `highByte(x)` - Extracts the high-order (leftmost) byte of a word (or the second lowest byte of a larger data type).
3. `bitRead(x, n)` - Reads a bit of a number.
4. `bitWrite(x, n, b)` - Writes a bit of a numeric variable.
   - `x`: the numeric variable to which to write
   - `n`: which bit of the number to write, starting at 0 for the least-significant (rightmost) bit
   - `b`: the value to write to the bit (0 or 1)
5. `bitSet(x, n)` - Sets (writes a 1 to) a bit of a numeric variable.
   - `x`: the numeric variable whose bit to set
   - `n`: which bit to set, starting at 0 for the least-significant (rightmost) bit
6. `bitClear(x, n)` - Clears (writes a 0 to) a bit of a numeric variable.
   - `x`: the numeric variable whose bit to clear
   - `n`: which bit to clear, starting at 0 for the least-significant (rightmost) bit
7. `bit(n)` - Computes the value of the specified bit (bit 0 is 1, bit 1 is 2, bit 2 is 4, etc.).
Serial Calls

1. `Serial.if (Serial)` - Indicates if the specified Serial port is ready.
2. `available()` - Get the number of bytes (characters) available for reading from the serial port. This is data that's already arrived and stored in the serial receive buffer (which holds 64 bytes). available() inherits from the Stream utility class.
3. `availableForWrite()` - Get the number of bytes (characters) available for writing in the serial buffer without blocking the write operation.
4. `begin()` - Sets the data rate in bits per second (baud) for serial data transmission.
5. `end()` - Disables serial communication, allowing the RX and TX pins to be used for general input and output.
6. `find(target)` - Reads data from the serial buffer until the target string of given length is found. The function returns true if target string is found, false if it times out.
7. `findUntil(target, terminator)` - Reads data from the serial buffer until a target string of given length or terminator string is found.
8. `flush()` - Waits for the transmission of outgoing serial data to complete.
9. `parseFloat()` - The first valid floating point number from the Serial buffer. Characters that are not digits (or the minus sign) are skipped. Terminated by the first character that is not a floating point number.
10. `parseInt()` - Looks for the next valid integer in the incoming serial stream.
11. `peek()` - Returns the next byte (character) of incoming serial data without removing it from the internal serial buffer.
12. `print()` - Prints data to the serial port as human-readable ASCII text.
13. `println()` - Prints data to the serial port as human-readable ASCII text followed by a carriage return character. Syntax: `Serial.println(val)` or `Serial.println(val, format)`.
   - `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)
14. read() - Reads incoming serial data.

15. readBytes(buffer, length) - Reads characters from the serial port into a buffer.
   - The function terminates if the determined length has been read, or it times out.
   - Returns the number of characters placed in the buffer.
   - A 0 means no valid data was found.

16. readBytesUntil(character, buffer, length) - Reads characters from the serial buffer into an array.
   - The function terminates if the terminator character is detected, the determined length has been read, or it times out.

17. readString() – Reads characters from the serial buffer into a string. The function terminates if it times out.

18. readStringUntil(terminator) - Reads characters from the serial buffer into a string. The function terminates if the terminator character is detected or it times out.

19. setTimeout() – Sets the maximum milliseconds to wait for serial data.

21. write() - 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. Syntax: Serial.write(val); Serial.write(str); Serial.write(buf, len).
   - val: value to send as a single byte
   - str: string to send as a series of bytes
   - buf: an array to send as a series of bytes
   - len: the length of the buffer

22. serialEvent() - Called when data is available.
Utilities: sizeof() & PROGMEM

- `sizeof(variable)` - The `sizeof` operator returns the number of bytes in a variable type, or the number of bytes occupied by an array.
- `PROGMEM` - Store data in flash (program) memory instead of SRAM. There's a description of the various types of memory available on an Arduino board.
  - The `PROGMEM` keyword is a variable modifier, it should be used only with the datatypes defined in `pgmspace.h`. It tells the compiler “put this information into flash memory”, instead of into SRAM, where it would normally go.
  - `PROGMEM` is part of the `pgmspace.h` library that is available in the AVR architecture only. So you first need to include the library at the top your sketch, like this: `#include <avr/pgmspace.h>`
  - Syntax: `const datatype  variableName[] PROGMEM = {data0, data1, data3...};`

Absolute Orientation Sensor BNO005

[Image of BNO005 sensor]

Serial Monitor

[Image of serial monitor window]