**ARDUINO Programming basics 2017**

**Index**

**1. The Arduino Software Interface 2**

Arduino Software Interface **2**

Connect to your Arduino board **3**

**2. Arduino Basic Code 4**

**3. Variables 5**

**4. Serial Communication 7**

**5. Inputs and Outputs 9**

**6. Loops 10**

While loop **10**

For loop **11**

**7. Conditional Statements 12**

if Statement **13**

Comparison Operators **13**

if / else if / else / Statement  **14**

Boolean Operators **15**

**8. Switch / Case statements 16**

**9. Arrays 17**

Single Arrays **17**

2D Arrays **19**

**10. The Map Function 20**

**11.The Constrain function 21**

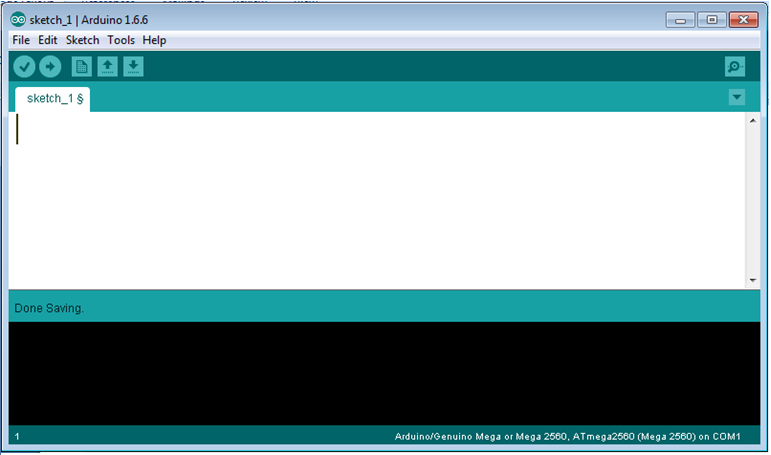
**12. Delay and Timing 22**

**13. Functions 24**

**14. bitRead() 26**

**1. The Arduino Software Interface**

The Arduino interface is pretty “bare-bones”. When you load the software, the first screen you will see is a white window (shown below) with several different shades of blue and blue-green as border.  Arduino projects are called “sketches” and when you start a new sketch several additional files are also created.



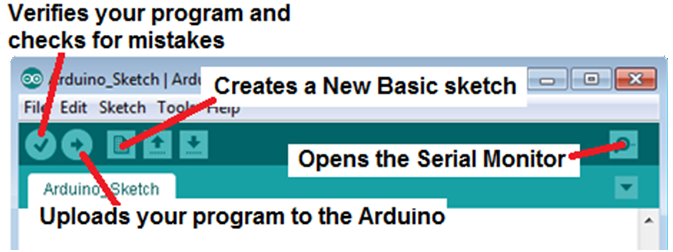
**Arduino Interface**

The Main Menu headings are labeled “File”, “Edit” ,“Sketch”, “Tools”, “Help”.

The File and Edit menus are typical Windows menus and most commands will be familiar. Unique features include the Examples option in the File menu which allows you to load premade programs. There are 2 main categories, the "Built in Examples" that come preinstalled with Arduino and the "Examples from Custom Libraries" which are installed when you add a Library from a Zip file. These programs are very useful for creating base programs that you can build on. Be aware that you should always use the "Save As" feature to prevent modifying the original code.

The Sketch option has the basic compile and upload functions and also provide a way to easily include libraries and external code such as bitmaps into your program.

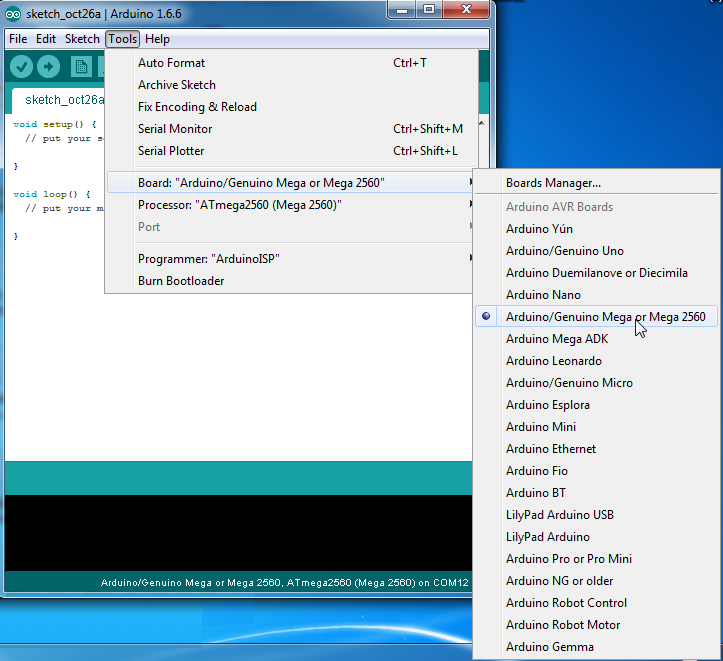
The Tools option is used to set up your computer to communicate with your arduino. See steps 4 and 5 in the next section, Connect to your arduino Board.



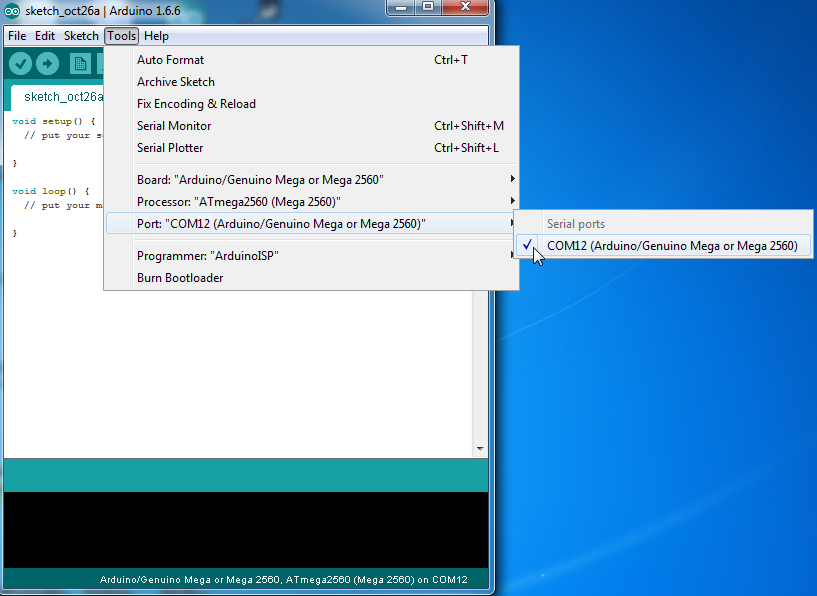
Note that all these icon functions are also available from the main menus

**Connecting an Arduino to** **your computer**

1. Launch the Arduino software by double-clicking the Arduino icon.
2. Plug one end of the USB into the Arduino and the other end into your computer.
3. If this is the first time connecting your Arduino the computer should detect the new device and tell you if it has installed correctly. After the first time there will not be any acknowledgement that you are connected.
4. In the software, select “Tools” -> “Board” -> you will get a list of possible boards. Select “Arduino/Genuino Mega or Mega 2560”. (see below)



1. In the software, select “Tools” -> “Serial Port” -> COM # (there may be several com ports listed- select the one that has Arduino/Genuino Mega or Mega 2560 beside it).

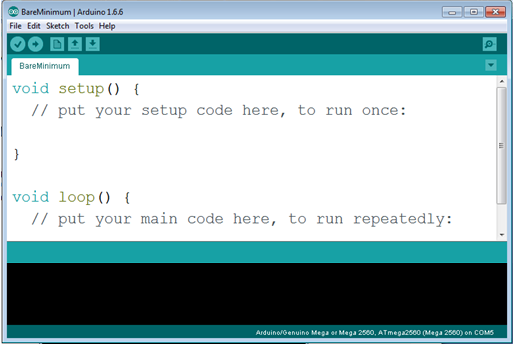


**2. Arduino Basic Code**

You’re now ready to start coding!

The Arduino language is CASE SENSITIVE: a capital letter is not the same as a lower case letter. The following code represents the minimum in order for a program to compile (to load this program select File in the main menu then

> Examples > Basics > Bare Minimum):



All Arduino Sketches must have 2 main components, the Void Setup and the Void loop.

The “**void setup()**” section is used to set up the Arduino before running the main code. This is where you create variables, tell the Arduino which pins will be used (pin modes), set the serial baud rate (communication standards) and other related instructions that tell the Arduino how to operate. The software only goes though this section once.

The “**void loop()**” section is the main part of the code that performs whatever function you assign it. It loops back onto itself so it will repeat the code over and over again.

Almost every line of code needs to end with a semicolon ‘;’ (there are a few exceptions which we will see later).

When you write code it is very helpful to be able to add comments to help you keep track of what you are doing. To write single line comments in the code, type two back slashes followed by the text:

//comments are overlooked when running your program- use them to help you keep your program organized

Anything written after the // will be ignored when the sketch actually runs.

To write multi-line comments, start the comment with /\* and end with \*/

/\* This is a multi-line comment and saves you having to always use double slashes at the beginning of every line. Comments are used to explain the code textually. Good code always has a lot of comments. all comments will appear in green text.\*/

**3. Variables**

A variable is a way of naming and storing a value for later use by the program, such as data from a sensor or an intermediate value used in a calculation. Variables allow you to easily change values while your program is running.

You should give your variables descriptive names, so as to make your code more readable. Variable names like **tiltSensor** or **pushButton** help you (and anyone else reading your code) understand what the variable represents. Variable names like **var** or **value**, on the other hand, do little to make your code readable.

You can name a variable any word that is not already one of the keywords in Arduino. Avoid beginning variable names with numeral characters.

#### Declaring Variables:

Before they are used, all variables have to be declared. Declaring a variable means defining its type, and optionally, setting a starting value (initializing the variable). There are several types of numerical variables that can be declared, each one best suited to an expected value. When a variable type is declared a certain amount of memory is set aside to contain all possible values of that variable. By using a type that is the smallest possible, memory usage is conserved and programs will run slightly faster.

* Char Characters such as letters or symbols - has no numerical value
* Byte Any 8-bit binary number, with a value from 0 (00000000) to 255 (11111111)
* Int Any whole number between -32,768 and 32,767. This is the most common variable.
* unsigned int Any whole positive number between 0 and 65,535.
* long Extended size variable for numbers stored in 32 bits of memory. From -2,147,483,648 to 2,147,483,647)
* unsigned long Extended size variable for positive number stored in 32 bits of memory from 0 to 4,294,967,295
* float A number that has a decimal point, accurate to 7 digits. 32 bits of storage.
* double Double precision floating point number, stored in 64 bits of memory on some Arduino boards. On Mega 2560 a double is only 32 bit and does not add precision so Float is normally used.

The most commonly used is the Int (Integers). To declare a int variable use the following format:

int a; // sets up the letter “a” as a variable with no current value

int numRedBlinks; // sets up the word “numRedBlinks” as a variable with no current value. Note how the name can //describe what the variable’s usage will be.

Variables do not have to be initialized (assigned a value) when they are declared, but it is often useful.

int a = 1; same as above but now “a” has a starting value of 1. This value can be changed in the program.

Programmers should consider the size of the numbers they wish to store in choosing variable types.

When variables are made to exceed their maximum capacity they "roll over" back to their minimum capacity, note that this happens in both directions.

int x;

x = -32,768;

x = x - 1; // x now contains 32,767 - rolls over in neg. direction

x = 32,767;

x = x + 1; // x now contains -32,768 - rolls over

#### Variable Scope

Another important choice that programmers face is where to declare variables. The specific place that variables are declared influences how various functions in a program will see the variable. This is called variable [scope](https://www.arduino.cc/en/Reference/Scope).

#### Initializing Variables

Variables may be initialized (assigned a starting value) when they are declared or not. It is always good programming practice however to double check that a variable has valid data in it, before it is accessed for some other purpose.

Example:

int counterValue = 17; // declares variable named counterValue and sets initial value at 17

#### Using Variables

Once variables have been declared, they are used by setting the variable equal to the value one wishes to store with the assignment operator (single equal sign). The assignment operator tells the program to put whatever is on the right side of the equal sign into the variable on the left side.

inputVariable1 = 7; // sets the variable named inputVariable1 to 7

inputVariable2 = analogRead(2); // sets the variable named inputVariable2 to the input voltage read from analog pin #2

#### Examples

int lightSensVal;

char currentLetter;

unsigned long speedOfLight = 186000UL;

char errorMessage = {"choose another option"}; // see string

Once a variable has been set (assigned a value), you can test its value to see if it meets certain conditions, or you can use its value directly. For instance, the following code tests whether the inputVariable2 is less than 100, then sets a delay based on inputVariable2 which is a minimum of 100:

if (inputVariable2 < 100)

{

inputVariable2 = 100;

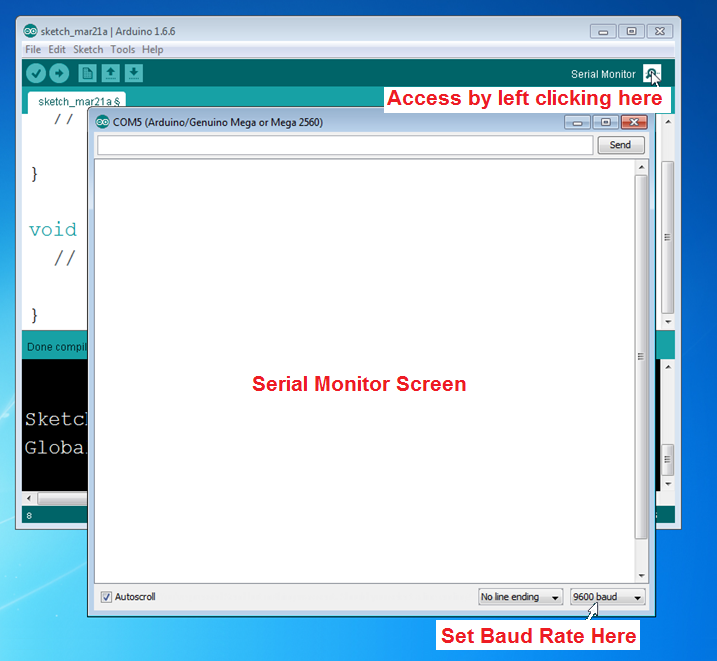
}

delay (inputVariable2);

This example shows all three useful operations with variables. It tests the variable ( if (inputVariable2 < 100) ), it sets the variable if it passes the test ( inputVariable2 = 100 ), and it uses the value of the variable as an input parameter to the delay() function ( delay(inputVariable2) )

**4. Serial Communication**

The Arduino board can communicate with a Virtual Monitor that can be used to display information related to your program. This is extremely helpfull when you need to debug a program and want to see what values your code is processing. Your Arduino and the Serial monitor need to communicate at the same speed for them to understand each other. This speed is known as baud rate and is measured in bits per second. The baud rate must be set properly for the board to convert incoming and outgoing information to useful data. If your Serial Monitor is expecting to communicate at a baud rate of 2400, but your transmitter is transmitting at a different rate (for example 9600), the data you get will not make sense. To set the baud rate in the Serial Monitor left click on the drop down menu and select the value that matches your code.



To tell arduino that you are going to use a serial monitor in your program and set the baud rate use the following code:

void setup() {

**Serial**.begin(9600); // 9600 is the standard rate and is recommended for stable communication

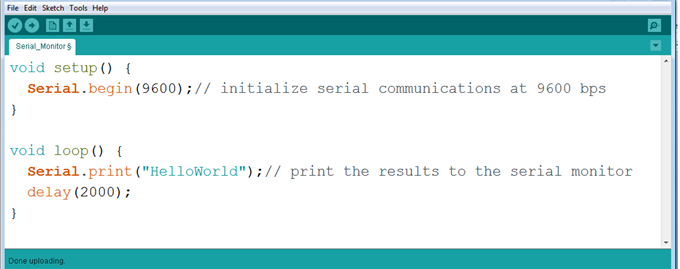
}

To send information out to the serial monitor use the following code in the Void Loop section of code:

**Serial**.print ( ); // use this for displaying numbers or variable values

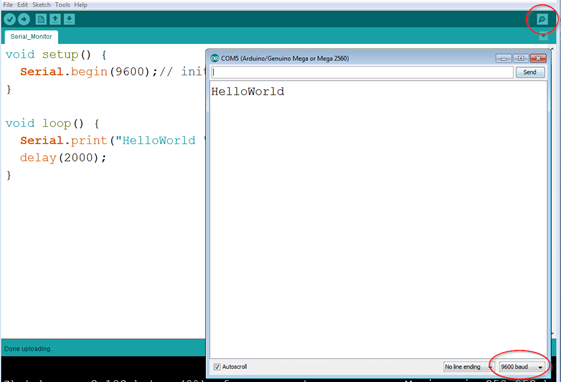
**Serial**.print (" " ); // use this for displaying words or symbols

Using either of the above will print directly after the last output on the same line. To print somthing on the next line use the command: **Serial**.println();



Create the above program by first selecting File > New. Then type in the extra lines as shown in the above diagram. Press the “verify” button (looks like a check mark just below “File”). The verify function checks your programs to make sure it will operate correctly before uploading to your Arduino. You should not get any errors at the bottom of the screen. If you get errors, check the syntax (spelling and capitalization). Note that it is better to type in the code yourself than copy/pasting text from another program which may include unwanted formatting.

Next, upload the sketch to the board using the “Upload to I/O Board” button (arrow pointing right- beside the verify button). Wait until the sketch has finished uploading. Open the “Serial Monitor using the magnifying glass icon in the upper right of the screen. Make sure Baud rate in serial monitor is set same value as in your program. (as shown in the red circle at lower right corner of diagram below).



**5. Inputs and Outputs**

If you were to sum up the basic purpose of the Arduino system it would be to use programming to control electronic devices. Examples of this can be as simple as turning an LED on and off all the way up to sophisticated systems such as controlling a Quad copter with self stabilizing flight, GPS guidance and Video first person control. In both of those cases the Arduino uses its pins to connect to outside electronic components. These components usually fall into one of two categories, inputs or outputs. An output is something the arduino will control such as an LED, a motor, a speaker, an LCD display etc. An input gives information to the Arduino and include sensors and switches. To control an output device or read an input you must first tell the Arduino which pin(s) the device is connected to and then specify that it will be considered as an output or input. This is done with the pinMode command. Once the pins have been assigned as being an input/output you can then turn these pins on and off using the Commands such as digitalWrite or read their value with digitalReadand analogRead.

**OUTPUTS:**

To help understand OUTPUTS we will use an example program that uses the pinMode and digitalWrite commands. Open the Arduino software and go to File > Examples > Basics > Blink LED.

Understanding the Blink LED Code

The three new lines of code you have not seen before are:

pinMode(13, OUTPUT);

The PinMode command establishes what the function of each of the Arduino pins #1 to 53 will be, either an OUTPUT or an INPUT. Note that in the key word pinMode the ‘M’ is capitalized. A lower case ‘m’ would cause the word “pinmode” to not be recognized. The word OUTPUT or INPUT must have all letters in upper case. In this program pin 13 will be declared an output and when the program turns the pin on (also known as setting the pin "HIGH") it will have 5 volts applied to it. This will be used to turn on a red LED.

digitalWrite(13, HIGH); and digitalWrite(13, LOW);

The line digitalWrite(pin, HIGH) is the part of the code that activates pin 13 by applying +5V to it (pin 13 on). If you were to replace HIGH with LOW, the pin would then be set to 0V (pin 13 off). These are the commands that turn the led on and off. Note that the ‘W’ is capitalized.

delay(1000);

The delay(1000); line causes the program to wait for 1000 milliseconds (which = 1 second) before proceeding. Note that during a delay, the microcontroller simply waits and does not execute any additional lines of code.

**INPUTS:**

Arduino pins can also be used to perform the opposite function which is to be an input. An input takes an electrical signal from an outside source such as a sensor or switch and feeds it into the program. You can then use that input to control the program, including the outputs. An example would be a simple on/off button switch could be used to turn an LED on or off.

Inputs must also be declared using a pinMode command:

pinMode(5, INTPUT);

### Obtaining input information requires a different coding method than using simple outputs such as turning an LED on with a digitalWrite command. Because the Arduino does not Know when the input will be activated it must constantly read the input pin and react once it sees an appropriate signal. The basic Arduino already has a built in Void loop that can be used as long as it is ok to repeat *everything* the program does. If you need your switch input to control only part of the code you will have to use a separate loop contained within the Void loop. The most common method is to use the while loop. (See CE3A5.0.5 Arduino Programming Guide\_2017.docx pg 10 for detailed usage).

### while (digitalRead(5) ==0) {

### digitalWrite(13,LOW);

### }

### In the above code the while loop keeps an LED turned off (LOW) as long as a Switch connected to pin 5 is not being pressed. The way a while loop works is it looks at the statement contained in the brackets, (digitalRead(5)==0) and determines if the statement is true or false. In this code a digital read command looks at pin 5 and compares its value to 0 (or LOW).If the statement is true which means there is no signal from the switch because it is not pressed, the code in between the curly braces, digitalWrite(13,LOW); will be executed over and over. This will continue indefinitely until the statement is read as false. The statement will become false when you press the switch and a 1 or HIGH signal will be read at pin 5 due to the switch now providing 5 volts. When the program sees the statement is false the program will stop looping and ignore the code in between the curly braces and move on to the next line of code in the program. The while loop is known as a conditional staement and is a very powerful part of programming because the program can now make decisions instead of just blindly following commands.

**6. Loops**

Computer programs in their most simple form start by executing the first line of code and then process all the remaining lines of code one after another until stopping at the last line. The majority of programs however must repeat all or part of the code many times. You saw this in your first "Blink" program where the LED continuosly turned on and off. The most common method of repeating code is to put it in a loop. The basic Arduino code already has a built in section called void loop() which repeats all the code within its 2 brackets { }. To only loop a specific section requires new methods.

## The *while* () Statement

**while** loops will loop continuously, and infinitely, until the condition inside the parenthesis ( ) becomes false. Something must change the tested variable, or the **while** loop will never exit. This could be in your code, such as an incremented variable, or an external condition, such as a switch input.

while(condition){

// do something here if condition is true

// when condition becomes false get out of loop and continue program after bracket in next line

}

// cont

#### Example

var = 0;  
while(var < 20){ *// repeats code within { } until expression (var>20) is no longer true*  
  *// insert code here that you want do something repetitive 20 times-example blink an LED*  
  var++; *// adds 1 to the variable "var"each time program goes through loop*  
}

**The"for" Statement**

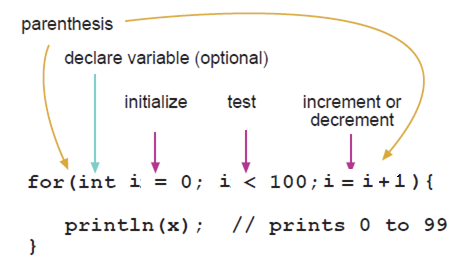
Many times we want to repeat a part of our code a certain amount of times. Instead of copying the code over and over to achieve this we can use the much more efficient “for” statement. The **for** statement is used to repeat any block of code that is enclosed in the pair of curly braces immediately following it. There are 3 parts to a “for” statement: Initialization, Test, and Increment and has the form:

**for** (**initialization**; **test**; **increment**) {

//statement(s);

}

You start by initializing (declaring) a variable and give it a starting value. This variable is used to keep track of how many times the for statement goes through its loop.You can use any variable you want but very often the letter i or j is used. The value is usually set to a starting value of 0 or 1 but is not limited to those 2 numbers. As the “for” statement is executed it looks at the test statement to determine how many times to go through the loop. Once the initialized variable reaches this value the program ignores the code it was repeating and goes to the next line in the program. The third section is the increment counter which is used keep track of how many times the code has been repeated. Each time the code is repeated the increment counter will increase (or decrease if set up that way). Typically you increase or decrease the value by 1 but any math expression can be used. There are three parts to the **for** loop header:



The **initialization** happens first and exactly once. Each time through the loop, the **condition** is tested; if it's true, the statement block, and the **increment** is executed, then the **condition** is tested again. When the **condition** becomes false, the loop ends.

#### Here is an example of a for statement that dims an LED using a PWM pin

void loop()

{

for (int i=0; i <= 255; i++){ // the program will repeat this code 255 times. Note: i++ is the same as i=i+1

analogWrite(PWMpin, i); // brightness increases by using i to also set the analog output value

delay(10);

}

}

The C **for** loop is much more flexible than **for** loops found in some other computer languages, including BASIC. Any or all of the three header elements may be omitted, although the semicolons are required. Also the statements for initialization, condition, and increment can be any valid C statements with unrelated variables, and use any C datatypes including floats. These types of unusual **for** statements may provide solutions to some rare programming problems.

For example, using a multiplication in the increment line will generate a logarithmic progression:

for(int x = 2; x < 100; x = x \* 1.5){

println(x);

}

Generates: 2,3,4,6,9,13,19,28,42,63,94

Another example, fade an LED up and down with one **for** loop:

void loop()

{

int x = 1;

for (int i = 0; i > -1; i = i + x){ // will keep repeating code forever if i is greater than 1

analogWrite(PWMpin, i);

if (i == 255) x = -1; // once counter reaches 255 the increment will decrease because x now = -1

delay(10);

}

}

**7. Conditional Statements**

Conditional statements are one of the most useful programming features available. A conditional statement looks to see if a certain condition exists, (for example if a button is being pressed) and if the condition is true (yes, the button is being pressed) a specific action will be carried out (e.g. turn a green LED on). If the condition is not true,(e.g. the button is not being pressed) the program will skip the related command ( turning the green LED on) and go to the next program line. There are several different types of conditional commands but this guide will only cover the most common one, the "if" conditional statement.

***if*  Statement**

*if*, which is used with a comparison operator,(==, !=, <, > ), tests whether a certain condition has been reached, such as an input being above a certain number. The format for an *if* test is:

if (someVariable > 50)

{

// do something here

}

The program tests to see if someVariable is greater than 50. If it is, the program takes a particular action. Put another way, if the statement in parentheses is true, the statements inside the brackets are run. If not, the program skips over the code.

In one of your assignments you are asked to have one LED turn on while the other LED turns off when the button switch is pressed. This is a perfect situation for a conditional statement. Here is a description of what we want the program to do:

If the button attached to digital pin 2 is pressed (HIGH signal) then turn the red LED LOW and turn the green LED HIGH.

Assuming that we have declared variables for our output pins, (pin13 is redLed, pin11 is greenLed) and input pins (pin 2 is buttonPin ), the program will look like this:

if (buttonPin==high)

{

digitalWrite(redLed,LOW);

digitalWrite(greenLed,HIGH);

}

Additional Notes:

The statements inside the parentheses ( ) require the use of one or more operators:

**Comparison Operators:**

**x == y** (x is equal to y) \*see note below

**x != y** (x is not equal to y)

**x < y** (x is less than y)

**x > y** (x is greater than y)

**x <= y** (x is less than or equal to y)

**x >= y** (x is greater than or equal to y)

\*Note: Whats the difference between x=y and x==y? x==y Can be read as "if the value of x is the same as the value of y". The double equal sign is used to compare 2 values. The expression x=y is used to change the value of x to whatever y's value is, In other words it makes x equal to whatever Y currently represents.

The brackets { } may be omitted after an if statement if your conditional statement only has 1 line of code as shown in the first 2 examples below. A single line of can still be put in brackets as in example 3 but 2 or more statements MUST be contained in brackets as in example 4.

1. if (x > 120) digitalWrite(LEDpin, HIGH);

2. if (x > 120)

digitalWrite(LEDpin, HIGH);

3. if (x > 120){ digitalWrite(LEDpin, HIGH); }

4. if (x > 120){

digitalWrite(LEDpin1, HIGH);

digitalWrite(LEDpin2, HIGH);

}

***if / else* if / *else* / Statement**

if/else if/else statements allows greater control over the flow of code than the basic if statement, by allowing multiple tests to be grouped together. For example, an input from a Button Switch could be tested and one action taken if the button switch is pressed and another action taken if it is not. The code would look similar to this:

buttonState = digitalRead(buttonSwitch); // reads input from the switch (HIGH or LOW) and gives that value to variable //button State

if (buttonState == LOW)) // switch is not being pressed

{

digitalWrite(11, LOW); // green LED will turn off and Red will turn on if button not pressed (LOW)

digitalWrite(13, HIGH);

}

else

{

digitalWrite(11, HIGH); // red LED will turn off and green will turn on if button is pressed (NOT LOW)

digitalWrite(13, LOW);

}

In the above example the Arduino looks at the "if" statement and if it is true (button is not being pressed) the next block of code within the brackets is executed.(green LED off, red LED on). Any code in the "else" section (green LED on, red LED off) will be ignored and not executed. The program will then continue to any code immediately after the if/else statements.

If on the other hand the "if" statement is false (this happens when you press the button and buttonState is HIGH instead of LOW) the program will skip the code in brackets (green LED off, red LED on) and execute the block of code after the "else" statement (green LED on, red LED off) . Note that the else statement does not have any condition attached to it, it just executes what ever code is in the brackets ONLY when the "if" statement above it is false.

If you have more than 2 conditions use the format if/else if/else. The first conditional statement uses the standard "if" format, and each following condition use "else if" until it gets to the last condition in which case an "else" statement is used. The last "else" statement tells the Arduino that it has reached the last conditional statement. The if/else if/else format works the same way as the 2 condition if/else except there are more than 2 conditions. When any true test is found, its associated block of code within the brackets is run, and the program then skips to the line following the entire if/else construction. If no test proves to be true, the default else block is executed, if one is present, and sets the default behavior.

Note that an "else if"  block may be used with or without a terminating "else" block and vice versa. An unlimited number of such "else if" branches is allowed.

if (pinFiveInput < 500)

{

// do Thing A

}

else if (pinFiveInput >= 1000)

{

// do Thing B

}

else

{

// do Thing C

}

# Boolean Operators

These can be used inside the condition of an [if](https://www.arduino.cc/en/Reference/If) statement when you want have 2 conditions (operands) in the same statement. An example would be if you wanted an action to occur depending on the position of 2 switches.

### && (logical and)

Both conditions must be true: (condition A and condition B)

if (sensorValue(switch1) =HIGH && sensorValue(Switch2) =LOW) {

// do this

}

Switch 1 must be pressed and Switch 2 not pressed for the above statement to be true and execute commands between brackets.

### || (logical or)

Only one of the conditions must be true: (condition A or condition B)

if (sensorValue > 0 || sensorValue <341) {

// do this

}

sensorValue must be greater than 0 **or** less than 341 to execute commands between brackets

### ! (not)

The condition does not equal a specific value

if (!x) {

// ...

}

If the condition does not equal x execute commands between brackets

Another way to express branching, mutually exclusive tests, is with the [switch case](file:///C:\Program%20Files%20(x86)\Arduino\reference\www.arduino.cc\en\Reference\SwitchCase.html) statement.

**8. Switch / Case statements**

Like if statements, switch...case controls the flow of programs by allowing you to specify different code that should be executed in various conditions. In particular, a switch statement compares the value of a variable to the values specified in case statements. When a case statement is found whose value matches that of the variable, the code in that case statement is run.The break keyword exits the switch statement, and is typically used at the end of each case. Without a break statement, the switch statement will continue executing the following expressions ("falling-through") until a break, or the end of the switch statement is reached.

### Parameters:

var: the variable whose value to compare to the various cases

label: a value to compare the variable to

### switch (var) {   case label:     *// statements*     break;   case label:     *// statements*     break;   default:      *// statements*   break; }

### Example

switch (var) {  
    case 1:  
      *//do something when var equals 1*  
      break;  
    case 2:  
      *//do something when var equals 2*  
      break;  
    default:   
      *// if nothing else matches, do the default*  
      *// default is optional*  
    break;  
  }

#### Note: in order to declare variables within a case brackets are needed. An example is showed below.

switch (var) {  
    case 1:  
      {  
      *//do something when var equals 1*  
      int a = 0;  
      .......  
      }  
      break;  
    default:   
      *// if nothing else matches, do the default- this is optional*  
    break;  
  }

**9. Arrays**

There are many times when we need to access values that are not in order. An example would be to turn on 5 LED’s in a random or odd order instead of from left to right. The best way to achieve this is to use an Array. An array is a collection of similar type data that allows access through the use of a common name. Arrays in the C programming language, on which Arduino is based, can be complicated, but using simple arrays is relatively straightforward.

Here is an example of how a simple array might look in a program:

int myPins[5] = {2, 4, 8, 3, 6};

In this Array the variable “myPins” can have up to 5 different values (or elements), 2,4,8,3, or 6.

#### Before using an array it must first be declared. You first create a standard variable then add [ ] after it. The contents of the Array are then listed between { }.

All of the methods below are valid ways to create (declare) an array.

int myInts[6];

int myPins[] = {2, 4, 8, 3, 6};

int mySensVals[6] = {2, 4, -8, 3, 2, 14}; // This is the preferred method. The [6] indicates how many elements in the array

char message[6] = "hello"; // used for letters and words instead of numbers

int lookup[8] = {B1000, B1100, B0100, B0110, B0010, B0011, B0001, B1001}

-You can declare an array without initializing it as in myInts.  
-In myPins we declare an array without explicitly choosing a size. The compiler counts the elements and creates an array of the appropriate size.

-you can both initialize and size your array, as in mySensVals

- Note that when declaring an array of type char, one more elements than your initialization is required, to hold a required null character.

-An array can contain binary numbers only. it is declared as a "byte" array and each element has a "B" marker

#### Using and Accessing an Array

To use the elements of an Array you simply use the variable with the # of the Element you want to use contained in [ ]. For example if you wanted to give the variable ”myPins” (from the original “ int myPins[5] = {2, 4, 8, 3, 6};” array) a value of the 3rd character (which is 8) we would use the line:

myPins[2]

In this line we give “myPins” the value of the #2 element which is actually the 3rd number. The numbering might seem a bit odd because Arrays are **zero indexed**, that is the first element of the array is at position 0 not 1. Referring to the array initialization above:

myPins[0] == 2, myPins[1] == 4, myPins[2] == 8, myPins[3] == 3, myPins[4] == 6,

As an example, in an Array of 10 elements, index nine is the last element. Hence:

int myArray[10]={9,3,2,4,3,2,7,8,9,11};

// myArray[9] contains the number 11

// myArray[10] is invalid and will give you random information

For this reason you should be careful in accessing arrays. Accessing past the end of an array (using an index number greater than your declared array size - 1) is reading from memory that is in use for other purposes. Reading from these locations is probably not going to do much except yield invalid data. Writing to random memory locations is definitely a bad idea and can often lead to unhappy results such as crashes or program malfunction. This can also be a difficult bug to track down.

Unlike BASIC or JAVA, the C compiler does no checking to see if array access is within legal bounds of the array size that you have declared.

#### To assign a value to an array:

mySensVals[0] = 10;

#### To retrieve a value from an array:

x = mySensVals[4];

#### Arrays and FOR Loops

Arrays are often manipulated inside **for** loops, where the loop counter is used as the index for each array element. For example, to print the elements of an array over the serial port, you could do something like this:

int myPins[] = {2, 4, 8, 3, 6}; // declare this array in void setup ()

int i;

for (i = 0; i < 5; i = i + 1) {

Serial.println(myPins[i]);

}

**2D Arrays.**

A 2 dimensional array uses 2 or more single arrays in a group. A 2D array

Basic 7seg array code:

int num\_array[10][8] = { { 1,1,1,1,1,1,1,1 }, // 0 // makes an array of 7 Seg pin config of numbers

{ 1,0,0,0,0,0,0,0 }, // 1

{ 0,1,0,0,0,0,0,0 }, // 2

{ 0,0,1,0,0,0,0,0 }, // 3

{ 0,0,0,1,0,0,0,0 }, // 4

{ 0,0,0,0,1,0,0,0 }, // 5

{ 0,0,0,0,0,1,0,0 }, // 6

{ 0,0,0,0,0,0,1,0 }, // 7

{ 1,0,1,0,1,0,0,1 }, // 8

{ 0,0,0,0,0,0,0,0 }}; // 9

void Num\_Write(int); //function header

void setup() {

pinMode(4, OUTPUT);

pinMode(5, OUTPUT);

pinMode(6, OUTPUT);

pinMode(7, OUTPUT);

pinMode(8, OUTPUT);

pinMode(9, OUTPUT);

pinMode(10,OUTPUT);

pinMode(11,OUTPUT);

Serial.begin(9600);//extra

}

void loop() {

for (int counter = 10; counter > 0; --counter) // counts down from 9 to 0

{

delay(3000);

Num\_Write(counter-1);

}

delay(5000);

this calls the num\_write function and gives it a value of 9 to 0. This will be used in the function later to determine which rows of o's and 1's will be used to turn on the individual leds.

}

void Num\_Write(int number) // 2. this functions writes values to the sev seg pins. the number determined by counter-1 in function call above is now the value of variable "number"

{

int pin= 4; // 4 is # of first pin an led is connected to on arduino

for (int j=0; j < 8; j++) {

digitalWrite(pin, num\_array[number][j]);

The previous 3 lines determine which of the 8 output pins gets a HIGH or Low to draw the correct number. The number to be drawn is determined assigns a HIGH (1) or LOW(0) by looking at each row in num\_array [ number](will be = 0to9) and each row's individual bit value is determined by reading num\_array [ J] will be (0to8). For example first number written to led display will be 9 (counter-1=9) so number = 9 and pin=4, j=1 so first bit from row 9 will be a zero so dig pin 4 is low (led off) .

pin++; //adds 1 to pin # for each array value

}

}

**10. The Map Function**

The map function scales a number from one range of values to another, and is often used when working with the analogRead() function. The arduino has a built in analog to digital converter that gives a value between 0 and 1023. This is built into the arduino's circuitry and cannot be changed. Unfortunately we usually need to use a different range of numbers in our program . The number 20 in the range 0–100 can be expressed as a percentage: 20%. When re-mapping that number to the range 0–1000, the number is scaled to be 20% of the new range, which is 200.

## Map and AnalogRead

The Arduino’s analog system can read 1024 levels between 0V and 5V, and so the value returned by the analogRead function is an integer in the range 0 through 1023. For many uses, this is fine. However, using map you can scale the value to represent any range. For example if we can express the value in volts in the range 0–5, which might be more helpful when specifically measuring voltages.

the map function takes the form, **map(input\_variable, start\_value\_of\_source, end\_value\_of\_source, start\_value\_of\_new\_range,end\_value\_of\_new\_range)**

For example here is how you would convert a potentiometer's input of 0-1023 to a value between 0 and 5.

int a = map(a, 0, 1023, 0, 5);

Now the analog input a would read between 0 and 5 when the potentiometer is turned through its full travel.

## 11.The Constrain function

## The constrain() function limits variable data to a values within a specified range between low end of range and high end of range. For example you could limit the values of a typical potentiometer from its normal 0 to 1023 range to a range of 10 to 200. It works on all data types and takes the form, ****constrain(input\_variable, low\_end\_of\_range, high\_end\_of\_range)****

output = constrain(input, low\_end\_of\_range, high\_end\_of\_range);

code fragment example:

 // constrain example (code fragment)  
// constrain a variable to between 10 and 200  
  
input = analogRead(AO);  // read sensor  
output = constrain(input, 10, 200);      // limit values to between 10 and 200  
output = map(output, 10, 200, 10, 2000)  // map to a range 10x larger

Constrain is often used with the map() function and may be used either before or after the map function to limit sensor data in particular, to a known range. Note that if you using constrain() before or after map() will require using a different parameters in the constrain command, since you are mapping numbers to a new range. There is no real advantage to constraining values before, or after, mapping them.

/\* Uses an ldr voltage divider circuit to control brightness of an led. The map function changes

any ldr analog in values that are between 140-1023 into a 1-255 analog write output value that

will be used for controlling the brightness of the LED (1=led off and 255= max brightness). 140 represents

the ldr value when a finger is held over the ldr.(found through experimentation). We want this to be the

LED off point so we map this to 1-255. The constrain function 0-255 prevents a negative ldrinput value

when the actual ldr divider circuit returns a value less than 140. We must do this because negative

analogWrite values give strange results ( -1 is read as 255, -2 = 254 etc.)

\*/

int ledPin = 13;// LED on pin 13 brightness proportional to Light on ldr

int ldrInput; // the analog value of the light hitting the ldr - standard analog input between 0-1023

int mapval, mapval2, noconstrain ;

void setup() {

pinMode(ledPin, OUTPUT);

Serial.begin(9600); //initialize serial monitor

}

void loop() {

ldrInput = map(analogRead(A0), 140, 1023 , 0 , 255); // scales the input 0-1023 to 0-255.

//If analog read is below 140 then ldrInput will be negative - requires constrain command:

noconstrain = ldrInput;

ldrInput = constrain(ldrInput, 0, 255);// prevents ldr value from going negative

analogWrite(ledPin, ldrInput); // set the LED pin brightness

Serial.print(" AnalogRead 0-1023: ");//actual analog in value 0-1023

Serial.print(analogRead(A0));//actual analog in value 0-1023

Serial.print(" PWM 1-255 LED: ");

Serial.print(ldrInput);// led brightness 0-255

Serial.print(" no Constrain PWM value: ");

Serial.println(noconstrain);

## 12. Delay and Timing

The delay function is used in most programs to control simple timing of outputs. You used delay in the first Arduino program you used - "blink". While it is easy to create a blinking LED with the delay() function, and many sketches use short delays for such tasks as switch debouncing, the use of delay() in a sketch has significant drawbacks. No other reading of sensors, mathematical calculations, or pin manipulation can go on during the delay function, so in effect, it brings most other activity to a halt. An alternative approach to controlling timing is the [millis()](https://www.arduino.cc/en/Reference/Millis) function as shown in the sketch below. You should avoid the use of delay() for timing of events longer than 10's of milliseconds delay(10); unless the Arduino sketch is very simple.

Certain things do go on while the delay() function is controlling the Atmega chip however, because the delay function does not disable interrupts. Serial communication that appears at the RX pin is recorded, PWM (analogWrite) values and pin states are maintained, and interrupts will work as they should.

Sometimes you need to do two things at once. For example you might want to blink an LED while reading a button press. In this case, you can't use delay(), because Arduino pauses your program during the delay() . If the button is pressed while Arduino is paused waiting for the delay() to pass, your program will miss the button press.

This sketch demonstrates how to blink an LED without using delay(). It turns on the LED on and then makes note of the time. Then, each time through loop(), it checks to see if the desired blink time has passed. If it has, it toggles the LED on or off and makes note of the new time. In this way the LED blinks continuously while the sketch execution never lags on a single instruction.

An analogy would be warming up a pizza in your microwave, and also waiting some important email. You put the pizza in the microwave and set it for 10 minutes. The analogy to using delay() would be to sit in front of the microwave watching the timer count down from 10 minutes until the timer reaches zero. If the important email arrives during this time you will miss it.

What you would do in real life would be to turn on the pizza, and then check your email, and then maybe do something else (that doesn't take too long!) and every so often you will come back to the microwave to see if the timer has reached zero, indicating that your pizza is done.

In this tutorial you will learn how to set up a similar timer. It will be based on the simple blink program with the built in LED attached to pin 13.

Start the Arduino Software (IDE), and enter the code below.

### Code The code below uses the [millis()](https://www.arduino.cc/en/Reference/Millis) function, a command that returns the number of milliseconds since the board started running its current sketch, to blink an LED.

*/\* Blink without Delay. Turns on and off an LED without using the delay() function.  This means that other code  
 can run at the same time without being interrupted by the LED code.*  
int ledPin =  13;         
int ledState = LOW;             *// ledState used to set the LED*

long interval = 1000;           *// interval at which to blink (milliseconds)*  
unsigned long previousMillis = 0;   *// Generally, you should use "unsigned long" for variables that hold time*  
*// The value will quickly become too large for an "int" variable to store* *will store last time LED was updated*  
  
  
 void **setup**()

 {  
  pinMode(ledPin, OUTPUT);  *// set the digital pin as output:*  
}  
  
void **loop**() {  
  *// check to see if it's time to blink the LED; that is, if the*  
  *// difference between the current time and last time you blinked*  
  *// the LED is bigger than the interval at which you want to blink the LED.*  
 unsigned long currentMillis = millis();  
  
 if (currentMillis - previousMillis >= interval) {   *// save the last time you blinked the LED*  
previousMillis = currentMillis;

 if (ledState == LOW) { *// if the LED is off turn it on and vice-versa:*  
      ledState = HIGH;  
    }

else {  
      ledState = LOW;  
    }    
    digitalWrite(ledPin, ledState); *// set the LED with the ledState of the variable:*  
  }  
}

**13.Functions**

Within a program, you often need to do a task over and over again with some small differences. Functions allow us to create “reusable” code.   For example, you might have three LEDs and you want to blink the first one twice, the second one once, and the third one four times.  We could do it the hard way and write the code out for each case, or we could come up with a clever function that we reuse for each case.  Here’s a function we could use:

void blinkLED(int ledPin, int times){  
  
for(int k = 0; k < times; k++){  
              digitalWrite(ledPin, HIGH);  
              delay(500);  
              digitalWrite(ledPin,LOW);  
              delay(500);  
  }  
}

This code snippet declaresa function by giving it a name, **blinkLED**, and providing a block of code to run when it is called.  It looks a lot like our **setup**() and **loop**() declarations (**setup**() and **loop**() are, indeed, types of functions).   Functions must be declared outside of the **setup**() and **loop**() blocks.  You can put the function declarations before or after the **setup**() and **loop**() blocks, but most programmers put them after.   
  
The values in parentheses following the function name are the function’s arguments.  We need to provide values of the appropriate type when we call this function.  In our example, the function expects two **int** values when it is called.   
  
Once we have a function declared, we can call it from **setup**(), **loop**(), or another function.  Here’s how we call the function to blink the LED on pin 13 four times:  
  
blinkLED(13,4);  
  
We could also use variables we declared elsewhere in our function call.  Just remember that the function will expect to receive two **int**variables:

int blueLED = 11;  
int times = 4;  
blinkLED(blueLED,times);

You might be wondering: what does **void** mean in the function declaration?  We use the label **void** when the function does not return any value that we need to store in memory; our function just blinks an LED.  But we do want to return a value in a lot of cases.  Here’s an example:

int lowestCommon(int num1, int num2){  
  int low = min(num1, num2);  
  int high = num1\*num2;  
  int LCD = high;  
  for(int k = high; k >= low; k--){  
    if(k % num1 == 0 & k % num2 == 0){  
      LCD = k;  
    }  
  }  
  return LCD;  
}

This function returns the lowest common denominator for two numbers (remember middle school algebra?).  The odd symbol **%** is the modulo operator.  It returns the remainder of the division of two integers.  We know one number is divisible by the other when the remainder is 0.  
  
 If we want to return a value from a function, the declaration needs to start with the variable type we want to return.  We return an **int**in our example, but it can be any variable type.  The **return** statement at the end of the function tells the Arno what value to return after the function is run.  
  
 We can call the function like this:  
  
int theLCD = lowestCommon(27,18);

One more important point has to do with the scopeof variables.  Variables declared within functions are local variables.  They can only be accessed by the function and they are reset each time the function is called.  In our example above, we declare the variables high**,** low**,**and LCD.  These variables can’t be accessed anywhere else in the code.  The same is true for variables declared within the **setup**() and **loop**() blocks (after all these are functions, too).  In contrast, a global variable can be accessed anywhere.  Global variables are declared at the beginning of the program, before the **setup**(), **loop**(), or other functions.  We mostly use global variables in our projects for the sake of simplicity.  An advantage of using local variables is that it frees up memory as the variables only take up space while the function is being processed.  This can be important in complex sketches that push the limits of variable memory in the microcontroller.

**14. The bitRead() command**

The stepper motor programs introduce several new concepts. One of these is the bitRead command found at the end of the first assignment's code. The purpose of a bitread command is to allow the program to easily access the individual bits (0's or 1's) in a binary number. This is especially useful for stepper motors because 0's and 1's are used with the digitalwrite command to determine when the 4 coils are energized. The first example code below shows the basic function of a bitRead command:

**Example 1**

int redLED = 13; // creates a variable named redLED and assigns it a value of 13  
byte a=101 // creates a binary variable named "a" and assigns it a value of 101. This is NOT one hundred and one, it is binary 101 and is equal to 5 in standard decimal notation

byte b=1011111

void setup() {  
  pinMode(redLED, OUTPUT); // sets arduino's pin 13 (redLED) as an output. }  
  
void loop(){

// the next 4 lines do exactly the same thing: turn on an LED on pin 13.  
 digitalWrite(13, HIGH)); // turns on arduino port 13

digitalWrite(13, 1)); // a 1 can also be used instead of HIGH

digitalWrite(redLED, 1)); // the variable redLED can be used instead of 13

digitalWrite(redLED, bitRead(a, 0)); //in the expression bitRead(a,0) the program is directed to look at the "0" or first bit of variable "a" which is a 1. (The first "0" bit is the right most bit of 101 (remember the variable a = 101).

//The 2 lines below access the 2 remaining bits of "a".

digitalWrite(redLED, bitRead(a, 1)); //will read the "1" or middle bit of variable "a" which is 0

digitalWrite(redLED, bitRead(a, 2)); //will read the "2" or left most bit of "a" which is 1

digitalWrite(redLED, bitRead(b, 5)); //This example reads the 6th bit from the right of variable b (remember bitread starts with the first bit on the far right being 0)

The "bitRead()" command in the above code is capable of accessing any specific single bit of a binary number. A bitRead is written in the form of: bitRead(x, n) where x represents the binary variable to be read and "n" is the position of the specific bit to be read in that binary number. n values start at 0 which represents the Least Significant Bit (the bit that is the farthest right in the number) and increase as you read bits going left. As an example in the code: digitalWrite(redLED, bitRead(a, 0)); In this line we replace the digitalWrite HIGH or LOW with " bitRead (a, 0) ". this expression will return a value of 1 which is the right most bit in 101 (remember that "a" was given a value of 101 in the variable declaration at the beginning of the program). This 1 is seen by the digitalWrite command as being the same as HIGH, so the red LED will be turned on. By changing the 0 in the bitRead command to a 1 we access the middle bit of 101 which is a 0. This would have the result of digitalWrite being given a LOW command which would turn the LED off.

Stepper motor code:

**Example 2**

int motorPin1 = 8; // Blue - 28BYJ48 pin 1 - next 4 lines declare variables for the motor pins

int motorPin2 = 9; // Pink - 28BYJ48 pin 2

int motorPin3 = 10; // Yellow - 28BYJ48 pin 3

int motorPin4 = 11; // Orange - 28BYJ48 pin 4

1st element 3rd element last element

| | |

V V V

int lookup[8] = {B1000, B1100, B0100, B0110, B0010, B0011, B0001, B1001}; // array to control which coils get energized for each step. The 1 in each of the 8 elements represents coil on and 0 represents coil off

void setup() {

//declare the motor pins as outputs

pinMode(motorPin1, OUTPUT);

pinMode(motorPin2, OUTPUT);

pinMode(motorPin3, OUTPUT);

pinMode(motorPin4, OUTPUT);

}

void loop() {

//for (int i = 7; i >= 0; i--) // counting loop that cycles through the 8 elements of lookup Array. To reverse motor have i count down instead of up.

for(int i = 0; i <= 7; i++) // counter loop to give variable i a value between 0 and 7 this will be used to access the lookup array's 0 to 7 elements

{

digitalWrite(motorPin1, bitRead(lookup[i], 0)); //bitRead looks at the lookup array's 1st bit (furthest right digit) from the "out" Element. (out=i)

digitalWrite(motorPin2, bitRead(lookup[i], 1)); //bitRead looks at the lookup array's 2nd bit (second from right digit)from the "out" Element. (out=i)

digitalWrite(motorPin3, bitRead(lookup[i], 2)); //bitRead looks at the lookup array's 3rd bit (third from right digit)from the "out" Element. (out=i)

digitalWrite(motorPin4, bitRead(lookup[i], 3)); //bitRead looks at the lookup array's 4th bit (fourth from right digit)from the "out" Element. (out=i)

delayMicroseconds(1000); // this controls the speed of the motor,

}

}

In the last 4 lines in the above code bitRead is again used to give a value of HIGH or LOW to the digitalWrite commands for the 4 coils. bitRead is capable of reading each of the 4 bits contained in the 8 elements in the "lookup" array. The counting loop "for(int i = 0; i <= 7; i++)" gives a value of 0 to 7 to "i" and this determines which array element each of the 4 lines looks at. These 0's or 1's are the binary data in each of the elements arrays:

int lookup[8] = {B1000, B1100, B0100, B0110, B0010, B0011, B0001, B1001};

For example the 0 element (first ) in the above array is B1000. A bitread (lookup[0], 0) will return the right most bit which is a 0 and bitread (lookup[0], 3) would return the left most bit which is a 1.