10.1 Introduction to Arduino Microcontroller:

Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino boards are able to read inputs: light on a sensor, a finger on a button, or a Twitter message, and turn it into an output: activating a motor, turning on an LED, publishing something online. You can tell your board what to do by sending a set of instructions to the microcontroller on the board. To do so you use the Arduino programming language (based on Wiring), and the Arduino Software (IDE), based on Processing.

Arduino was born at the Ivrea Interaction Design Institute as an easy tool for fast prototyping, aimed at students without a background in electronics and programming. As soon as it reached a wider community, the Arduino board started changing to adapt to new needs and challenges, differentiating its offer from simple 8-bit boards to products for IoT applications, wearable, 3D printing, and embedded environments.

10.2 The Importance of Arduino:

Arduino has been used in thousands of different projects and applications. The Arduino software is easy-to-use for beginners, yet flexible enough for advanced users. It runs on Mac, Windows, and Linux. Teachers and students use it to build low-cost scientific instruments, to prove chemistry and physics principles, or to get started with programming and robotics. Designers and architects build interactive prototypes, musicians and artists use it for installations and to experiment with new musical instruments. Makers, of course, use it to build many of the projects exhibited at the Maker Faire, for example. Arduino is a key tool to learn new things. Anyone - children, hobbyists, artists, programmers - can start tinkering just following the step-by-step instructions of a kit, or sharing ideas online with other members of the Arduino community.

Arduino offers some advantage for teachers, students, and interested amateurs over other systems:

- ***Inexpensive:*** Arduino boards are relatively inexpensive compared to other microcontroller platforms.
- ***Cross-platform:*** The Arduino Software (IDE) runs on Windows, Macintosh OSX, and Linux operating systems. Most microcontroller systems are limited to Windows.

- ***Simple, clear programming environment:*** The Arduino Software (IDE) is easy-to-use for beginners, yet flexible enough for advanced users to take advantage of as well. For teachers, it's conveniently based on the Processing programming environment, so students learning to program in that environment will be familiar with how the Arduino IDE works.
- ***Open source and extensible software:*** The Arduino software is published as open-source tools, available for extension by experienced programmers. The language can be expanded through C++ libraries, and people wanting to understand the technical details can make the leap from Arduino to the AVR C programming language on which it's based. Similarly, you can add AVR-C code directly into your Arduino programs if you want to.
- ***Open source and extensible hardware:*** The plans of the Arduino boards are published under a Creative Commons license, so experienced circuit designers can make their own version of the module, extending it and improving it. Even relatively inexperienced users can build the breadboard version of the module in order to understand how it works and save money.

10.3 Types of Arduino Boards:

There are 20 different Arduino types, each offering unique features and capabilities. From basic microcontrollers to more advanced modules, let’s explore some of the popular types of Arduino boards.

1. Arduino Uno R3:
2. Arduino Nano
3. Arduino Micro
4. Arduino Leonardo
5. Arduino Micro
6. Arduino Mega2560 Rev3
7. Arduino Nano 33 BLE
8. Arduino Due
9. LilyPad Arduino Board
10. Arduino Bluetooth

10.4 The Features of Arduino Boards:

The following are some of the key features of Arduino boards.

- **Microcontroller:** It acts as Arduino's "brain" by handling all processing tasks and providing access to input/output pins (I/O).
- **Power Supply Source:** An external power source, such as a battery or USB port is needed. Some models also offer alternative methods, like solar panels or AC adapters that allow more flexibility when powering up.
- **Digital & Analog I/O Pins:** General-purpose digital inputs and outputs read signals from sensors or buttons while analogs enable complex elements like distance sensors or motor controllers to connect easily.
- **USB Interface (e.g., FTDI):** The serial communication protocol used by most Arduinos is UART over a mini-USB port. It enables connection with computers for simple data transfer tasks and programming.
- **Clock Speed & Memory Capacity:** Higher clock speeds result in faster performance while larger memory capacities enable more complex projects. These are important considerations when selecting an appropriate Arduino model.

The Arduino Uno R3 is a popular board among DIY electronics that offers features, such as 14 digital input/output pins, 6 analog pins, and an ICSP (In-Circuit Serial Programming) header.

- It runs on the ATmega328P 16MHz microchip providing up to 5V voltage supply to attached components.
- While it uses a USB-B connector for the computer interface, this doesn't mean that projects requiring advanced skills cannot be built with it.
- The key specs include 2kB SRAM memory capability, 32kB flash storage space, and 1KB EEPROM chip along with UART, and I2C SPI communication capabilities which can also be replaced in case of any problem.
- It is a great choice for anyone wanting to get into DIY electronics and programming. It can be used for projects both simple and complex.

10.5 Program Structure:

The basic structure of the Arduino programming language is fairly simple and runs in at least two parts. These two required parts, or functions, enclose blocks of statements.

```
void setup( )
{
   statements;
}
void loop( )
{
   statements;
}
```

Where setup() is the preparation, loop() is the execution. Both functions are required for the program to work. The setup function should follow the declaration of any variables at the very beginning of the program. It is the first function to run in the program, is run only once, and is used to set pinMode or initialize serial communication. The loop function follows next and includes the code to be executed continuously-reading inputs, triggering outputs, etc. This function is the core of all Arduino programs and does the bulk of the work.

1. Setup():

The setup() function is called once when your program starts. Use it to initialize pin modes, or begin serial. It must be included in a program even if there are no statements to run.

```
void setup( )
{
   pinMode(pin, OUTPUT);          // sets the ‘pin’ as output
}
```

2. Loop():

After calling the setup() function, the loop() function does precisely what its name suggests, and loops consecutively, allowing the program to change, respond, and control the Arduino board.

```
void loop( )
{
   digitalWrite(pin, HIGH);       // turns ‘pin’ on
   delay(1000);                   // pause for one second (1000ms=1s)
   digitalWrite(pin, LOW);        // turns ‘pin’ off
   delay(1000);                   // pause for one second
}
```

3. Curly Braces{}:

Curly braces (also referred to as just "braces" or "curly brackets") define the beginning and end of function blocks and statement blocks such as the void loop() function and the for and if statements.

```
type function( )
{
   statements;
}
```

An opening curly brace { must always be followed by a closing curly brace}. This is often referred to as the braces being balanced. Unbalanced braces can often lead to cryptic, impenetrable compiler errors that can sometimes be hard to track down in a large program. The Arduino environment includes a convenient feature to check the balance of curly braces. Just select a brace, or even click the insertion point immediately following a brace, and its logical companion will be highlighted.

4. Semicolon;:

A semicolon must be used to end a statement and separate elements of the program. A semicolon is also used to separate elements in a for loop.

```
int x=13;              // declares variable ‘x’ as the integer 13
```

Note: Forgetting to end a line in a semicolon will result in a compiler error. The error text may be obvious, and refer to a missing semicolon, or it may not. If an impenetrable or seemingly illogical compiler error comes up, one of the first things to check is a missing semicolon, near the line where the compiler complained.

5. Block Comments /*…*/:

Block comments, or multi-line comments, are areas of text ignored by the program and are used for large text descriptions of code or comments that help others understand parts of the program. They begin with /* and end with */ and can span multiple lines.

```
/*   this is an enclosed block   comment don’t forget
     the closing  comment they  have to  be  balanced!
*/
```

Because comments are ignored by the program and take no memory space they should be used generously and can also be used to "comment out" blocks of code for debugging purposes.

Note: While it is possible to enclose single line comments within a block comment, enclosing a second block comment is not allowed.

6. Line Comments //:

Single line comments begin with II and end with the next line of code. Like block comments, they are ignored by the program and take no memory space.

```
//  this is a single line comment
```

Single line comments are often used after a valid statement to provide more information about what the statement accomplishes or to provide a future reminder.

7. pinMode(pin, mode):

Used in void setup () to configure a specified pin to behave either as an INPUT or an OUTPUT

```
pinMode(pin,  OUTPUT);        // sets 'pin'  to output
```

Arduino digital pins default to inputs, so they don't need to be explicitly declared as inputs with pinMode(). Pins configured as INPUT are said to be in a high impedance state.

There are also convenient 20KΩ pullup resistors built into the Atmega chip that can be accessed from software. These built-in pull up resistors are accessed in the following manner:

```
pinMode(pin,  INPUT);              // set 'pin'  to input
digitalWrite(pin, HIGH);           // turn on  pullup  resistors
```

Pull up resistors would normally be used for connecting inputs like switches. Notice in the above example it does not convert pin to an output, it is merely a method for activating the internal pull-ups.

Pins configured as OUTPUT are said to be in a low-impedance state and can provide 40mA (milliamps) of current to other devices/circuits. This is enough current to brightly light up an LED (don't forget the series resistor), but not enough current to run most relays, solenoids, or motors.

Short circuits on Arduino pins and excessive current can damage or destroy the output pin, or damage the entire Atmega chip. It is often a good idea to connect an OUTPUT pin to an external device in series with a 4700 or 1KΩ resistor.

8. digitalRead(pin):

Reads the value from a specified digital pin with the result either HIGH or LOW. The pin can be specified as either a variable or constant (0-13).

```
                    value= digitalRead(Pin);    // sets 'value'  equal to the input  pin
int inPin = 7;    // pushbutton connected to digital pin 7
int val = 0;      // variable to store the read value
void setup() {
  pinMode(inPin, INPUT);    // sets the digital pin 7 as input
}
void loop() {
  val = digitalRead(inPin);   // read the input pin
}
```

9. digitalWrite(pin, value):

Ouputs either logic level HIGH or LOW at (turns on or off) a specified digital pin. The pin can be specified as either a variable or constant (0-13).

```
digitalWrite(pin, HIGH);      // sets 'pin'   to high
```

The following example reads a pushbutton connected to a digital input and turns on an LED connected to a digital output when the button has been pressed:

```
int led = 13;              // connect  LED to  pin 13
int pin = 7;               // connect pushbutton to  pin  7
int value = o;             // variable  to  store the read value
void setup()
{
     pinMode(led,  OUTPUT);          // sets pin 13  as   output
     pinMode(pin,  INPUT);           // sets pin 7  as   input
}
void loop ()
{
     value= digitalRead(pin);   //sets 'value'  equal to the input pin
     digitalWrite(led,  value);  //sets 'led'  to the button's value
}
```

10. analogRead(pin):

Reads the value from a specified analog pin with a 1O-bit resolution. This function only works on the analog in pins (0-5). The resulting integer values range from 0 to 1023.

```
                    value=  analogRead(pin);          //sets 'value'  equal to  'pin'
```

Note: Analog pins unlike digital ones, do not need to be first declared as INPUT nor OUTPUT.

```
int analogPin = A3; // potentiometer wiper (middle terminal) connected to analog pin 3
                  // outside leads to ground and +5V
int val = 0;  // variable to store the value read
void setup() {
  Serial.begin(9600);          //  setup serial
}
void loop() {
  val = analogRead(analogPin);  // read the input pin
  Serial.println(val);         // debug value
}
```

11. analogWrite(pin, value):

Writes a pseudo-analog value using hardware enabled pulse width modulation (PWM) to an output pin marked PWM. On newer Arduinos with the ATmega168 chip, this function works on pins 3, 5, 6, 9, 10, and 11. Older Arduinos with an ATmega8 only support pins 9, 10, and 11. The value can be specified as a variable or constant with a value from 0-255.

```
analogWrite(pin, value);    //writes 'value'  to analog 'pin'
```

A value of 0 generates a steady 0 volts output at the specified pin; a value of 255 generates a steady 5 volts output at the specified pin. For values in between 0 and 255, the pin rapidly alternates between 0 and 5 volts -the higher the value, the more often the pin is HIGH (5 volts). For example, a value of 64 will be 0 volts three-quarters of the time, and 5 volts one quarter of the time; a value of 128 will be at 0 half the time and 255 half the time; and a value of 192 will be 0 volts one quarter of the time and 5 volts three-quarters of the time.

Because this is a hardware function, the pin will generate a steady wave after a call to analogWrite in the background until the next call to analogWrite (or a call to digitaiRead or digitaiWrite on the same pin).

Note: Analog pins unlike digital ones, do not need to be first declared as INPUT nor OUTPUT.

The following example reads an analog value from an analog input pin, converts the value by dividing by 4, and outputs a PWM signal on a PWM pin:

```
int led =  10;                  // LED  with 220 resistor on   pin 10
int pin = 0;                    // potentiometer on  analog pin  0
int value;                      // value for  reading
void setup(){}                  // no  setup  needed
void loop ()
{
    value=   analogRead(pin);        // sets  'value'   equal to 'pin'
    value /=  4;                     // converts 0-1023 to 0-255
    analogWrite(led,   value);       // outputs  PWM   signal to led
}
```

12. delay(ms):

Pauses your program for the amount of time as specified in milliseconds, where 1000 equals 1 second.

```
delay ( 1000);                  // waits for one second
```

13. millis():

Returns the number of milliseconds since the Arduino board began running the current program as an unsigned long value.

```
value= millis();                // sets  'value'  equal to millis()
```

Note: This number will overflow (reset back to zero), after approximately 9 hours.

10.6 Functions:

A function is a block of code that has a name and a block of statements that are executed when the function is called. Custom functions can be written to perform repetitive tasks and reduce clutter in a program. Functions are declared by first declaring the function type. This is the type of value to be returned by the function such as 'int' for an integer type function. If no value is to be returned the function type would be void. After type, declare the name given to the function and in parenthesis any parameters being passed to the function.

```
type functionName(parameters)
{
   statements;
}
```

The following integer type function delayVal() is used to set a delay value in a program by reading the value of a potentiometer. It first declares a local variable v, sets v to the value of the potentiometer which gives a number between 0-1023, then divides that value by 4 for a final value between 0-255, and finally returns that value back to the main program.

Example: Delay function program.

```
int delayVal( )
{
   int v;                          // creat temporary variable ‘v’
   v=analogRead(pot);              // read potentiometer value
   v/=4;                           // converts 0-1023 to 0-255
   return v;                       // return final value
}
```

Example: Summing function program.

```
int sum_func(int x,int y) )
{
   int z=0;                        // initialize the value of z
   z=x+y;                          // read potentiometer value
   return z;                       // return final value
}
```

Note: In the main program the calling function are:

```
void loop( )
{
   result=sum_func(5,6);
}
```

10.7 Variables:

A variable is a way of naming and storing a numerical value for later use by the program. As their namesake suggests, variables are numbers that can be continually changed as opposed to constants whose value never changes. A variable needs to be declared and optionally assigned to the value needing to be stored. The following code declares a variable called inputVariable and then assigns it the value obtained on analog input pin 2:

```
int inputVariable = 0;                // declares a variable and assigns value of
0
inputVariable = analogRead(2);  //  set variable to value of analog  pin  2
```

'inputVariable' is the variable itself. The first line declares that it will contain an int, short for integer. The second line sets the variable to the value at analog pin 2. This makes the value of pin 2 accessible elsewhere in the code. Once a variable has been assigned, or re-assigned, you can test its value to see if it meets certain conditions, or you can use its value directly. As an example to illustrate three useful operations with variables, the following code tests whether the inputVariable is less than 100, if true it assigns the value 100 to inputVariable, and then sets a delay based on inputVariable which is now a minimum of 100:

```
if (inputVariable <  100)       // tests variable if less  than 100
{
inputVa riable = 100;           // if true  assigns value of  100
}
delay(inputVariable);           // uses variable  as   delay
```

Note: Variables should be given descriptive names, to make the code more readable. Variable names like tiltSensor or pushButton help the programmer and anyone else reading the code to understand what the variable represents. Variable names like var or value, on the other hand, do little to make the code readable and are only used here as examples. A variable can be named any word that is not already one of the keywords in the Arduino language.

1. Variable declaration:

All variables have to be declared before they can be used. Declaring a variable means defining its value type, as in int, long, float, etc., setting a specified name, and optionally assigning an initial value. This only needs to be done once in a program but the value can be changed at any time using arithmetic and various assignments. The following example declares that inputVariable is an int, or integer type, and that its initial value equals zero. This is called a simple assignment

```
int inputVariable = 0;
```

A variable can be declared in a number of locations throughout the program and where this definition takes place determines what parts of the program can use the variable.

2. Variable scope:

A variable can be declared at the beginning of the program before void setup(), locally inside of functions, and sometimes within a statement block such as for loops. Where the variable is declared determines the variable scope, or the ability of certain parts of a program to make use of the variable.

A global variable is one that can be seen and used by every function and statement in a program. This variable is declared at the beginning of the program, before the setup() function.

A local variable is one that is defined inside a function or as part of a for loop. It is only visible and can only be used inside the function in which it was declared. It is therefore possible to have two or more variables of the same name in different parts of the same program that contain different values.

Ensuring that only one function has access to its variables simplifies the program and reduces the potential for programming errors.

The following example shows how to declare a few different types of variables and demonstrates each variable's visibility:

```
int value;                  //  'value' is visible to any   function
void setup()
{
      // no  setup  needed
}
void loop ()
{
for  (int  i=0; i<20;)          //  ‘i’  is only visible
{                                // inside the for-loop
i++;
}
float f;                        // ‘f’ is only visible
}                                // inside loop
```

a. Byte:

Byte stores an 8-bit numerical value without decimal points. They have a range of 0-255.

```
byte  someVariable = 180;     // declares  'someVariable' as   a  byte type
```

b. Int:

Integers are the primary datatype for storage of numbers without decimal points and store a 16-bit value with a range of 32,767 to -32,768.

```
int someVariable = 1500;   // declares  'someVariable' as   an  integer  type
```

Note: Integer variables will roll over if forced past their maximum or minimum values by an assignment or comparison. For example, if x=32767 and a subsequent statement adds 1 to x, x=x+1 or x++, x will then rollover and equal -32,768.

c. Long:

Extended size datatype for long integers, without decimal points, stored in a 32-bit value with a range of 2,147,483,647 to -2,147,483,648.

```
long  someVariable = 90000;  // declares  'someVariable' as   a  long type
```

d. Float:

A datatype for floating-point numbers, or numbers that have a decimal point. Floating-point numbers have greater resolution than integers and are stored as a 32-bit value with a range of 3.4028235E+38 to -3.4028235E+38.

```
float someVariable = 3.14;  // declares  'someVariable' as   a  floating-point  type
```

Note: Floating-point numbers are not exact, and may yield strange results when compared. Floating point math is also much slower than integer math in performing calculations, so should be avoided if possible.

10.8 Arrays:

An array is a collection of values that are accessed with an index number. Any value in the array may be called upon by calling the name of the array and the index number of the value. Arrays are zero indexed, with the first value in the array beginning at index number 0. An array needs to be declared and optionally assigned values before they can be used.

```
int myArray[ ] =  {value0,  value1, value2 ... }
```

Likewise it is possible to declare an array by declaring the array type and size and later assign values to an index position:

```
int  myArray[5];       // declares integer  array wl 6  positions
myArray[3] = 10;       // assigns the 4th  index  the value 10
```

To retrieve a value from an array, assign a variable to the array and index position:

```
x = myArray[3];       // x   now  equals  10
```

Arrays are often used in for loops, where the increment counter is also used as the index position for each array value. The following example uses an array to flicker an LED. Using a for loop, the counter begins at 0, writes the value contained at index position 0 in the array flicker[], in this case 180, to the PWM pin 10, pauses for 200ms, then moves to the next index position.

```
int  ledPin =  10;       // LED on  pin  10
byte flicker [ ] = {180,  30,    255,  200,  10,  90,    150, 60};         // above array   of  8
void setup()             // different  values
{
    pinMode(ledPin,  OUTPUT);                     // sets  OUTPUT pin
}
void loop ()
{
    for(int  i=0; i<7; i++)                 // loop  equals number of values in array
    {
      analogWrite(ledPin,  flicker[i]);        //write  index   value
      delay(200);                              // pause  200ms
    }
}
```

10.9 Arithmetic:

Arithmetic operators include addition, subtraction, multiplication, and division. They return the sum, difference, product, or quotient (respectively) of two operands.

```
y = y + 3;
X = X – 7;
i = j * 6;
```

```
r = r / 5;
```

The operation is conducted using the data type of the operands, so, for example, 9/4 results in 2 instead of 2.25 since 9 and 4 are int’s and are incapable of using decimal points. This also means that the operation can overflow if the result is larger than what can be stored in the data type.

If the operands are of different types, the larger type is used for the calculation. For example, if one of the numbers (operands) are of the type float and the other of type integer, floating point math will be used for the calculation.

Choose variable sizes that are large enough to hold the largest results from your calculations. Know at what point your variable will rollover and also what happens in the other direction e.g. (0-1) OR (0--32768). For math that requires fractions, use float variables, but be aware of their drawbacks: large size and slow computation speeds.

Note: Use the cast operator e.g. (int)myFloat to convert one variable type to another on the fly. For example, i = (int) 3. 6 will set i equal to 3.

10.10 Compound Assignments:

Compound assignments combine an arithmetic operation with a variable assignment. These are commonly found in for loops as described later. The most common compound assignments include:

```
X   ++              // same as X = X + 1, or increments x by +1
X   - -             // same as X = X - 1, or decrements x by -1
X + = y             // same as X = X + y, or increments x by +y
X - = y             // same as X = X - y, or decrements x by -y
X * = y             // same as X = X * y, or multiplies x by y
X / = y             // same as X = X / y, or divides x by y
```

Note: For example, x*=3 would triple the old value of x and re-assign the resulting value to x.

10.11 Comparison operators:

Comparisons of one variable or constant against another are often used in if statements to test if a specified condition is true. In the examples found on the following pages, ?? is used to indicate any of the following conditions:

```
X = = y             // x is equal to y
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
```

10.12 Logical operators:

Logical operators are usually a way to compare two expressions and return a TRUE or FALSE depending on the operator. There are three logical operators, AND, OR, and NOT, that are often used in if statements:

Logical AND

```
if (x > 0 && X < 5)        // true only if both expressions  are true
```

Logical OR

```
if (x > 0 II y > 0)         // true if either expression is true
```

Logical NOT

```
if (!x>0)                     // true only if expression is false
```

10.13 Constants:

The Arduino language has a few predefined values, which are called constants. They are used to make the programs easier to read. Constants are classified in groups.

1. True/False:

These are Boolean constants that define logic levels. FALSE is easily defined as 0 (zero) while TRUE is often defined as 1, but can also be anything else except zero. So in a Boolean sense, -1, 2, and -200 are all also defined as TRUE.

```
if ( b  ==  TRUE) ;
{
    doSomething;
}
```

2. High/Low:

These constants define pin levels as HIGH or LOW and are used when reading or writing to digital pins. HIGH is defined as logic level 1, ON, or 5 volts while LOW is logic level 0, OFF, or 0 volts.

```
digitalWrite(13, HIGH);
```

3. Input/Output:

Constants used with the pinMode() function to define the mode of a digital pin as either INPUT or OUTPUT.

```
pinMode(13,  OUTPUT);
```

4. If:

if statements test whether a certain condition has been reached, such as an analog value being above a certain number, and executes any statements inside the brackets if the statement is true. If false the program skips over the statement. The format for an if test is:

```
if (someVariable ?? value)
{
     doSomething;
}
```

The above example compares someVariable to another value, which can be either a variable or constant. If the comparison, or condition in parentheses is true, the statements inside the brackets are run. If not, the program skips over them and continues on after the brackets.

Note: Beware of accidentally using'=', as in if (x=l0), while technically valid, defines the variable x to the value of 10 and is as a result always true. Instead use'==', as in if (x==l0), which only tests whether x happens to equal the value 10 or not. Think of'=' as "equals" opposed to '==' being "is equal to".

5. If…else:

if... else allows for 'either-or' decisions to be made. For example, if you wanted to test a digital input, and do one thing if the input went HIGH or instead do another thing if the input was LOW, you would write that this way:

```
if (inputPin == HIGH)
{
    doThingA;
}
else
{
    doThingB;
}
```

else can also precede another if test, so that multiple, mutually exclusive tests can be run at the same time. It is even possible to have an unlimited number of these else branches. Remember though, only one set of statements will be run depending on the condition tests:

```
if (inputPin < 500)
{
    doThingA;
}
else if (inputPin >= 1000)
{
    doThingB;
}
else
{
    doThingC;
}
```

Note: An if statement simply tests whether the condition inside the parenthesis is true or false. This statement can be any valid C statement as in the first example, if (inputPin == HIGH). In this example, the if statement only checks to see if indeed the specified input is at logic level high, or +5v.

6. For:

The for statement is used to repeat a block of statements enclosed in curly braces a specified number of times. An increment counter is often used to increment and terminate the loop. There are three parts, separated by semicolons (;), to the for loop header:

```
for (initialization;  condition;  expression)
{
    doSomething;
}
```

The initialization of a local variable, or increment counter, happens first and only once. Each time through the loop, the following condition is tested. If the condition remains true, the following statements and expression are executed and the condition is tested again. When the condition becomes false, the loop ends.

The following example starts the integer i at 0, tests to see if i is still less than 20 and if true, increments i by 1 and executes the enclosed statements:

```
for  (int  i=0; i<20; i++)        // declares i, tests if less
{                                 // than 20, increments i by   1
    digitalWrite(13,  HIGH);  // turns pin 13  on
    delay (250);                  // pauses for  1/4   second
    digitalWrite(13,  LOW);  // turns pin 13  off
    delay (250);                  // pauses for  1/4   second
}
```

Note: 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 expression can be any valid C statements with unrelated variables. These types of unusual for statements may provide solutions to some rare programming problems.

7. While:

while loops will loop continuously, and infinitely, until the expression 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 testing a sensor.

```
while  (someVariable ?? value)
{
    doSomething;
}
```

The following example tests whether ‘someVariable’ is less than 200 and if true executes the statements inside the brackets and will continue looping until ‘someVariable’ is no longer less than 200.

```
while (someVariable < 200)          // tests if less  than 200
{
    doSomething;                    // executes enclosed statements
    someVariable++;                 // increments variable  by 1
}
```

8. Do…While:

The do loop is a bottom driven loop that works in the same manner as the while loop, with the exception that the condition is tested at the end of the loop, so the do loop will always run at least once.

```
do
{
     doSomething;
}  while  (someVariable ?? value);
```

The following example assigns readSensors() to the variable 'x', pauses for 50 milliseconds, then loops indefinitely until 'x' is no longer less than 100:

```
do
{
    x  = readSensors();       // assigns the value of readSensors() to  x
    delay(50);                // pauses  50  milliseconds
} while  (x < 100);           // loops if x  is less  than  100
```

9. min(x, y):

Calculates the minimum of two numbers of any data type and returns the smaller number.

```
value=  min(value, 100);      // sets  'value'  to  the smaller of 'value'  or 100, ensuring that
                              // it never  gets  above 100.
```

10. max(x, y):

Calculates the maximum of two numbers of any data type and returns the larger number.

```
value=  max(value, 100);      // sets  'value'  to the larger  of 'value'  or 100, ensuring that
                              // it is at least 100.
```

11. randomSeed(seed):

Sets a value, or seed, as the starting point for the random() function.

```
randomSeed(value);            // sets 'value'  as   the  random seed
```

Because the Arduino is unable to create a truly random number, randomSeed allows you to place a variable, constant, or other function into the random function, which helps to generate more random "random" numbers. There are a variety of different seeds, or functions, that can be used in this function including millis() or even analogRead() to read electrical noise through an analog pin.

12. random(max) or random(min, max):

The random function allows you to return pseudo-random numbers within a range specified by min and max values.

```
value= random(100, 200);    // sets 'value'  to a   random number between 100-200
```

Note: Use this after using the randomSeed() function.

The following example creates a random value between 0-255 and outputs a PWM signal on a PWM pin equal to the random value:

```
int randNumber;              // variable  to store the random   value
int led =  10;               // LED  with 220   resistor on   pin  10
void setup() {}              // no  setup  needed
void loop ()
{
    randomSeed(millis());    // sets millis() as   seed
    randNumber  =  random(255);       // random   number from  0-255
    analogWrite(led,  randNumber);         // outputs  PWM   signal
    delay(500);                       // pauses for  half  a  second
}
```

13. Serial.begin(rate):

Opens serial port and sets the baud rate for serial data transmission. The typical baud rate for communicating with the computer is 9600 although other speeds are supported.

```
void setup()
{
    Serial.begin(9600);    // opens  serial  port sets data rate  to 9600   bps
}
```

Note: When using serial communication, digital pins 0 (RX) and 1 (TX) cannot be used at the same time.

Example: To start serial port and send data at a speed rate of 9600bps write the following statements in the setup() function:

```
Serial.begin(Speed)
Serial.begin(Speed,config)   // config: sets data, parity, and stop bits.
```

14. Serial.println(data):

Prints data to the serial port, followed by an automatic carriage return and line feed. This command takes the same form as Serial. print(), but is easier for reading data on the Serial Monitor.

```
Serial.println(analogValue);   // sends  the value of 'analogValue'
```

Note: For more information on the various permutations of the Serial. println() and Serial. print() functions please refer to the Arduino website.

The following simple example takes a reading from analog pinO and sends this data to the computer every 1 second.

```
void setup()
{
Serial.begin(9600);                  // sets serial  to 9600bps
}
void loop ()
{
Serial.println(analogRead(0));       // sends analog value
delay(l000);                         // pauses  for 1 second
}
```

10.14 Digital Input/Output:

Arduino pins default Configured to inputs (INPUT), so don't need to be explicitly declared as inputs with pinMode() when you're using them as inputs. Pins configured this way are said to be in a **high-impedance state**. Input pins make extremely small demands on the circuit that they are sampling, equivalent to a series resistor of 100 megohm in front of the pin. This means that it takes very little current to move the input pin from one state to another, and can make the pins useful for such tasks as implementing a capacitive touch sensor, reading an LED as a photodiode, or reading an analog sensor with a scheme such as RCTime.

This also means however, that pins configured as pinMode(pin, INPUT) with nothing connected to them, or with wires connected to them that are not connected to other circuits, will report seemingly random changes in pin state, picking up electrical noise from the environment, or capacitively coupling the state of a nearby pin.

Pullup Resistors with pins configured as INPUT: Often it is useful to steer an input pin to a known state if no input is present. This can be done by adding a pullup resistor (to +5V), or a pulldown resistor (resistor to ground) on the input. A 10K resistor is a good value for a pullup or pulldown resistor.

Prior to Arduino IDE, it was possible to configure the internal pull-ups in the following manner:

```
pinMode(pin, INPUT);          // set pin to input

digitalWrite(pin, HIGH);      // turn on pullup resistors

digitalWrite(pin, LOW);       // turn off pullup resistors
```

Pins configured as OUTPUT with pinMode() are said to be in a low-impedance state. This means that they can provide a substantial amount of current to other circuits. Arduino pins can source (provide positive current) or sink (provide negative current) up to 40 mA (milliamps) of current to other devices/circuits. This is enough current to brightly light up an LED (don't forget the series resistor), or run many sensors, for example, but not enough current to run most relays, solenoids, or motors.

Example: Turns an LED on for one second, then off for one second, repeatedly.

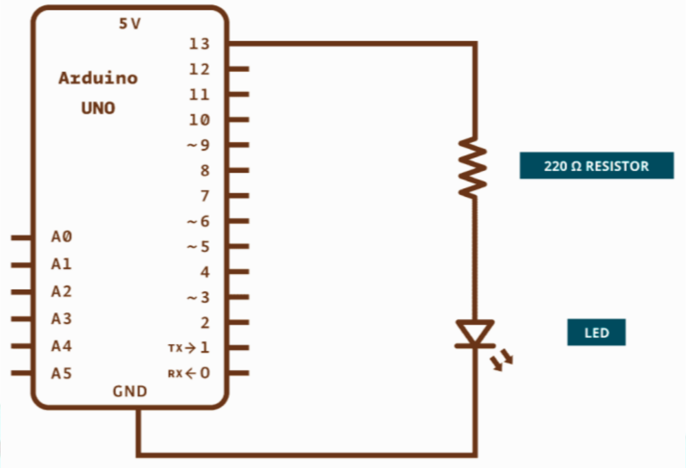

Fig. (10-1): Arduino LED Blinking.

```
const int ledPin = LED_BUILTIN;  // constants used here to set a LED pin number
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);  // initialize digital pin LED_BUILTIN as an output.
}
// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);  // turn the LED on (HIGH is the voltage level)
  delay(1000);                      // wait for a second
  digitalWrite(LED_BUILTIN, LOW);   // turn the LED off by making the voltage LOW
  delay(1000);                      // wait for a second
}
```

Example: Reads a digital input on pin 2, prints the result to the Serial Monitor.

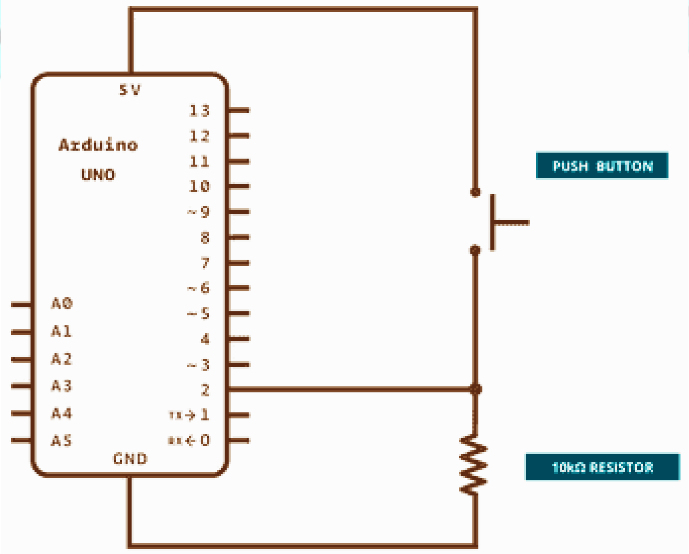

Fig. (10-2): Arduino Digital Read.

```
int pushButton = 2;
// the setup routine runs once when you press reset:
void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
  // make the pushbutton's pin an input:
  pinMode(pushButton, INPUT);
}
// the loop routine runs over and over again forever:
void loop() {
  // read the input pin:
  int buttonState = digitalRead(pushButton);
  // print out the state of the button:
  Serial.println(buttonState);
  delay(1);  // delay in between reads for stability
}
```

Seven Segment Displays on the Arduino: Seven segment displays are used in common household appliances like microwave ovens, washing machines, and air conditioners. They’re a simple and effective way to display numerical information like sensor readings, time, or quantities. In this tutorial, we’ll see how to set up and program single digit and multi-digit seven segment displays on an Arduino.

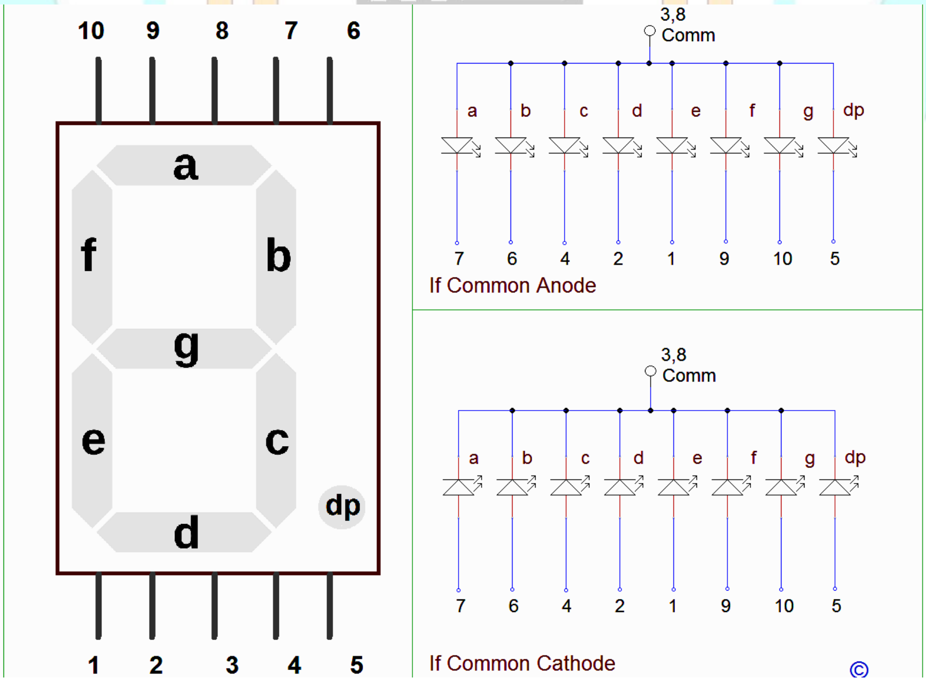

Fig. (10-3): 7-Segment Display.

Example: This simple program will count up from zero to 9 and then loop back to the start.

Segment pin	Arduino Pin
A	6
B	5
C	2
D	3
E	4
F	7
G	8
DP	9

```
#include "SevSeg.h"
SevSeg sevseg;
void setup(){
    byte numDigits = 1;
    byte digitPins[] = {};
    byte segmentPins[] = {6, 5, 2, 3, 4, 7, 8, 9};
    bool resistorsOnSegments = true;
    byte hardwareConfig = COMMON_CATHODE;
    sevseg.begin(hardwareConfig, numDigits, digitPins, segmentPins, resistorsOnSegments);
    sevseg.setBrightness(90);
}
void loop(){
    for(int i = 0; i < 10; i++){
        sevseg.setNumber(i, i%2);
        delay(1000);
        sevseg.refreshDisplay();
    }
}
```

10.6 Analog Input/Output:

- Adjust the volume of a speaker continuously by turning a knob
- Adjust the brightness of a lamp continuously when the ambient light level changes
- Adjust the speed of a motor continuously by varying how deep the accelerator is pressed

In these situations, the variables are changing continuously. Continuous signals that change with time are called **analog signals**. Arduino can process analog signal with its built-in **analog-to-digital converter (ADC)**.

However, it *cannot* produce real analog signals, as it lacks a **digital-to-analog converter (DAC)**. Of course, we can add an external DAC to the Arduino, but we usually don't need to do so. In most of the cases, a technique called **pulse-width modulation (PWM)** is enough for controlling the average power delivered by an electrical signal.

As mentioned, the Arduino has a built-in ADC. Essentially, this ADC measures the voltage at a pin, and map the measured value to an integer from 0 to 1023 linearly, i.e.

$$Output = \frac{V_{measured}}{5} \times 1023$$

In order to convert this value, use a function called map():

outputValue = map(sensorValue, 0, 1023, 0, 255);

At home, we usually adjust the brightness of a light bulb with a physical knob. However, to control the brightness of the light bulb programmatically, say, via your smartphone? This is one of the examples of Internet of Things (IoTs). It turns out that it's quite easy to control the brightness of an LED with **Pulse Width Modulation** (PWM) signal.

PWM is essentially switching on and off the power very rapidly at a particular frequency. By varying the amount of 'on' time and 'off' time in the period of each cycle, we can control the average power output. The proportion of 'on' time to the period of each cycle is known as the **duty cycle**. 25% duty cycle means that the power is on for 25% of the time in each cycle, 50% duty cycle means that the power is on for 50% of the time in each cycle.

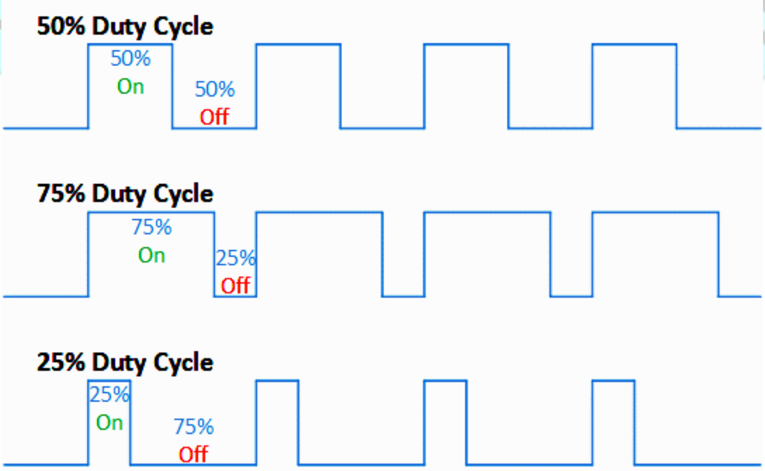

Fig. (10-3): 50%, 75% and 25% duty cycles.

Example: Reads an analog input pin, maps the result to a range from 0 to 255 and uses the result to set the pulse width modulation (PWM) of an output pin. Also prints the results to the Serial Monitor.

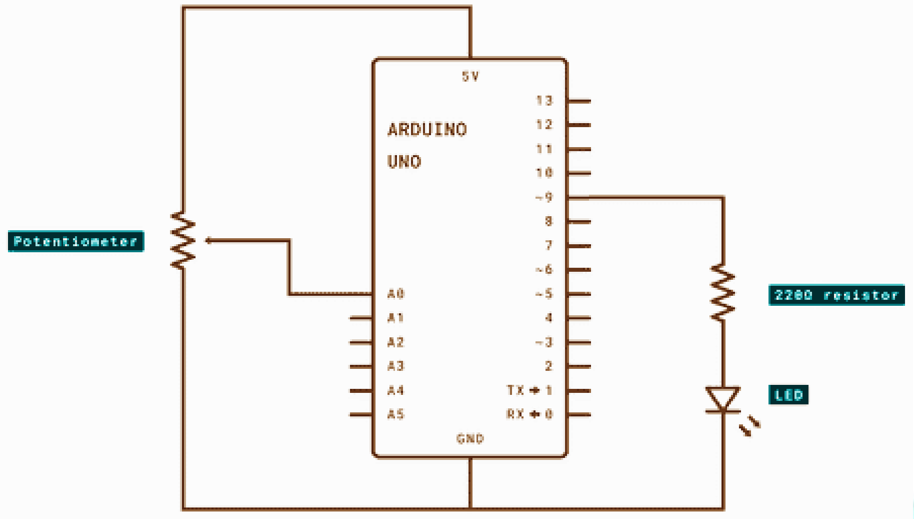

Fig. (10-4): The Analog Input/Output Arduino System.

```
const int analogInPin = A0;  // Analog input pin that the potentiometer is attached to
const int analogOutPin = 9; // Analog output pin that the LED is attached to
int sensorValue = 0;        // value read from the pot
int outputValue = 0;        // value output to the PWM (analog out)
void setup() {
Serial.begin(9600); // initialize serial communications at 9600 bps:
}
void loop() {
  sensorValue = analogRead(analogInPin);  // read the analog in value:
  outputValue = map(sensorValue, 0, 1023, 0, 255);  // map it to the range of the analog out:
  // change the analog out value:
  analogWrite(analogOutPin, outputValue);
  Serial.print("sensor = ");  // print the results to the Serial Monitor:
  Serial.print(sensorValue);
  Serial.print("\t output = ");
  Serial.println(outputValue);
  // wait 2 milliseconds before the next loop for the analog-to-digital
  // converter to settle after the last reading:
  delay(2);
}
```