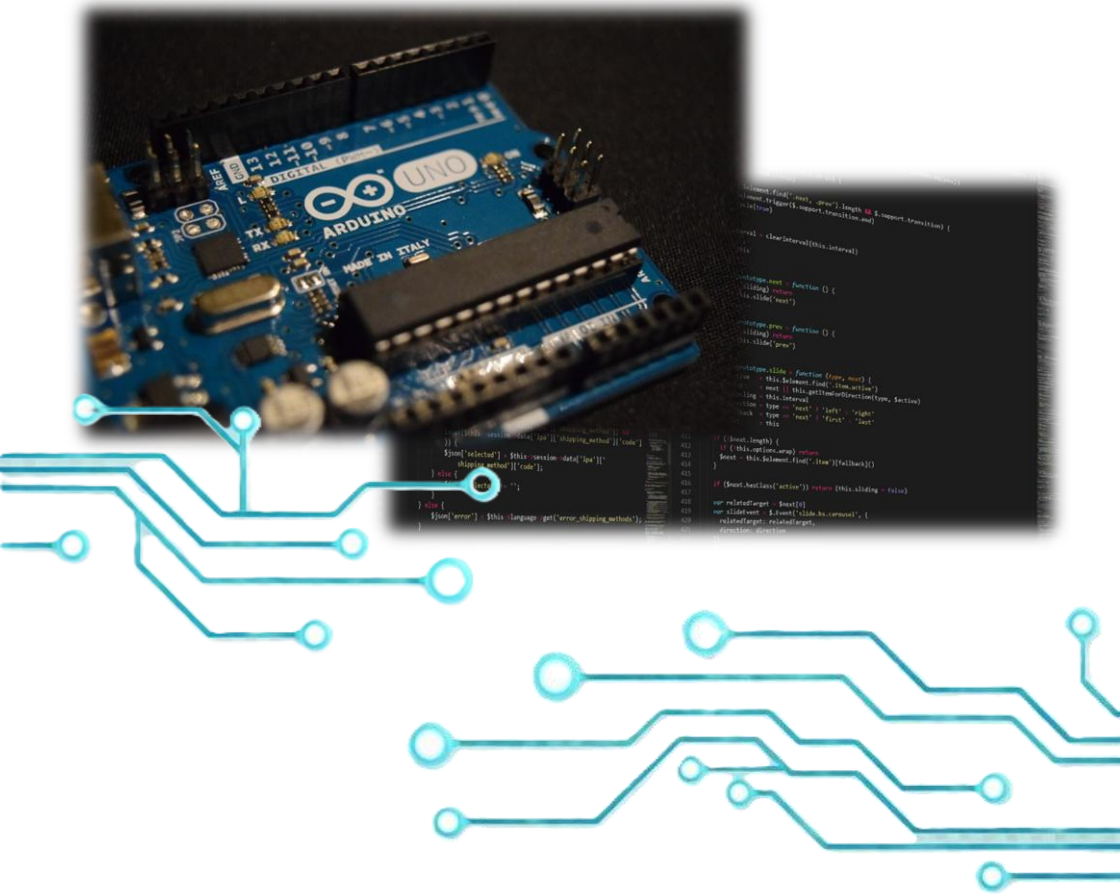
Persilience presents

Coding Motion

Arduino Basics & Physical Computing



by

Agrim Sharma

Persilience

Coding Motion- Arduino Basics & Physical Computing

Contents

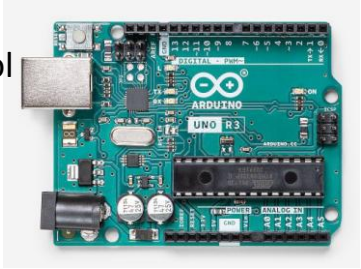
Author- Agrim Sharma
Publishing & Manager- Arnav Sharma

V1.0- February 2026

Introduction

Arduino is an open-source platform used to create interactive electronic projects. It is commonly used in devices and robots where code must continuously observe and react to the real world. "Open-source platform" means it is free to use, with publicly available designs and code libraries and circuits for everyone!

MCU refers microcontroller unit. It is a programmable integrated circuit that is used in electronics devices as a brain. Arduino Uno is a beginner-friendly microcontroller board that acts as the brain of many electronic systems. It allows sensors to provide information as inputs and enables the board to control outputs such as lights, motors, and displays. We use Arduino IDE software for coding in it.



Arduino UNO R3 board DIP version

At the core of the Arduino Uno is a programmable microcontroller chip. The surrounding circuitry on the board is designed to make this chip (ATmega328P) easier and safer to use, allowing us to focus on behavior and logic rather than hardware complexity. In this booklet, you will learn how code creates motion using the Arduino Uno.

SMD version of ATmega328P

ATmega328P in DIP package

DIP (Dual In-line Package): Chip with legs that you can see, removable and beginner-friendly.

SMD (Surface Mount Device): Chip soldered flat on printed circuit board, more compact but permanent.

Go to **arduino.cc** and download the Arduino IDE according to your PC's operating system.
Select Arduino Uno and the corresponding USB port[1] from the board and port selection menu.

The functions of the Toolbar buttons present in the top-left corner are:

1.  : The verify or checkmark button is used to compile and verify the code.
2.  :To upload the code in Arduino, we use the upload or arrow button.

[1]*The port number may vary (*Windows: COM3, COM4, etc. | Mac/Linux: /dev/ttyUSB0 or /dev/cu.usbmodem..."*). Select the port that appears when you plug in the Arduino. You can check the port of your Arduino in Device Manager.*

Values

Values play an important role in any programming language. Values or data helps in many tasks like assigning pin numbers, reading and storing information, etc. Data is of various types ie data types. Some of the data types of Arduino IDE are integers, float, double, strings(char array), boolean(true/false), etc. Different data types take up different amount of storage in the processor(Arduino), therefore, we must be careful in choosing the correct data type for particular tasks. The following data types are arranged by storage they take up in decreasing order.

char (1B) < bool (1B) < int (2B) < float (4B) = double (4B)

Integers: Integer refers to numbers without decimal parts. They are used for sensor values, counting, mode selection, PWM, angle control, time control, pin definitions, simple logic etc. To create an integer value in Arduino IDE, we use the `int` prefix before a variable during its declaration. Look at the following example.

```
const int led_pin = 2;
int counter = 0;
```

In this code snippet, the integer data type is used in many places and plays a crucial role on the logic of the hardware. The `const` (constant)keyword is used to define that the value of this integer is fixed and can't be changed during the run time of the code. For e.g., `const int led_pin = 2;` means that the integer by the name led_pin is equal to 2 and its value can never change during the runtime. However, `int counter` means that the integer by the name of counter is a variable and its value can be changed during the runtime of the code. Note that values that are constant take a little less storage than variables.

Float: This refers to the numbers with decimal parts, for e.g., 1.2, 3.142857, 6.7, etc. Float values provide more precision to readings but are slower to be executed by the program, hence, general practice is to convert the float values into integers. We use the `float` prefix to declare a value as float. For e.g., `float pi = 3.14;` declares a variable by name `pi` with decimal value of 3.14. The `const` keyword can also be used with them. Consider this example:

```
float pi = 3.14;
```

As you have notice, we declare a variable only once in a program and can assign its value using assignment(=) sign. But assignment can also be done during declaration. Unless the value is constant, its value can change after declaration. Hence, the above is same as `float pi; pi = 3.14;`

Double: The data type double can store the values of more precession than float; taking up more storage than float. E.g.,

```
double pi = 3.14159265358979323846264338327795;
```

Boolean: It only takes 1 byte in the memory. The boolean data type is declared by bool prefix before a value, and inputs/outputs the values true/false. Although the bool data type is hardly used by us, it can be very useful when we need to store the on/off states or as conditions in place of integers 0 or 1. This data type is the internal output of if…else statements. For e.g., the code inside an if statement runs only when its condition is 'true', if its false, the else statements are executed.

For e.g.,

```
bool state = false;
```

String: String refers to a data type used to store text, such as words and sentences. Strings can contain letters, numbers, spaces, and symbols which are treated as text. Using strings makes it easier to perform operations like joining, comparing, and finding the length of text. In Arduino IDE, strings can be

created using a character array (`char[]`). A character array stores characters and ends with a null terminator (\0)[2]. The array can also represent an empty string. Look at the following use cases of the character array in Arduino IDE:

```
char author[] = "Agrim";
```

This snippet declares a string array named `author[]` and it stores the text `"Agrim"`. This is the most common method to create strings and it automatically stores the null terminator along with the length of text.

```
char code_by[8] = {'A', 'g', 'i', '@', '4',
'0', '4', \0'};
```

This is the manual declaration. It is the real way how the code stores string arrays internally which is essentially same as

```
char code_by[] = "Agi@404";
```

In order to assign string value to an array, we use `strcpy()` function. It takes the source string as first argument and assigns it to the destination string , which is second argument. Note that the destination must be large enough to store string. For e.g., here source size is 13 chars + null = 14 bytes, and destination is of 20 size.

```
char source[] = "Hello, world!”; char
destination[20]; strcpy(source, destination);
```

!! Arduino has limited memory. Use short strings or consider String objects (capital S) for dynamic text:

```
String myText = "This can grow and shrink";
```

But char[] are more stable on UNO’s tiny memory as compared to String

[2]*The null terminator (\0) is a special character with value of integer 0. It marks the end of a string in a character array. It works like a full stop (.) for the compiler, telling it to stop reading the string. When we write a string normally, the compiler automatically adds \0.*

Functions

While writing the code, we divide it into multiple blocks for clean, readable and organized code structure. Functions are these blocks which perform specific tasks in the code. A function is a block of code that runs when called. It may take input values and may return a value.

`void setup() {}`

It is used to initialize the communication(like serial, I2C, etc.), define pins and do one-time initial work. It runs once at startup to prepare Arduino: defining pin modes (INPUT/OUTPUT), initializing communication (Serial, sensors) , setting starting values. Think of it as 'getting dressed' before starting your day.

```
void setup() {
  pinMode(13, OUTPUT);
}
```

The above code snippet gives instruction to Arduino Uno that pin 13 will be used as an output. This task is only done once, hence, it is written in `setup()`.

`void loop() {}`

The code written in this function repeats indefinitely. It is used when a task is to be done by Arduino again and again, for e.g., to check sensor readings, run LEDs, check conditions, etc. It is executed after `setup()` and runs till Arduino is powered.

```
void loop() {
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```

This code snippet:
• Sends a HIGH (1) signal to pin 13 using `digitalWrite()`.
• Waits for 1 second using the `delay()` function. `delay()` function is used to pause the program for certain amount of time, in milliseconds[1].
• Sends a LOW (0) signal to pin 13.
• Waits for 1 second again.
• Repeats this process forever.
To understand `setup()` and `loop()`, we can imagine `setup()` as starting the mobile phone and `loop()`as the doom scrolling of reels .
A semicolon (`;`) marks the end of the instruction. It is not used when an instruction that opens a block by braces({).
`void setup() {` ← no semicolon
`if (x > 5) {` ← no semicolon
It is mandatory to use `setup()` and `loop()` functions and code will not run without them.
Summary from this chapter:
• Functions are the collection of instruction that are needed to be called/run again and again.
• These are of various types. Void means absence of any type or type less.
• setup() & loop() are the main functions executed Arduino. Any other code or function reside in either of the two.

[1] *1 second = 1000 milliseconds.*
[2] *void means that the function we've created doesn't return any value and only performs the task. However, if we need the function to return any value, we specify its type instead of void- like int, float, etc. For e.g.,*

```
int sum(int a, int b){int result = a+b; return result;}
```

Our function takes integer values for a and b, hence, we define that while function creation. Since the function is defined as integer, it returns integer values. NOTE: If your function need no inputs, we won't define them.

Modes and States

Modes: These refer to the assignment of tasks that are performed by pins of MCU. An MCU like the one used in Arduino has many pins which can perform various tasks. Some pins have specialized capabilities. These are called GPIO or general purpose input/output pins. According to their capability, we can use these pins for various tasks like taking inputs, giving out signals, PWM, analog, etc. It is done by the help of assigning modes or tasks in the code.

Modes are of 4 types:

• Input: This makes the pin act as an input. Input can be of two types, digital or analog depending on the capabilities of the pin.

• Output: This makes the pin an output. Arduino can supply outputs as either HIGH or LOW and on some pins, PWM.

• Input Pull-up: This connects the pin from the built-in resisters of the chip. It is used to eliminate random floating noise and reads the input as HIGH by default. When a switch or any other sensor connects this pin to GND with certain maximum resistance, the input becomes LOW.

PWM stands for Pulse Width Modulation. Microcontroller boards like the Arduino Uno can't really output arbitrary voltages between 0V and 5V. Instead, they fake different voltage levels by rapidly switching the output between LOW and HIGH (on and off). This technique is used to control LED brightness, motor speed, etc., because the average voltage appears different depending on how long the signal stays HIGH versus LOW. Think of it like rapidly switching a fan on and off, you will see that its speed has reduced. Arduino does the same thing internally. These pins are marked with a ~ mark.

Analog pins (A0-A5) are capable of reading any voltage between 0V to 5V. Like PWM, Arduino uses a trick here too to convert

voltages into 0s and 1s. It uses a device called ADC or analog to digital convertors to convert Analog voltages into on/off states. It employees a 10-bit ADC, meaning 0V to 5V is divided into 0-1023 digits of input readings.

Syntax:

We use `pinMode()` in-built function to define the mode of a pin. It takes two arguments, pin number and mode. For e.g.,

```
const int led = 3;
bool flipper = false;
void setup() {
  pinMode(led, OUTPUT);
}
```

This code declares a constant integer named led as 3, a boolean variable flipper as false and sets the mode of pin led as OUTPUT. Since pin 3 on Arduino is PWM capable, we can dim this pin digitally:

```
void loop() {
  analogWrite(led, dim); delay(5);
}
```

Here, dim is a variable that can store any value between 0-255. The brightness of led or apparent output voltage will depend on the value of dim.

In the case of inputs, we set the pin mode to `INPUT` or `INPUT_PULLUP` as per the need. Pins A0-A5 can read analog values via `analogRead(pin)` function. For `INPUT_PULLUP` or `HIGH/LOW`, we use `digitalRead(pin)` function. For e.g., `x = analogRead(sensor);` reads the output of pin named sensor and stores its values in an integer variable x.

`FUN FACT:` When we write `const int sensor = A0;` on IDE, it internally recognizes the pin number of A0(or any pin) on Arduino UNO as 14(or corresponding value. This makes the same code run on various boards.

States: An Arduino program usually exists in multiple states. State is basically the current 'mode' or behavior of the machine. Using states, we may escape multiple if…else statements for simple mode selection or other tasks. We need states because we don't want our machine to do everything at once (as you can't both study and play in the same time). For e.g., An LED may exist in the following states in a program:

ON OFF DIM BLINK

States are usually stored in a variable. This example shows the implementation of a button via INPUT_PULLUP. Note the use of conditions and try to implement above mentioned states in it:

```
int state = 0, lastButtonState = 1, current= 0;
const int button = 2, led = 3;
void setup() {pinMode(button, INPUT_PULLUP);
  pinMode(led, OUTPUT);}
void loop() {
  current = digitalRead(button); /*state of a
pin/peripheral also means HIGH or LOW*/
  if (current ==LOW && lastButtonState==HIGH) {
    state = !state; //toggle
  }
  lastButtonState = current;
  digitalWrite(led, state); delay(100);}
```

Multiple variables of same type can be declared in one line like above(!using commas!). `/*__*/` is the syntax to write multiline comments in Arduino IDE. `//` is to initiate a single line comment (that's why it has no terminating token). It is perfectly valid to write multiple instructions in a single line like how written above. The braces ({}) define the scope or *'visibility'* of instructions they contain. Like anything in between the braces of void loop() will be visible only to it. (Anything outside any of the braces is visible to all the functions but can be overridden, ie, it becomes global)

Calculation

To perform calculations in Arduino IDE, we use arithmetic operators like addition(+) and subtraction(-), multiplication(*) and division(/). Modulo (%) returns the remainder of the division. For e.g.,

```
int a = 10; int b = 2; mod = a%b;
```

The value of mod variable will be the remainder, ie, 0.
As discussed before, the float data type yields more precise results. For e.g., in this snippet, the div is an int:

```
int a = 1;
int b = 2;
div = 1/2;
```

The value of div is 0 as decimal part(0.5) is removed. Division does not default to float. If both numbers are int, the result is an int and the decimal part is removed. If even one number is float, the result becomes a decimal value. Consider this snippet

```
a = 1;
b = 2.0;
div = a/b;
```

The value of div here is 0.5 since b is a float (2.0), despite that a is an integer. *Note that division by 0 is undefined and can cause malfunctions on Arduino Uno.*
The abs function output the absolute value of the variable, ie, it yields only the magnitude and not the sign of the no. For e.g., `abs(-10);` removes negative(-) sign from the number and outputs 10. It is useful when we are concerned with the magnitude only.
`min()` and `max()` functions return the smallest and the largest number amongst the given inputs respectively. Look at the following examples:

```
a = min(10, 2.0); b = max(10, 2.0);
```

Here, the value of a is 2.0(the smaller value) and of b is 10(the larger value).

To find the square and square root of the input, we use `sq()` and `sqrt()` functions respectively. In the following snippet, The value of `val_square` = 16 and value of `val_root` = 2.0:

```
#include <math.h>
const int val = 4;
int val_square = sq(val);
float val_root = sqrt(val);
```

- Imports `math` library.
- Declares integer `val` as 4.
- Built – in, `math library` not needed
- It requires `math library`.

sq() is built into Arduino and does not require any library, while sqrt() requires the math library. The math library[3] imports some functions that make working with complex math easier; one such function is power or `pow()` function that finds the value of given base raised to the power of given exponention. It is to be noted that the output of `pow()` by default is in double data type. The difference between float and double is that double takes up more memory on board to store data with more precision than float values; however, on simple boards like Arduino Uno, both behave the same.

```
int base = 2;
int exponention = 3;
int power = 4;
float cube = pow(base, exponention);
float val = pow(base, power);
double square = pow(base, 2);
```

`cube` = `base` raised to `exponention`, ie, 2*2*2 = 8.0

Same precision on Uno (try to predict the output)

Map Function: Before we dive straight into `map()`, we will first understand its need. While programming in Arduino, we frequently need to convert one range of values into another.

[3] Library is supposed to be visible to every function, hence, it is written at the top of the code. *You'll learn more about libraries in later chapters.*

For instance, we might need to convert sensor readings into percentages for display, control output modes, adjust PWM, etc. Without map() function, we would end up dividing integers for such repetitive tasks again and again for different values. It does the heavy lifting and arranges values in a clean understandable syntax.
Think of it like a bus conductor- he can arrange 40 passengers on the seats of the bus so that the driver (or programmer) can focus on his task.
The `map()` function takes 5 arguments.
•`from_val` is the value that needs to be converted
•`fromLow` and `fromHigh` define the range in which the value currently lies
•`toLow` and `toHigh` define the range to which the value must be converted

```
const int sensor = A0;
int sensorValue = 0;
int state = 0; //state is initialized to 0
void setup() {
  pinMode(sensor, INPUT);
}
void loop() {
  sensorValue = analogRead(sensor);
  state = map(sensorValue, 0, 1023, 0, 4);
}
```

If the number of input values is divisible by the number of output values, map() distributes values evenly. If it's not divisible, the highest output value will usually be reached only at (or very near) the maximum input. It does not clamp values within a range like min/max functions- it just carries out integral division. The map() function mathematically rescales the input range. When the ranges are not perfectly divisible, some output values receive slightly more input values than others.

0-255	0
256-511	1
512-767	2
768-1022	3
1023	4

Since the input range has 1024 values (0–1023) and the output range has 5 values (0–4), the division is not even. Because of this, the highest output value (4) is produced only when the input is exactly 1023. If the input range were perfectly divisible by the number of output values, all steps would be equal in size. Note that map() doesn't clamps, meaning if input becomes more than 1023, it'll just provide the larger unwanted output.

all the array operations can be used directly on the char array. There are a few additional methods that are used to modify and do calculations on chars.

You can calculate the number of characters (excluding '\0') using `strlen(text);` For e.g., here `length` = 3.

```
char name[] = "Agi";
int length = strlen(name);
```

To access individual characters, we input character no.(starts from 0) into the bracket of char value. For e.g., here, `first` = "A" and `last` = "o".

```
char word[] = "Arduino";
char first = word[0];
char last  = word[6];
```

Modifying the individual characters is similar. Now the `text` = "wow" since first char is converted to "w":

```
char text[] = "how";
text[0] = 'w';
```

To convert a char into other data types, we can use

`atoi()`	means array to integer
`atol()`	means array to long
`atof()`	means array to float

```
char num[] = "123";
int value= atoi(num);
```

value is integer; ie 123.

These default to 0 if no value is found in array to be convertible.

To search character, `strchr(text, value);` or substring, `strstr(text, value);` is used.

For e.g., this condition checks whether “x” exists in the array named “word”.

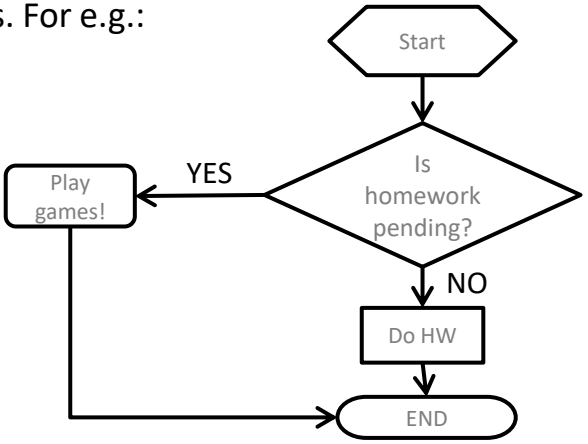
```
if (strchr(word, 'x')) {}
```

OPERATOR	USAGE	FUNCTION	USAGE
+	a + b	min()	min(a, b)
-	a - b	max ()	max(a, b)
*	a * b	sq()	sq(a)
/	a / b	sqrt()	sqrt(a)
%	a % b	pow()	pow(base, exp)
		abs()	abs(a)
		map()	map(value, inputMin, inputMax, outputMin, outputMax)

Note: You can include the inbuilt library math.h into your Arduino sketches to use inbuilt mathematical functions if your code doesn’t compile on using them. You’ll learn more about libraries in the following chapters.
Values like HIGH, LOW, TRUE, FALSE, A0, etc. are defined in Arduino core. HIGH and TRUE corresponds to 1, LOW and FALSE corresponds to 0, A0 to A5 corresponds to 14-19 in IDE definitions. This makes logic and code easy to read and clear to debug. In the example sketch of chapter state was declared to 1 to match how INPUT_PULLUP readings are processed (HIGH for no input, LOW for input). Because we used INPUT_PULLUP, the pin stays at 1 (HIGH) when the button is NOT pressed. By starting our variable at 1, we ensure our project starts in the correct "resting" state.

Decision Making

To take decisions in programming, we make various comparisons on values. To visualize a set of decisions or *logic,* we can use flowcharts. For e.g.:



Here, we make decisions using if/else statements like:

```
if(a<b){
  strcpy (result, "small");
  }else{
  strcpy (result, "large");
  }
```

If there are multiple if statements in a single logic, we use '`else if`' in between if and else. In the above snippet, a<b is a conditional statement or expression. If the output(of "a<b") is true, result will equal to small, else, large.

To group or evaluated different conditional statements separately, we use grouping operators. Parentheses "()" is the primary grouping operator. The expressions are written and separated in Arduino IDE using parentheses.

A few logical operators are:

- **&&** is for AND. It outputs true only if both the conditions are true.

|| means OR. This means that either of the conditions can be true for the logic to give true output.

- **!** is for NOT. It reverses the output of an expression.

Note: We don't use these logical operators everywhere. They are used to group multiple conditions as needed.

Along with if…else, we use comparison operators to make logics. The output of comparisons or expressions like a<b is in "true" or "false". Working with if…else and logical operators, they play a vital role in creating conditions and logics. The Comparison operators in Arduino IDE are:

- **<** means smaller than. It outputs true only when first variable is smaller than the second one.
- **>** means grater than. It outputs true only when first variable is greater than the second one.
- **==** is for comparative equal to. It outputs true only when both the variables are equal.[1]
- **<=** means smaller than equal to. It outputs true when the first variable is either smaller than the second one, or is equal to it.
- **>=** means smaller than equal to. It outputs true when the first variable is either greater than the second one, or is equal to it.
- **!=** is for not equal to. It outputs true every time except when first and second operators are equal. It is the opposite of ==.

The use of logical operators is that they are used to evaluate multiple expressions with a certain relationship with them. Like say && is used to evaluated two different expressions and outputs true only when both of them are true. On the other hand, expressions or comparisons evaluate the relationship or perform comparisons on different values.

[1] The difference between = and == is that = is for assignment; ie, when we need to assign a value to a variable, we use =. The later is used for comparisons.

Have a look on this snippet. It uses Serial communication to enable Arduino to send messages directly to the PC (try to find output and execution yourself):

```
int num = 2;
void setup() {
  Serial.begin(9600);
  if (num % 2 == 0) {  // Check even or odd
    Serial.println("The number is even");
  } else {Serial.println("The number is odd");}
  if (num > 0) { // Check sign of number
    Serial.println("The number is positive");}
  else if (num < 0) {
    Serial.println("The number is negative");}
  else {Serial.println("The number is 0");}
}
void loop() {/* nothing in loop */}
```

Truth Table: It refers to the outputs of a logic/ logical operator, that are arranged in the form of table. A truth table shows all possible input combinations of a logical operator and the corresponding output. Outputs are usually represented as True/False or 1/0. Some truth tables also include states or other configurations but they are essentially input modifiers. We use truth tables to understand how a logic, code or circuit works internally. In programming languages like Arduino, logical operators are written as &&, ||, and !. The truth table of &&, || and ! are provided below for better understanding:

Input	Output
True	False
False	True

Truth table of NOT/!

Input 1	Input 2	Output
True	True	True
True	False	False
False	True	False
False	False	False

Truth table of AND/&&

Input 1	Input 2	Output
True	True	True
True	False	True
False	True	True
False	False	False

Truth table of OR/||

Loops: Loops are a set of instructions that run again and again until a specific condition is met. They are different from loop() function in a sense that loop() doesn't requires any conditions and it is the firmware part that keeps the machine *'alive*'. Loops are of 2 main types:

• **For loop-** When number of iterations(steps) are know beforehand, we use for loop. Its syntax requires an integer counter, then a 'while' condition, then an action that is done after every iteration. For e.g., `for(int i=0;`i>steps; i++) . Here ++(incrementation) means i = i + 1 (we can also write - - if required anywhere). i or any other variable can be declared inside or outside for statement. Look at this snippet:

```
for(int i = 0; i < 5; i++){
  digitalWrite(13, HIGH);
  delay(200);
  digitalWrite(13, LOW);
  delay(200);
}
```

• **While loop-** When no. of iterations is unknown beforehand, we use while loops. We use only the while condition to run the loop, hence it need no integer counter and maybe no after iteration action (like incrementation). It runs while condition given is TRUE. For e.g.,

```
while(digitalRead(2) == LOW) {digitalWrite(13, HIGH);}
```

Time, communication and libraries

We have used delay till here in some basic programs. delay() stops the execution of the whole program. And is especially redundant if we wish to handle multiple tasks simultaneously. To have delays in instructions without pausing the whole program, we make use of millis() function. millis() function counts the no. of milliseconds since the program started running. We can make use the millis counter and a value variable to achieve non blocking delays. When the difference of current timer and millis = desired value, we reset the timer and do our required tasks(we use subtraction because comparison will fail when millis() overflows after 49 days). Look at the following example code to better understand its usage:

```
unsigned long previous = 0, now = 0; //unsigned
//saves only the positive numbers including 0,
//giving a larger usable range
const long interval = 1000; //1 second
void loop() {
  if(now - previous >= interval){now =millis();
    previous = now; //resets
    digitalWrite(13, !digitalRead(13));}//! is
//used to reverse the state (HIGH/LOW) of pin
//which it currently is in
}
```

Communication is one of the most important part of any endeavor. Whether its via e-mail or discord, we need a strong communication between collaborators to accomplish a team build. In a similar way, Arduino can act as the head of a *'team'* of peripherals to make sophisticated projects. Components are parts of a circuit that are rather simple to use; for e.g., an LED just needs a resistor along with suitable power to make it glow.

Other parts like a display are a collection of other components and IC controllers. These are the peripherals. These components are a separate *'device'* on their own. Arduino needs special communication methods to *'talk'* to them. In the same manner as we need to know the language of the person we are communicating with, or at least a translator, we need protocols to communicate with such peripherals. Following are some protocols supported by Arduino UNO:

IIC/I2C: Inter Integrated Circuit is a simple 2 wire protocol that is used by many peripherals like displays, pin extenders, sensors, etc. This protocol employees a data pin (SDA - pin A4) and a clock pin (SCL - A5) to control low speed processors.

• **SPI**: This protocol has 4 wires but offers way faster data transfer as compared to I2C. It is used by devices that need high speed data transfer and robust architecture. It is used by LCDs, flash memory, sensors, SD cards, and ADCs.

Serial Peripheral Interface uses a master-slave architecture via pins:

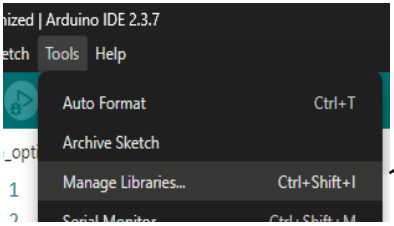
1. SCLK- Serial Clock (output from master) - Pin 13 on UNO
2. MOSI: Master Out Slave In (data from master to slave) - 12
3. MISO: Master In Slave Out (data from slave to master) – 11
4. SS/CS: Slave Select/Chip Select (active low signal to select a specific device) – 10

UART/USART: UART (Universal Asynchronous Receiver/Transmitter) is a serial communication protocol used to send and receive data between electronic devices using TX (Transmit) and RX (Receive) pins. UART communication does not use a clock signal. Instead, both devices must agree beforehand on a transmission speed called the baud rate (the speed at

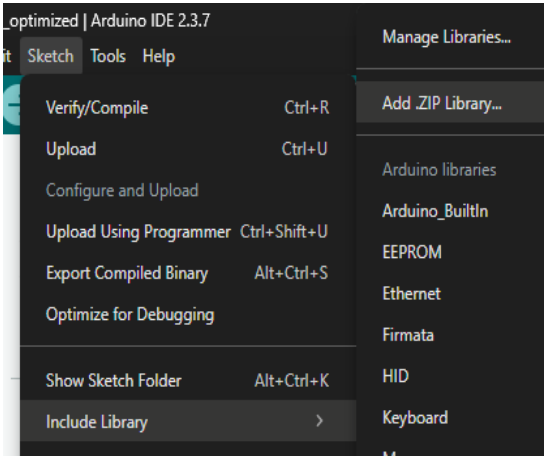
[1]*Arduino Uno contains a separate microcontroller, the ATmega16U2, which converts USB communication from the computer into UART signals understood by the ATmega328P. UART is the primary method used to communicate with a computer.*

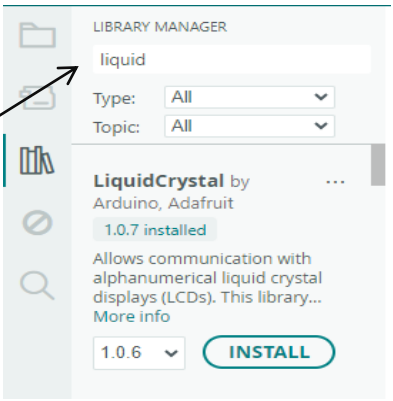
which data bits are sent).UART is widely used in microcontrollers and embedded systems.
USART (Universal Synchronous/Asynchronous Receiver/Transmitter) is an extended version of UART. It can operate in:
•Asynchronous mode (same as UART), or
•Synchronous mode, where an additional clock signal keeps both devices precisely synchronized.
Because of this flexibility, USART hardware can be used in more communication scenarios while still remaining compatible with
In both UART and USART communication, connections are made in a cross configuration:
•TX (Device 1- pin 1 on UNO) → RX (Device 2)
•RX (Device 1- pin 0 on UNO) → TX (Device 2)
These pins are usually avoided for external components because they are already connected to the USB communication system used for uploading programs and serial monitoring.
Libraries: Libraries are collections of pre-written code that help us interface with complex components and perform advanced tasks more easily. Instead of writing everything from scratch, we can use libraries to control sensors, displays, communication modules, and other peripherals. Although these features can be implemented without libraries, using them allows us to build upon others' work, making development faster, simpler, and more reliable. Different libraries enable us to perform different tasks, hence, all libraries cannot be explained in this booklet. General procedure is mentioned which will explain how libraries can be implemented in the program.
INSTALLING: Before using a library, we need to first install it and make it available to the IDE. It can be done in the IDE itself by navigating to library manager in tools/sketch tab and installing the required library by searching. Note that some

libraries are inbuilt and don't need to be installed, for e.g., math.h. We can also include custom libraries by installing their .ZIP package in the IDE. We can even make our own libraries with C++ and saving in .h extension!



OR





INCLUDING: Once installed, the libraries can be imported in the code by using #include keyword, followed by library name in <__>.
For e.g., `#include <LiquidCrystal_I2C.h>`

in the very first line of the program. Servo library make it easier to run servo motors by Arduino boards by encoding their motion in degrees.

EXAMPLES: In the following examples, we will be understanding how to use the LCD library for I2C interface, IR remote to read commands and serial. Although serial is inbuilt in IDE, it demonstrates general message sending and uses UART protocol internally to communicate to the Arduino. Note that different libraries may have different syntax and usage. Refer to the documents of those libraries or the example sketches (in file tab) provided by them.

```
#include <LiquidCrystal_I2C.h> //-lcd example-
LiquidCrystal_I2C lcd(0x27,16,2); /*sets the
LCD address to 0x27 for a 16x2 display*/
void setup(){lcd.init(); // initialize the lcd
  lcd.backlight(); //turns on backlight
  lcd.setCursor(0,0); //setCursor(column, row)
  lcd.print("Hello, world!"); //prints on lcd
  lcd.setCursor(0,1); lcd.print("by Agi!");
// same addresses overwrite
  delay(500); lcd.clear();//clears after 500ms
} /*--*/ void loop() {/*empty*/} //------------
```

```
void setup() { //--serial read/write example--
  Serial.begin(9600);//set baud at 9600
  Serial.println("Hello!");//prints in new line
} //--
void loop() {
  if (Serial.available() > 0) { //check data
    char data = Serial.read(); //read one char
    Serial.print("Read:");//prints in same line
    Serial.println(data);}
} //----------------------------------------
```

```
#include <IRremote.h> //-irremote read example-
int RECV_PIN = 2; IRrecv irrecv(RECV_PIN);
decode_results rslts; //stores data in rslts
void setup(){
  Serial.begin(9600); irrecv.enableIRIn();}//--
void loop() {
  if (irrecv.decode(&rslts)) {
    Serial.println(rslts.value, HEX); //hex
    irrecv.resume(); //continues reading
  }
} //----------------------------------------
```

Power Management

There are multiple ways to power our Arduino projects:

• **USB**: The native voltage of Arduino Uno is 5V, hence, it can be powered directly from USB via a computer. USB power is typically limited to about 500 mA because of the USB polyfuse of Arduino and port limits. Its enough for many applications but when we need to run motors or other power hungry peripherals, it can be inadequate. To make our projects portable, we can either use batteries and/or power bank/wall adapter with correct wiring circuitry.

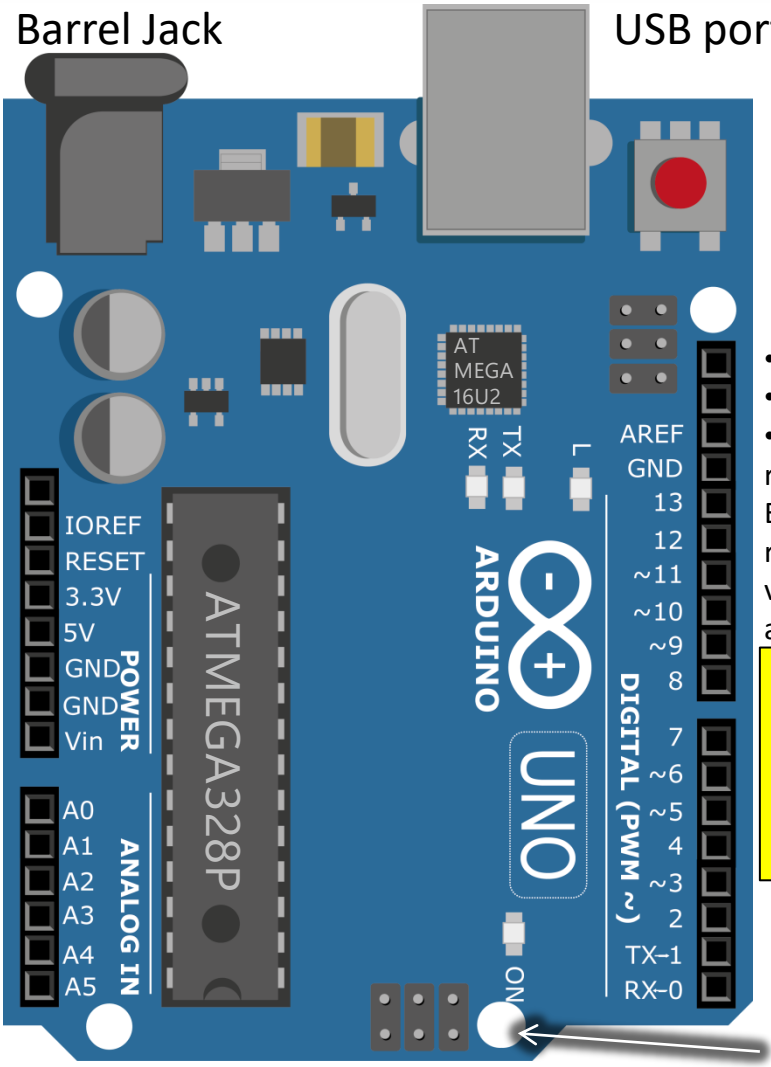
• **Barrel Jack**: A DC barrel jack can also be used to power the Arduino. The input voltage for barrel jack ranges from 7V-12V for Uno R3 and 7V-24V for Uno R4. While connecting a barrel jack to power Arduino, we must ensure that outer shell of the jack must be –ve while centre must be +ve, with appropriate pin size. When powered via the barrel jack, voltage first passes through the onboard voltage regulator, which converts it to 5V for the microcontroller. VIN (voltage input) pin carries the unregulated input voltage. VIN allows higher input voltage and current compared to USB, but external devices should *ideally* use a separate power supply instead of drawing large current through Arduino. Adaptor label must be referred before use.

+ve centre symbol

• **Battery**: Since the barrel jack internally connects to VIN, a battery can also be connected through the VIN pin (within safe voltage limits). GND is connected to –ve to battery to make Arduino battery powered. However, polarity must be paid special attention during this process.

• **3V3**: This pin PROVIDES 3.3V, with limited amps, to power 3.3V compatible components. The Arduino Uno itself operates at 5V logic, so it cannot normally run from the 3.3V pin.

Pin Reference



- I2C (SCL) – A5
- I2C (SDA) – A4
- AREF (analog reference): External reference voltage for analog inputs.

!! Use only when configured with `analogReference()`.

Analog pins can also be used as digital input or output pins. PWM-enabled pins are marked with (~). The TX LED blinks when the Arduino transmits data through the TX line; similarly, the RX LED blinks when data is received. The ON LED glows when the board is powered. 'L' indicates the built-in LED connected to digital pin 13. The Reset button restarts the program from the beginning.

An Example Program

The following project demonstrates a simple **RC car controlled using an IR remote(IRremote library 2.8.0)** by using the concepts of this booklet. The car can be controlled using a common TV remote, or you may build your own IR remote.

Components Required for the car:

1. Arduino Uno (or any compatible board)
2. Motor Driver (L293D or similar)
3. 2 DC Motors
4. IR Receiver (IR demodulator module)
5. A few LEDs (for indication) with appropriate resistors
6. 9V battery or power bank

The IR codes used in this example may vary depending on the remote. You can read the commands sent by the buttons of your TV remote from the last IR lib “read” example.

Circuit diagram and an e-copy of this project’s code is encoded in the following QR codes:

Circuit

Code

```
#include <IRremote.h>
const int recv = 2;
IRrecv irrecv(recv);
decode_results results;//store results
const int m1L = 3, m1R = 4, m2L = 5, m2R = 6; // Motor pins
const int leftLED = 9, rightLED = 11; // LEDs
int state = 0;//0:stp 1:fwd 2:bck 3:lft 4:rght
unsigned long timer = 0;
const unsigned long runTime = 3000; // 3 seconds
void setup() {//==================================
  pinMode(m1L, OUTPUT); pinMode(m1R, OUTPUT);
  pinMode(m2L, OUTPUT); pinMode(m2R, OUTPUT);
  pinMode(leftLED, OUTPUT); pinMode(rightLED, OUTPUT);
```

```
irrecv.enableIRIn(); // start IR receiver
} void loop() {
  if (irrecv.decode(&results)) {// ---- READ REMOTE ----
    unsigned long cmd = results.value;
    if (cmd != 0xFFFFFFFF) {//to prevent repetition
      if (cmd == 0x2FD9867) {// FORWARD- Update these!
        state = 1;
        timer = millis();
      }else if (cmd == 0x2FDB847) {// BACKWARD
        state = 2;
        timer = millis();
      }else if (cmd == 0x2FD42BD) {// LEFT
        state = 3;
        timer = millis();
      }else if (cmd == 0x2FD02FD) {// RIGHT
        state = 4;
        timer = millis();
      }else if (cmd == 0x2FD847B) {state = 0;}// STOP BUTTON
      else {state = 0;} //UNKNOWN BUTTON→STOP
    }irrecv.resume();}
  if (millis() - timer > runTime) {state = 0;}//Safety timer
  if (state == 1) { //---- MOVEMENT CONTROL ---- FORWARD
    digitalWrite(m1L, HIGH); digitalWrite(m1R, LOW);
    digitalWrite(m2L, HIGH); digitalWrite(m2R, LOW);
    digitalWrite(leftLED, LOW); digitalWrite(rightLED, LOW);
  }else if (state == 2) { // BACKWARD
    digitalWrite(m1L, LOW); digitalWrite(m1R, HIGH);
    digitalWrite(m2L, LOW); digitalWrite(m2R, HIGH);
    digitalWrite(leftLED, LOW); digitalWrite(rightLED, LOW);
  }else if (state == 3) {// LEFT
    digitalWrite(m1L, HIGH); digitalWrite(m1R, LOW);
    digitalWrite(m2R, HIGH); digitalWrite(m2L, LOW);
    digitalWrite(leftLED,HIGH); digitalWrite(rightLED, LOW);
  }else if (state == 4) {// RIGHT
    digitalWrite(m1R, HIGH); digitalWrite(m1L, LOW);
    digitalWrite(m2L, HIGH); digitalWrite(m2R, LOW);
    digitalWrite(leftLED, LOW); digitalWrite(rightLED,HIGH);
  }else { // DEFAULT → STOP
    digitalWrite(m1L, LOW); digitalWrite(m1R, LOW);
    digitalWrite(m2L, LOW); digitalWrite(m2R, LOW);
    digitalWrite(leftLED,HIGH);digitalWrite(rightLED,HIGH);}
}
```

THE END… *Really?*

Arduino is not just a device; it is a tool.
If electronics is math, then Arduino is like a simple calculator. You have accomplished the milestone of learning an entry-level MCU. Coding on Arduino UNO is not just writing paragraphs on a computer; it is about creating your own smart electronic creations—ones that carry human ambitions from earthly matters to the space. The underlying technology that governs the smartest machines starts from here.

Stay curious.

Way Forward: This is not an end, it is a new beginning. Your next intuition might be learning fundamental electronic components. You can consider

Persilience

Beyond Circuit- Basics & Principles

Persilience is focused on providing the simple explanations for concepts that often seem complex.

This book covers basic Arduino coding & concepts, including:

- Reading Inputs
- Controlling Outputs
- Time Handling
- Designing Systems

If this helped you, star the project on GitHub

PER-CM-Dig

Published by

Persilience

Cover Design by Author & Pictures from Pixabay.

ig: @bibliophile_agi | @alternate_arnav | @cosmicdev404
Github: cosmicdev404
Editorial Advisor: Arnav Sharma