S.no	Title	Page
1	Introduction	
2	Arduino - board description	
3	Download arduino ide software.	
4	Arduino - program structure	
5	Arduino - data types	
6	Arduino - variables & constants	
7	Arduino - operators	
8	Arduino - control statements	
9	Arduino - loops	
10	Arduino – functions	
11	Arduino - strings	
12	Arduino - time	
13	Arduino – arrays	
14	Arduino - i/o functions	
15	Arduino - character functions	
16	Arduino - math library	
17	Arduino - pulse width modulation	
18	Arduino - random numbers	
19	Arduino - communication	
20	Arduino - blinking led	
21	Arduino - fading led	
22	Arduino - humidity sensor	
23	Arduino –water level sensor	
24	Arduino - ultrasonic sensor	
25	Arduino - servo motor	

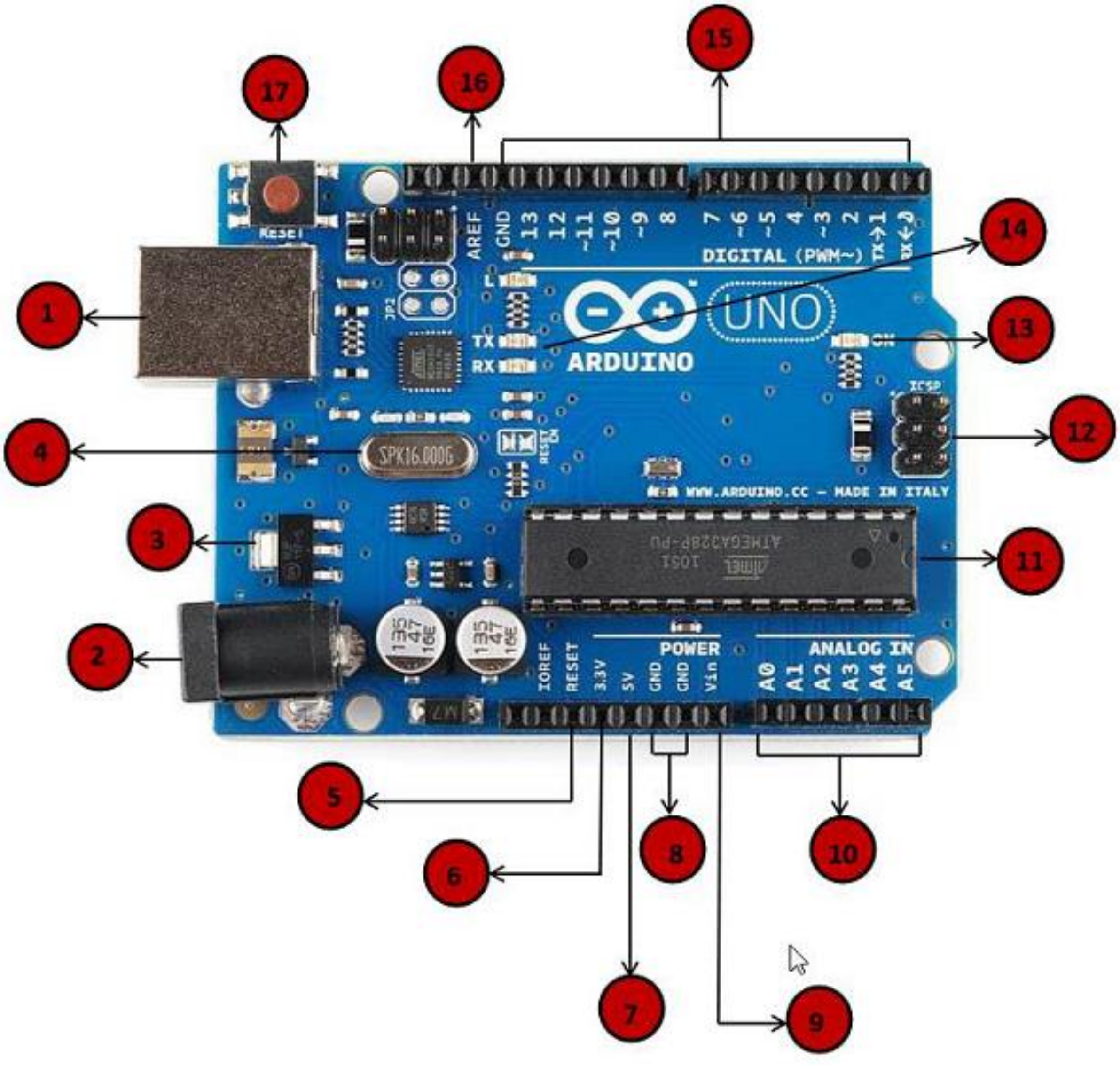
1.INTRODUCTION

The arduino environment has been designed to be easy to use for beginners who have no software or electronics experience. with arduino, you can build objects that can respond to and/or control light, sound, touch, and movement. arduino has been used to create an amazing variety of things, including musical instruments, robots, light sculptures, games, interactive furniture, and even interactive clothing.

Arduino is used in many educational programs around the world, particularly by designers and artists who want to easily create prototypes but do not need a deep understanding of the technical details behind their creations. because it is designed to be used by nontechnical people, the software includes plenty of example code to demonstrate how to use the arduino board's various facilities. though it is easy to use, arduino's underlying hardware works at the same level of sophistication that engineers employ to build embedded devices. people already working with microcontrollers are also attracted to arduino because of its agile development capabilities and its facility for quick implementation of ideas. arduino is best known for its hardware, but you also need software to program that hardware. both the hardware and the software are called "arduino." the combination enables you to create projects that sense and control the physical world. the software is free, open source, and cross-platform. the boards are inexpensive to buy, or you can build your own (the hardware designs are also open source). in addition, there is an active and supportive arduino community that is accessible worldwide through the arduino forums and the wiki (known as the arduino playground

2.ARDUINO - BOARD DESCRIPTION

In this chapter, we will learn about the different components on the Arduino board. We will study the Arduino UNO board because it is the most popular board in the Arduino board family. In addition, it is the best board to get started with electronics and coding. Some boards look a bit different from the one given below, but most Arduinos have majority of these components in common.



1	**Power USB** Arduino board can be powered by using the USB cable from your computer. All you need to do is connect the USB cable to the USB connection (1).
2	**Power (Barrel Jack)** Arduino boards can be powered directly from the AC mains power supply by connecting it to the Barrel Jack (2).
3	**Voltage Regulator** The function of the voltage regulator is to control the voltage given to the Arduino board and stabilize the DC voltages used by the processor and other elements.
4	**Crystal Oscillator** The crystal oscillator helps Arduino in dealing with time issues. How does Arduino calculate time? The answer is, by using the crystal oscillator. The number printed on top of the Arduino crystal is 16.000H9H. It tells us that the frequency is 16,000,000 Hertz or 16 MHz.
5,17	**Arduino Reset** You can reset your Arduino board, i.e., start your program from the beginning. You can reset the UNO board in two ways. First, by using the reset button (17) on the board. Second, you can connect an external reset button to the Arduino pin labelled RESET (5).
6,7 8,9	**Pins (3.3, 5, GND, Vin)** • 3.3V (6) − Supply 3.3 output volt • 5V (7) − Supply 5 output volt • Most of the components used with Arduino board works fine with 3.3 volt and 5 volt. • GND (8)(Ground) − There are several GND pins on the Arduino, any of which can be used to ground your circuit. • Vin (9) − This pin also can be used to power the Arduino board from an external power source, like AC mains power supply.

10	**Analog pins** The Arduino UNO board has six analog input pins A0 through A5. These pins can read the signal from an analog sensor like the humidity sensor or temperature sensor and convert it into a digital value that can be read by the microprocessor.
11	**Main microcontroller** Each Arduino board has its own microcontroller (11). You can assume it as the brain of your board. The main IC (integrated circuit) on the Arduino is slightly different from board to board. The microcontrollers are usually of the ATMEL Company. You must know what IC your board has before loading up a new program from the Arduino IDE. This information is available on the top of the IC. For more details about the IC construction and functions, you can refer to the data sheet.
12	**ICSP pin** Mostly, ICSP (12) is an AVR, a tiny programming header for the Arduino consisting of MOSI, MISO, SCK, RESET, VCC, and GND. It is often referred to as an SPI (Serial Peripheral Interface), which could be considered as an "expansion" of the output. Actually, you are slaving the output device to the master of the SPI bus.
13	**Power LED indicator** This LED should light up when you plug your Arduino into a power source to indicate that your board is powered up correctly. If this light does not turn on, then there is something wrong with the connection.
14	**TX and RX LEDs** On your board, you will find two labels: TX (transmit) and RX (receive). They appear in two places on the Arduino UNO board. First, at the digital pins 0 and 1, to indicate the pins responsible for serial communication. Second, the TX and RX led (13). The TX led flashes with different speed while sending the serial data. The speed of flashing depends on the baud rate used by the board. RX flashes during the receiving process.
15	**Digital I/O** The Arduino UNO board has 14 digital I/O pins (15) (of which 6 provide PWM (Pulse Width Modulation) output. These pins can be configured to work as input digital pins to read logic values (0 or 1) or as digital output pins to drive different modules like LEDs, relays, etc. The pins labeled “~” can be used to generate PWM.

16	**AREF** AREF stands for Analog Reference. It is sometimes, used to set an external reference voltage (between 0 and 5 Volts) as the upper limit for the analog input pins.

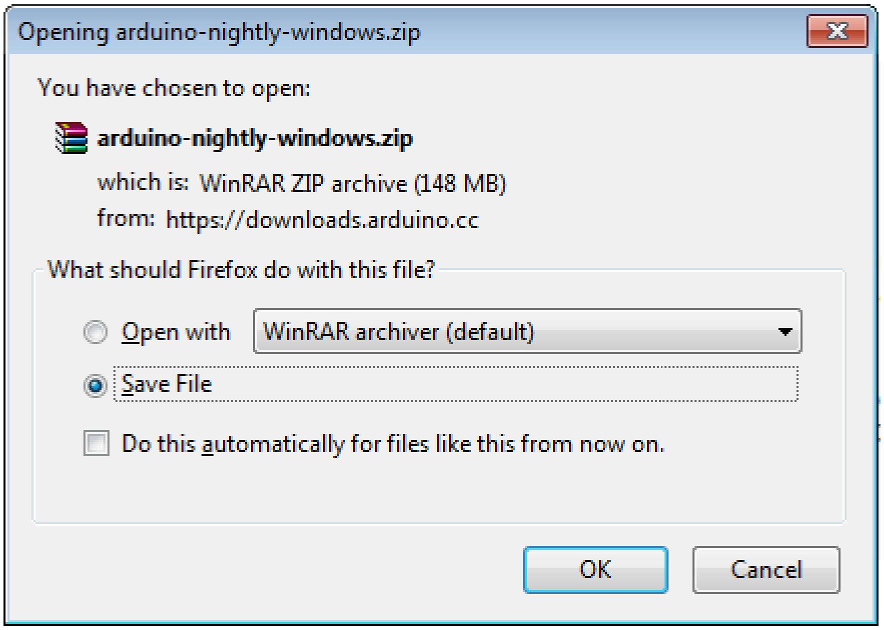
3.DOWNLOAD ARDUINO IDE SOFTWARE.

Step 1 − First you must have your Arduino board (you can choose your favorite board) and a USB cable. In case you use Arduino UNO, Arduino Duemilanove, Nano, Arduino Mega 2560, or Diecimila, you will need a standard USB cable (A plug to B plug), the kind you would connect to a USB printer as shown in the following image.

In case you use Arduino Nano, you will need an A to Mini-B cable instead as shown in the following image.

Step 2 − Download Arduino IDE Software.

You can get different versions of Arduino IDE from the Download page on the Arduino Official website. You must select your software, which is compatible with your operating system (Windows, IOS, or Linux). After your file download is complete, unzip the file.



Opening arduino-nightly-windows.zip

You have chosen to open:

arduino-nightly-windows.zip

which is: WinRAR ZIP archive (148 MB)

from: https://downloads.arduino.cc

What should Firefox do with this file?

○ Open with WinRAR archiver (default)

● Save File

□ Do this automatically for files like this from now on.

Step 3 − Power up your board.

The Arduino Uno, Mega, Duemilanove and Arduino Nano automatically draw power from either, the USB connection to the computer or an external power supply. If you are using an Arduino Diecimila, you have to make sure that the board is configured to draw power from the USB connection. The power source is selected with a jumper, a small piece of plastic that fits onto two of the three pins between the USB and power jacks. Check that it is on the two pins closest to the USB port.

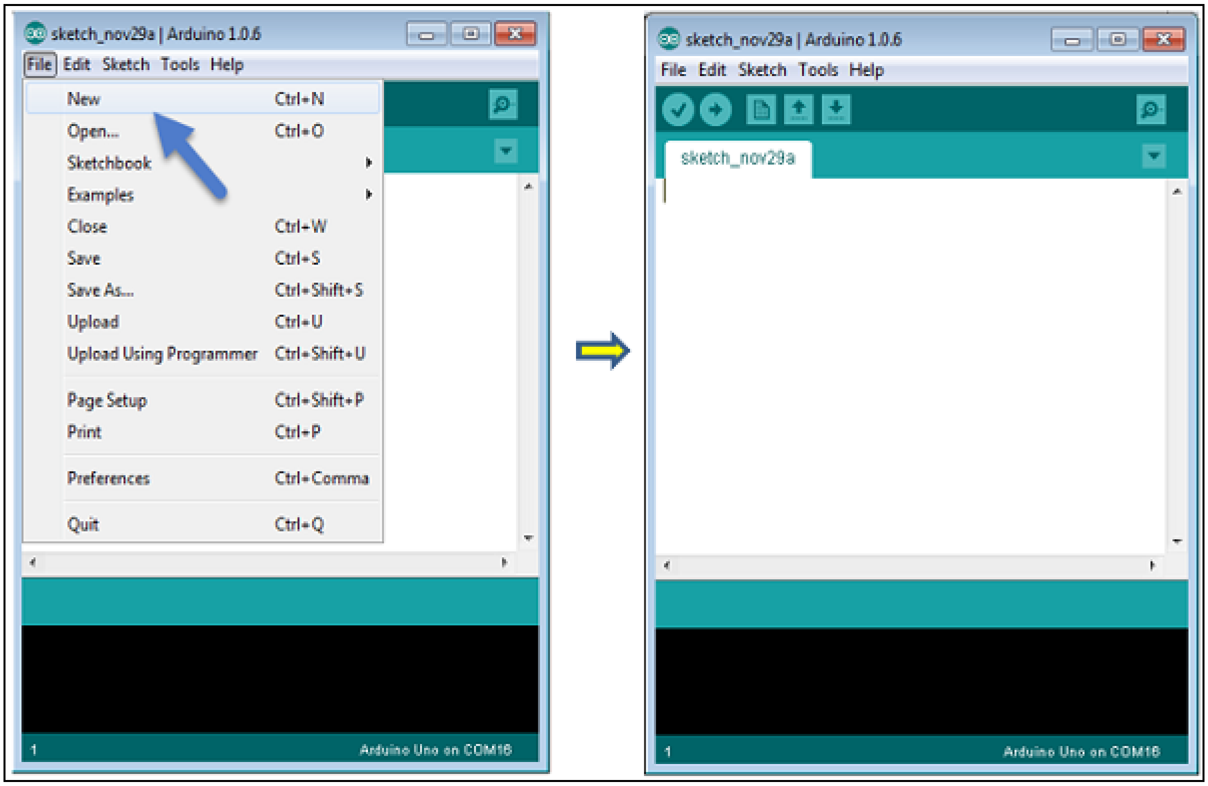
Connect the Arduino board to your computer using the USB cable. The green power LED (labeled PWR) should glow.

Step 5 − Open your first project.

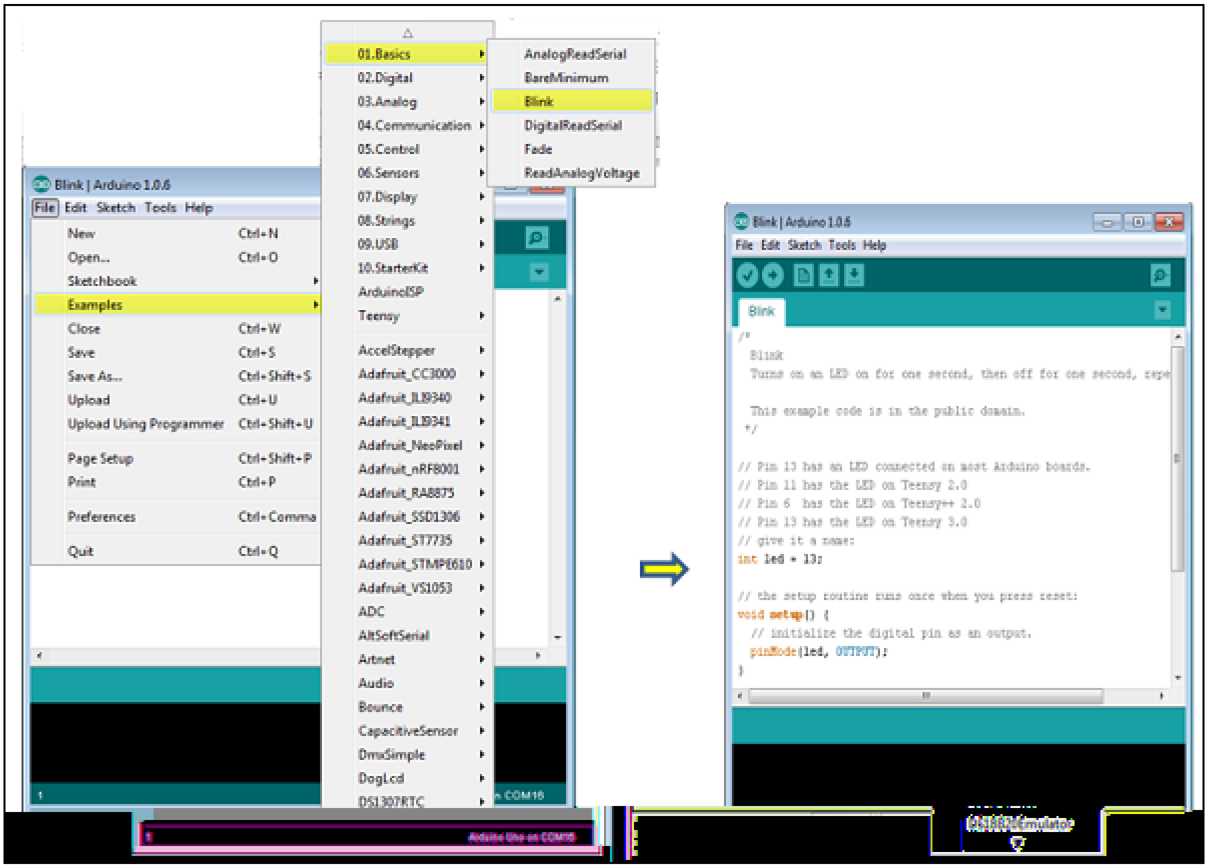
Once the software starts, you have two options −

- Create a new project.
- Open an existing project example.

To create a new project, select File → **New**.



To open an existing project example, select File → Example → Basics → Blink.



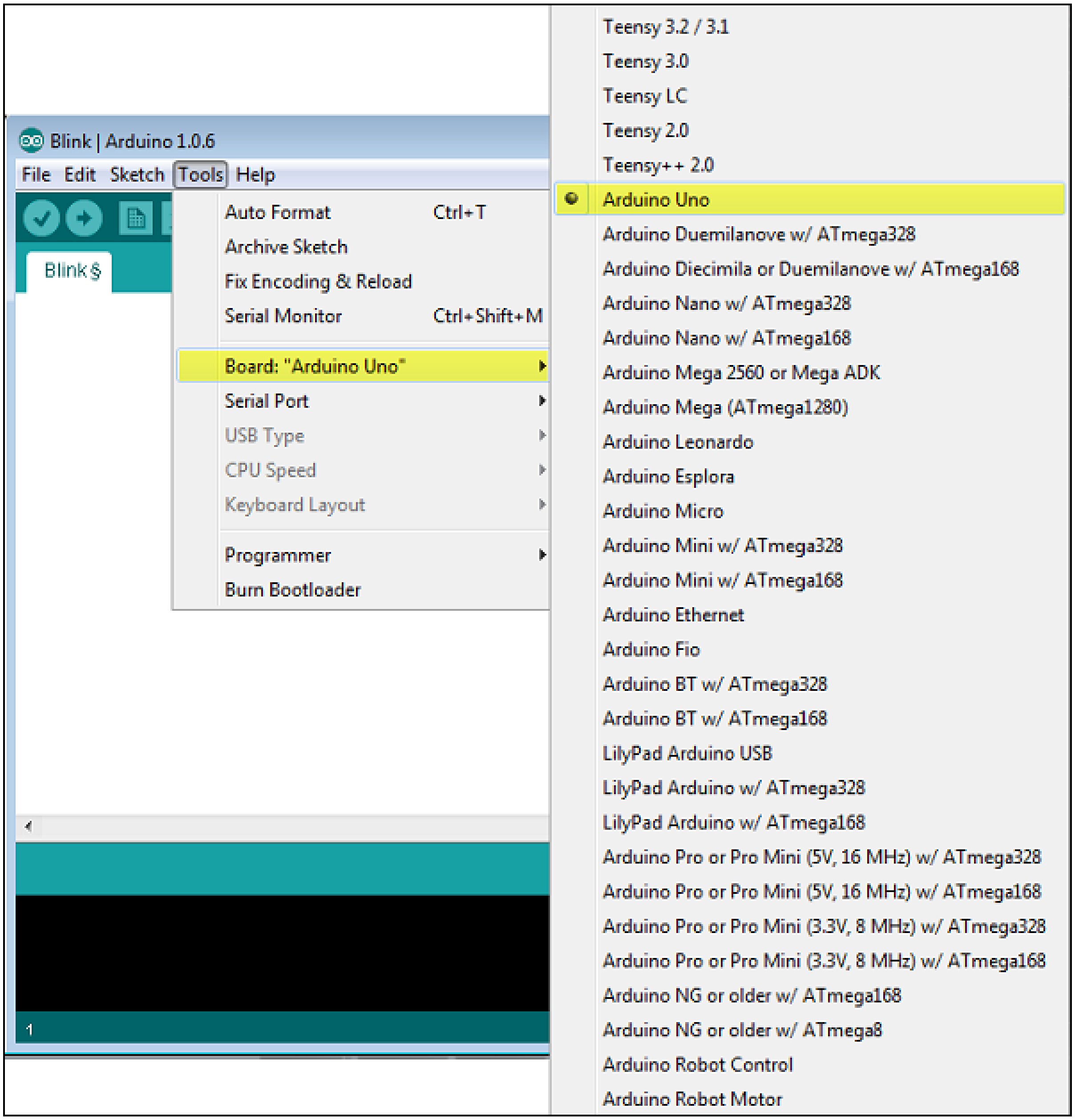
```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repe

  This example code is in the public domain.
 */

// Pin 13 has an LED connected on most Arduino boards.
// Pin 11 has the LED on Teensy 2.0
// Pin 6  has the LED on Teensy++ 2.0
// Pin 13 has the LED on Teensy 3.0
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}
```

Here, we are selecting just one of the examples with the name **Blink**. It turns the LED on and off with some time delay. You can select any other example from the list.

Step 6 − Select your Arduino board.

To avoid any error while uploading your program to the board, you must select the correct Arduino board name, which matches with the board connected to your computer.

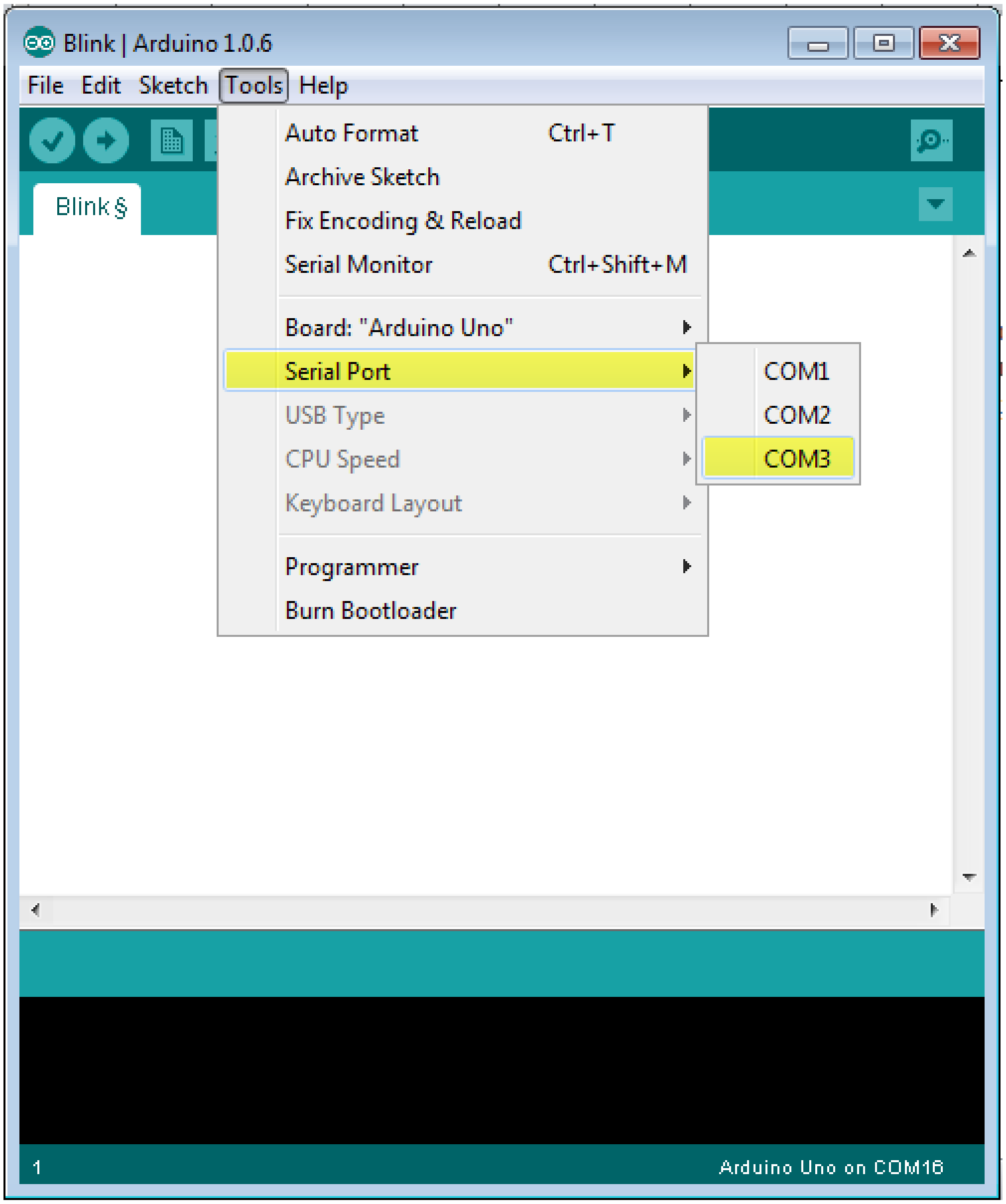
Go to Tools → Board and select your board.


Blink | Arduino 1.0.6
File Edit Sketch Tools Help
Blink §
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Board: "Arduino Uno"
Serial Port
USB Type
CPU Speed
Keyboard Layout
Programmer
Burn Bootloader
1
Teensy 3.2 / 3.1
Teensy 3.0
Teensy LC
Teensy 2.0
Teensy++ 2.0
● Arduino Uno
Arduino Duemilanove w/ ATmega328
Arduino Diecimila or Duemilanove w/ ATmega168
Arduino Nano w/ ATmega328
Arduino Nano w/ ATmega168
Arduino Mega 2560 or Mega ADK
Arduino Mega (ATmega1280)
Arduino Leonardo
Arduino Esplora
Arduino Micro
Arduino Mini w/ ATmega328
Arduino Mini w/ ATmega168
Arduino Ethernet
Arduino Fio
Arduino BT w/ ATmega328
Arduino BT w/ ATmega168
LilyPad Arduino USB
LilyPad Arduino w/ ATmega328
LilyPad Arduino w/ ATmega168
Arduino Pro or Pro Mini (5V, 16 MHz) w/ ATmega328
Arduino Pro or Pro Mini (5V, 16 MHz) w/ ATmega168
Arduino Pro or Pro Mini (3.3V, 8 MHz) w/ ATmega328
Arduino Pro or Pro Mini (3.3V, 8 MHz) w/ ATmega168
Arduino NG or older w/ ATmega168
Arduino NG or older w/ ATmega8
Arduino Robot Control
Arduino Robot Motor

Here, we have selected Arduino Uno board according to our tutorial, but you must select the name matching the board that you are using.

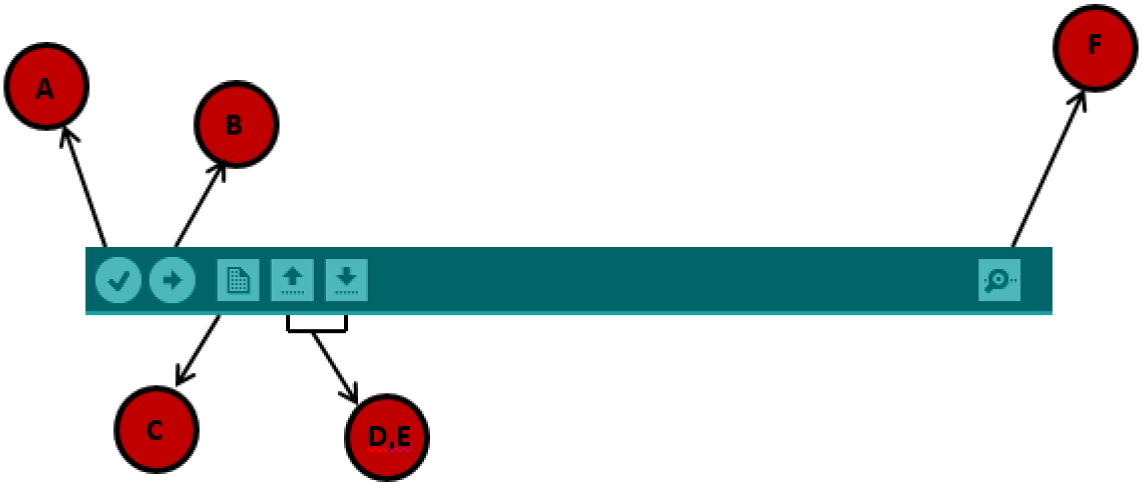
Step 7 − Select your serial port.

Select the serial device of the Arduino board. Go to **Tools** → **Serial Port** menu. This is likely to be COM3 or higher (COM1 and COM2 are usually reserved for hardware serial ports). To find out, you can disconnect your Arduino board and re-open the menu, the entry that disappears should be of the Arduino board. Reconnect the board and select that serial port.



Step 8 − Upload the program to your board.

Before explaining how we can upload our program to the board, we must demonstrate the function of each symbol appearing in the Arduino IDE toolbar.



A − Used to check if there is any compilation error.

B − Used to upload a program to the Arduino board.

C − Shortcut used to create a new sketch.

D − Used to directly open one of the example sketch.

E − Used to save your sketch.

F − Serial monitor used to receive serial data from the board and send the serial data to the board.

Now, simply click the "Upload" button in the environment. Wait a few seconds; you will see the RX and TX LEDs on the board, flashing. If the upload is successful, the message "Done uploading" will appear in the status bar.

Note − If you have an Arduino Mini, NG, or other board, you need to press the reset button physically on the board, immediately before clicking the upload button on the Arduino Software.

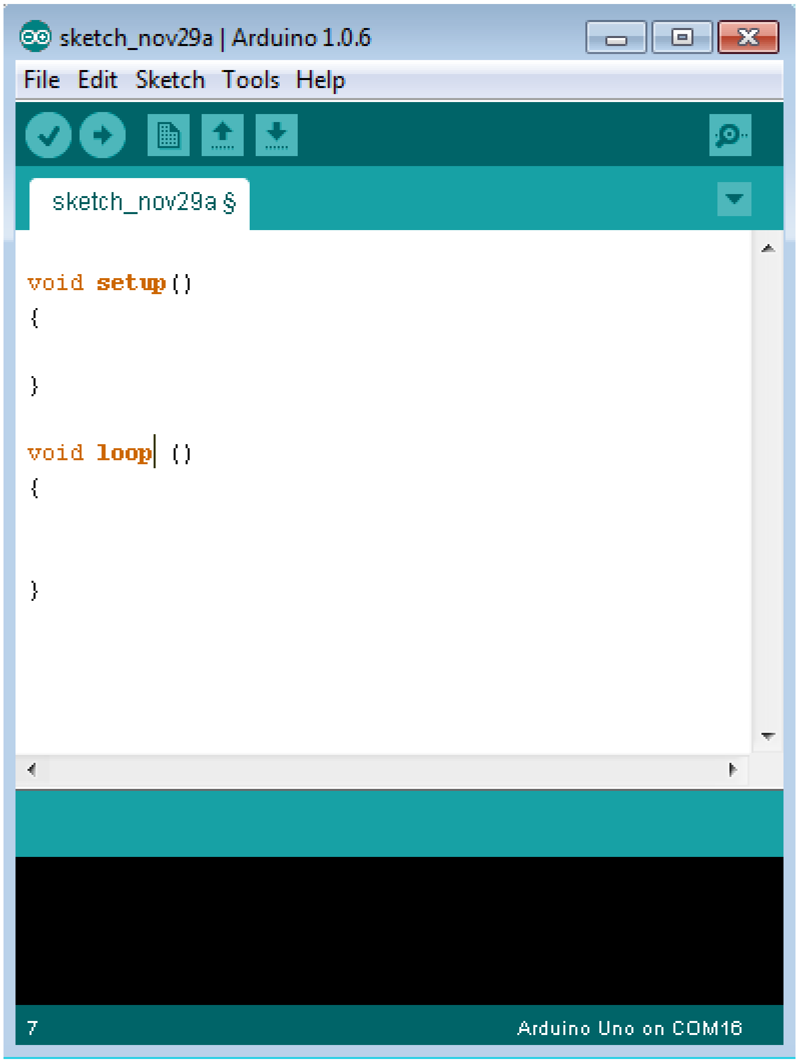
4.ARDUINO - PROGRAM STRUCTURE

Structure

Arduino programs can be divided in three main parts: **Structure, Values** (variables and constants), and **Functions**. In this tutorial, we will learn about the Arduino software program, step by step, and how we can write the program without any syntax or compilation error.

Let us start with the **Structure**. Software structure consist of two main functions –

- Setup( ) function
- Loop( ) function



```
void setup()
{

}

void loop ()
{


}
```

```
Void setup ( ) {

}
```

- **PURPOSE −** The **setup()** function is called when a sketch starts. Use it to initialize the variables, pin modes, start using libraries, etc. The setup function will only run once, after each power up or reset of the Arduino board.
- **INPUT −** -
- **OUTPUT −** -
- **RETURN −** -

```
Void Loop ( ) {

}
```

- **PURPOSE −** After creating a **setup()** function, which initializes and sets the initial values, the **loop()** function does precisely what its name suggests, and loops consecutively, allowing your program to change and respond. Use it to actively control the Arduino board.
- **INPUT −** -
- **OUTPUT −** -
- **RETURN −** -

5.ARDUINO - DATA TYPES

Data types in C refers to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in the storage and how the bit pattern stored is interpreted.

The following table provides all the data types that you will use during Arduino programming.

Void	Boolean	char	Unsigned char	byte	Int	Unsigned int	word
Long	Unsigned long	short	float	double	Array	String-char array	String-object

void

The void keyword is used only in function declarations. It indicates that the function is expected to return no information to the function from which it was called.

Example

```
Void Loop ( ) {
  // rest of the code
}
```

Boolean

A Boolean holds one of two values, true or false. Each Boolean variable occupies one byte of memory.

Example

```
boolean val = false ; // declaration of variable with type boolean and initialize it with false
boolean state = true ; // declaration of variable with type boolean and initialize it with true
```

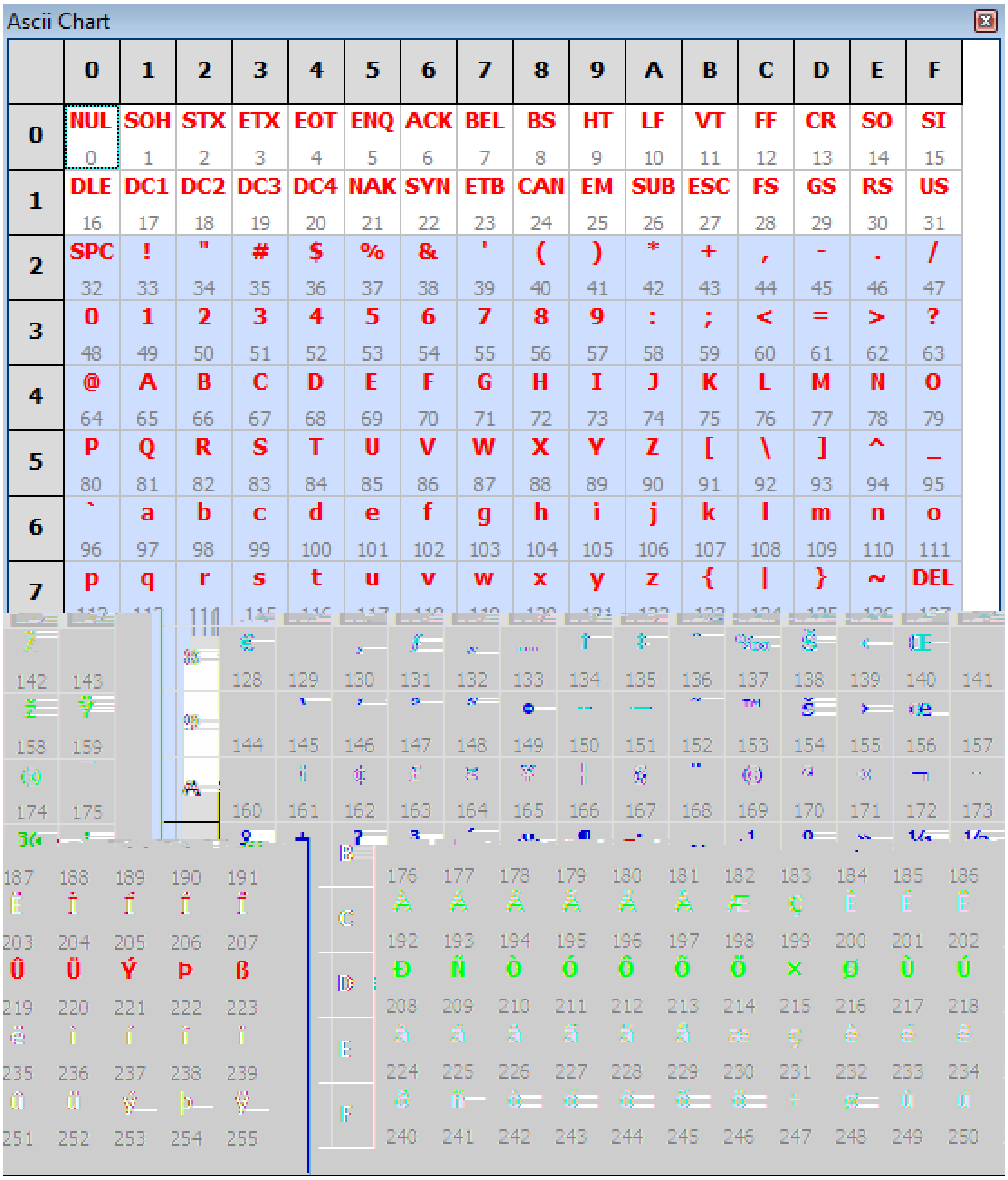
Char

A data type that takes up one byte of memory that stores a character value. Character literals are written in single quotes like this: 'A' and for multiple characters, strings use double quotes: "ABC".

However, characters are stored as numbers. You can see the specific encoding in the ASCII chart. This means that it is possible to do arithmetic operations on characters, in which the ASCII value of the character is used. For example, 'A' + 1 has the value 66, since the ASCII value of the capital letter A is 65.

Example

```
Char chr_a = ‘a’ ;//declaration of variable with type char and initialize it with character a
Char chr_c = 97 ;//declaration of variable with type char and initialize it with character 97
```


Ascii Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL 0	SOH 1	STX 2	ETX 3	EOT 4	ENQ 5	ACK 6	BEL 7	BS 8	HT 9	LF 10	VT 11	FF 12	CR 13	SO 14	SI 15
1	DLE 16	DC1 17	DC2 18	DC3 19	DC4 20	NAK 21	SYN 22	ETB 23	CAN 24	EM 25	SUB 26	ESC 27	FS 28	GS 29	RS 30	US 31
2	SPC 32	! 33	" 34	# 35	$ 36	% 37	& 38	' 39	(40	) 41	* 42	+ 43	, 44	- 45	. 46	/ 47
3	0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
4	@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
5	P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	_ 95
6	` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
7	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	| 124	} 125	~ 126	DEL 127

Ž 142	143	8	€ 128	129	‚ 130	ƒ 131	„ 132	… 133	† 134	‡ 135	ˆ 136	‰ 137	Š 138	‹ 139	Œ 140	141
ž 158	Ÿ 159	9	144	' 145	' 146	" 147	" 148	• 149	– 150	— 151	˜ 152	™ 153	š 154	› 155	œ 156	157
® 174	175	A	160	¡ 161	¢ 162	£ 163	¤ 164	¥ 165	¦ 166	§ 167	¨ 168	© 169	ª 170	« 171	¬ 172	173
187	188	189	190	191	B	176	177	178	179	180	181	182	183	184	185	186
Ë	Ì	Í	Î	Ï		À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê
203	204	205	206	207	C	192	193	194	195	196	197	198	199	200	201	202
Û	Ü	Ý	Þ	ß		Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú
219	220	221	222	223	D	208	209	210	211	212	213	214	215	216	217	218
ë	ì	í	î	ï		à	á	â	ã	ä	å	æ	ç	è	é	ê
235	236	237	238	239	E	224	225	226	227	228	229	230	231	232	233	234
û	ü	ý	þ	ÿ		ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú
251	252	253	254	255	F	240	241	242	243	244	245	246	247	248	249	250

unsigned char

Unsigned char is an unsigned data type that occupies one byte of memory. The unsigned char data type encodes numbers from 0 to 255.

Example

Unsigned Char chr_y = 121 ; // declaration of variable with type Unsigned char and initialize it with character y

byte

A byte stores an 8-bit unsigned number, from 0 to 255.

Example

byte m = 25 ;//declaration of variable with type byte and initialize it with 25

int

Integers are the primary data-type for number storage. int stores a 16-bit (2-byte) value. This yields a range of -32,768 to 32,767 (minimum value of -2^15 and a maximum value of (2^15) - 1).

The **int** size varies from board to board. On the Arduino Due, for example, an **int** stores a 32-bit (4-byte) value. This yields a range of -2,147,483,648 to 2,147,483,647 (minimum value of -2^31 and a maximum value of (2^31) - 1).

Example

int counter = 32 ;// declaration of variable with type int and initialize it with 32

Unsigned int

Unsigned ints (unsigned integers) are the same as int in the way that they store a 2 byte value. Instead of storing negative numbers, however, they only store positive values, yielding a useful range of 0 to 65,535 (2^16) - 1). The Due stores a 4 byte (32-bit) value, ranging from 0 to 4,294,967,295 (2^32 - 1).

Example

Unsigned int counter = 60 ; // declaration of variable with
type unsigned int and initialize it with 60

Word

On the Uno and other ATMEGA based boards, a word stores a 16-bit unsigned number. On the Due and Zero, it stores a 32-bit unsigned number.

Example

word w = 1000 ;//declaration of variable with type word and initialize it with 1000

Long

Long variables are extended size variables for number storage, and store 32 bits (4 bytes), from -2,147,483,648 to 2,147,483,647.

Example

Long velocity = 102346 ;//declaration of variable with type Long and initialize it with 102346

unsigned long

Unsigned long variables are extended size variables for number storage and store 32 bits (4 bytes). Unlike standard longs, unsigned longs will not store negative numbers, making their range from 0 to 4,294,967,295 (2^32 - 1).

Example

Unsigned Long velocity = 101006 ;// declaration of variable with

short

A short is a 16-bit data-type. On all Arduinos (ATMega and ARM based), a short stores a 16-bit (2-byte) value. This yields a range of -32,768 to 32,767 (minimum value of -2^15 and a maximum value of (2^15) - 1).

Example

short val = 13 ;//declaration of variable with type short and initialize it with 13

float

Data type for floating-point number is a number that has a decimal point. Floating-point numbers are often used to approximate the analog and continuous values because they have greater resolution than integers.

Floating-point numbers can be as large as 3.4028235E+38 and as low as -3.4028235E+38. They are stored as 32 bits (4 bytes) of information.

Example

float num = 1.352;//declaration of variable with type float and initialize it with 1.352

double

On the Uno and other ATMEGA based boards, Double precision floating-point number occupies four bytes. That is, the double implementation is exactly the same as the float, with no gain in precision. On the Arduino Due, doubles have 8-byte (64 bit) precision.

Example

double num = 45.352 ;// declaration of variable with type double and initialize it with 45.352

6.ARDUINO - VARIABLES & CONSTANTS

What is Variable Scope?

Variables in C programming language, which Arduino uses, have a property called scope. A scope is a region of the program and there are three places where variables can be declared. They are −

- Inside a function or a block, which is called **local variables**.
- In the definition of function parameters, which is called **formal parameters**.
- Outside of all functions, which is called **global variables**.

Local Variables

Variables that are declared inside a function or block are local variables. They can be used only by the statements that are inside that function or block of code. Local variables are not known to function outside their own. Following is the example using local variables −

```
Void setup () {

}

Void loop () {
   int x , y ;
   int z ; Local variable declaration
   x = 0;
   y = 0; actual initialization
   z = 10;
}
```

Global Variables

Global variables are defined outside of all the functions, usually at the top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

The following example uses global and local variables –

```
Int T , S ;
float c = 0 ; Global variable declaration

Void setup () {

}

Void loop () {
   int x , y ;
   int z ; Local variable declaration
   x = 0;
   y = 0; actual initialization
   z = 10;
}
```

7.ARDUINO - OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators −

- Arithmetic Operators
- Comparison Operators
- Boolean Operators
- Bitwise Operators
- Compound Operators

Arithmetic Operators

Assume variable A holds 10 and variable B holds 20 then −

Show Example

Operator name	Operator simple	Description
assignment operator	=	Stores the value to the right of the equal sign in the variable to the left of the equal sign.
addition	+	Adds two operands
subtraction	-	Subtracts second operand from the first
multiplication	*	Multiply both operands
division	/	Divide numerator by denominator
modulo	%	Modulus Operator and remainder of after an integer division

Comparison Operators

Assume variable A holds 10 and variable B holds 20 then −

Show Example

Operator name	Operator simple	Description	Example
equal to	==	Checks if the value of two operands is equal or not, if yes then condition becomes true.	(A == B) is not true
not equal to	!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.	(A != B) is true
less than	<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true
greater than	>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true
less than or equal to	<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true
greater than or equal to	>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true

Boolean Operators

Assume variable A holds 10 and variable B holds 20 then −

Show Example

Operator name	Operator simple	Description	Example
and	&&	Called Logical AND operator. If both the operands are non-zero then then condition becomes true.	(A && B) is true

or	||	Called Logical OR Operator. If any of the two operands is non-zero then then condition becomes true.	(A || B) is true
not	!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is false

Bitwise Operators

Assume variable A holds 60 and variable B holds 13 then −

Show Example

Operator name	Operator simple	Description	Example
and	&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
or	|	Binary OR Operator copies a bit if it exists in either operand	(A | B) will give 61 which is 0011 1101
xor	^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
not	~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
shift left	<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
shift right	>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Compound Operators

Assume variable A holds 10 and variable B holds 20 then −

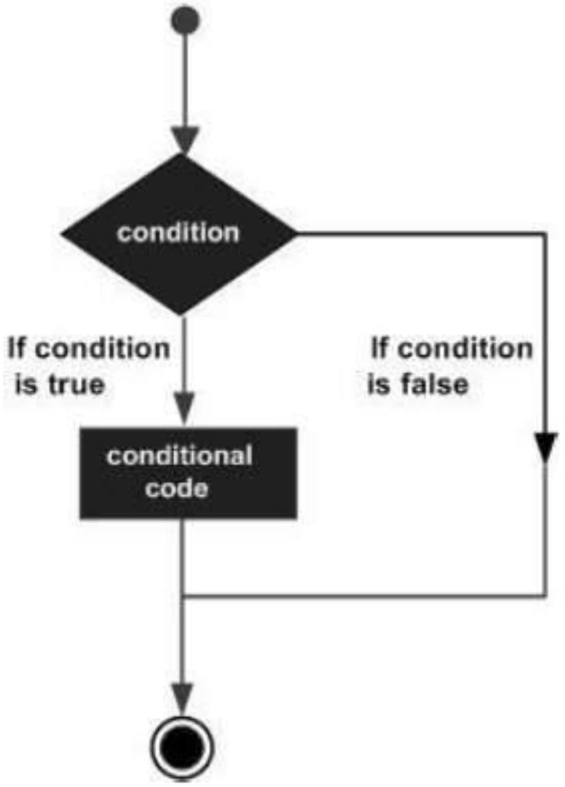
Show Example

Operator name	Operator simple	Description	Example
increment	++	Increment operator, increases integer value by one	A++ will give 11
decrement	--	Decrement operator, decreases integer value by one	A-- will give 9
compound addition	+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand	B += A is equivalent to B = B+ A
compound subtraction	-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand	B -= A is equivalent to B = B - A
compound multiplication	*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand	B*= A is equivalent to B = B* A
compound division	/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand	B /= A is equivalent to B = B / A
compound modulo	%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand	B %= A is equivalent to B = B % A
compound bitwise or	|=	bitwise inclusive OR and assignment operator	A |= 2 is same as A = A | 2

8.ARDUINO - CONTROL STATEMENTS

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program. It should be along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −



Control Statements are elements in Source Code that control the flow of program execution. They are −

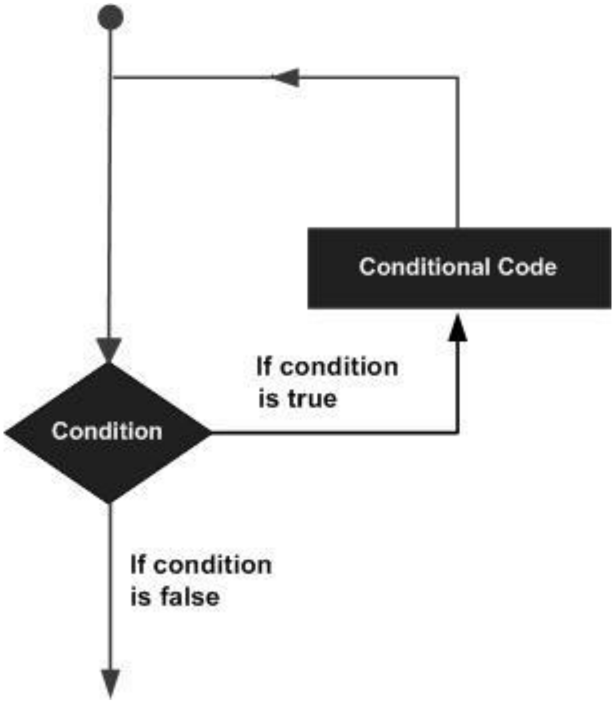
S.NO.	Control Statement & Description
1	If statement It takes an expression in parenthesis and a statement or block of statements. If the expression is true then the statement or block of statements gets executed otherwise these statements are skipped.

2	If …else statement An **if** statement can be followed by an optional else statement, which executes when the expression is false.
3	If…else if …else statement The **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.
4	switch case statement Similar to the if statements, **switch...case** controls the flow of programs by allowing the programmers to specify different codes that should be executed in various conditions.
5	Conditional Operator ? : The conditional operator ? : is the only ternary operator in C.

9.ARDUINO - LOOPS

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −



C programming language provides the following types of loops to handle looping requirements.

S.NO.	Loop & Description
1	while loop 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.
2	do…while loop The **do…while** loop is similar to the while loop. In the while loop, the loop-continuation condition is tested at the beginning of the loop before performed the body of the loop.
3	for loop A **for loop** executes statements a predetermined number of times. The control expression for the loop is initialized, tested and manipulated entirely within the for loop parentheses.
4	Nested Loop

	C language allows you to use one loop inside another loop. The following example illustrates the concept.
5	Infinite loop It is the loop having no terminating condition, so the loop becomes infinite.

10.ARDUINO – FUNCTIONS

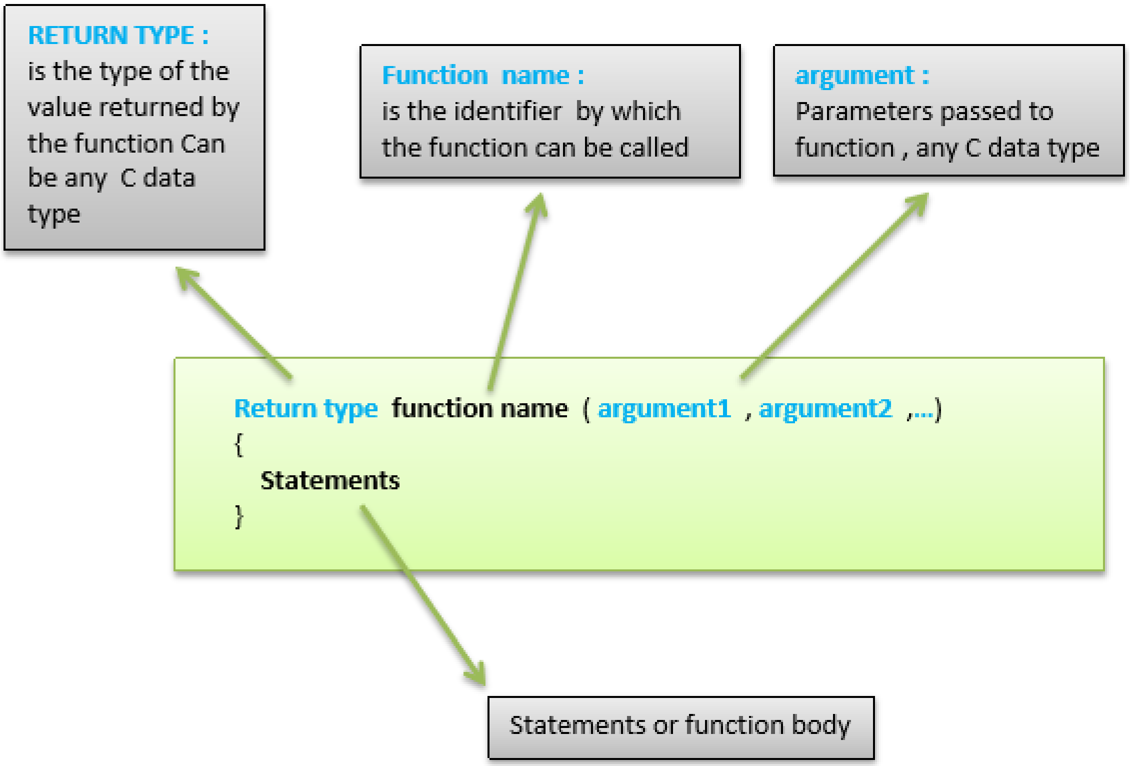
Functions allow structuring the programs in segments of code to perform individual tasks. The typical case for creating a function is when one needs to perform the same action multiple times in a program.

Standardizing code fragments into functions has several advantages −

- Functions help the programmer stay organized. Often this helps to conceptualize the program.
- Functions codify one action in one place so that the function only has to be thought about and debugged once.
- This also reduces chances for errors in modification, if the code needs to be changed.
- Functions make the whole sketch smaller and more compact because sections of code are reused many times.
- They make it easier to reuse code in other programs by making it modular, and using functions often makes the code more readable.

There are two required functions in an Arduino sketch or a program i.e. setup () and loop(). Other functions must be created outside the brackets of these two functions.

The most common syntax to define a function is −



RETURN TYPE : is the type of the value returned by the function Can be any C data type

Function name : is the identifier by which the function can be called

argument : Parameters passed to function , any C data type

```
Return type  function name  ( argument1  , argument2  ,...)
{
   Statements
}
```

Statements or function body

Function Declaration

A function is declared outside any other functions, above or below the loop function.

We can declare the function in two different ways −

The first way is just writing the part of the function called **a function prototype** above the loop function, which consists of −

- Function return type
- Function name
- Function argument type, no need to write the argument name

Function prototype must be followed by a semicolon (;).

The following example shows the demonstration of the function declaration using the first method.

Example

```
int sum_func (int x, int y) // function declaration {
   int z = 0;
   z = x+y ;
   return z; // return the value
}

void setup () {
   Statements // group of statements
}

Void loop () {
   int result = 0 ;
   result = Sum_func (5,6) ; // function call
}
```

The second part, which is called the function definition or declaration, must be declared below the loop function, which consists of −

- Function return type
- Function name
- Function argument type, here you must add the argument name
- The function body (statements inside the function executing when the function is called)

The following example demonstrates the declaration of function using the second method.

Example

```
int sum_func (int , int ) ; // function prototype

void setup () {
   Statements // group of statements
}

Void loop () {
   int result = 0 ;
   result = Sum_func (5,6) ; // function call
}

int sum_func (int x, int y) // function declaration {
   int z = 0;
   z = x+y ;
   return z; // return the value
}
```

11.ARDUINO - STRINGS

Strings are used to store text. They can be used to display text on an LCD or in the Arduino IDE Serial Monitor window. Strings are also useful for storing the user input. For example, the characters that a user types on a keypad connected to the Arduino.

There are two types of strings in Arduino programming −

- Arrays of characters, which are the same as the strings used in C programming.
- The Arduino String, which lets us use a string object in a sketch.

In this chapter, we will learn Strings, objects and the use of strings in Arduino sketches. By the end of the chapter, you will learn which type of string to use in a sketch.

String Character Arrays

The first type of string that we will learn is the string that is a series of characters of the type **char**. In the previous chapter, we learned what an array is; a consecutive series of the same type of variable stored in memory. A string is an array of char variables.

String Character Array Example

This example will show how to make a string and print it to the serial monitor window.

Example

```
void setup() {
  char my_str[6]; // an array big enough for a 5 character string
  Serial.begin(9600);
  my_str[0] = 'H'; // the string consists of 5 characters
  my_str[1] = 'e';
  my_str[2] = 'l';
  my_str[3] = 'l';
  my_str[4] = 'o';
  my_str[5] = 0; // 6th array element is a null terminator
  Serial.println(my_str);
}

void loop() {

}
```

The following example shows what a string is made up of; a character array with printable characters and 0 as the last element of the array to show that this is where the string ends. The string can be printed out to the Arduino IDE Serial Monitor window by using **Serial.println()** and passing the name of the string.

This same example can be written in a more convenient way as shown below −

Example

```
void setup() {
  char my_str[] = "Hello";
  Serial.begin(9600);
  Serial.println(my_str);
}

void loop() {

}
```

Manipulating String Arrays

We can alter a string array within a sketch as shown in the following sketch.

Example

```
void setup() {
  char like[] = "I like coffee and cake"; // create a string
  Serial.begin(9600);
  // (1) print the string
  Serial.println(like);
  // (2) delete part of the string
  like[13] = 0;
  Serial.println(like);
  // (3) substitute a word into the string
  like[13] = ' '; // replace the null terminator with a space
  like[18] = 't'; // insert the new word
  like[19] = 'e';
  like[20] = 'a';
  like[21] = 0; // terminate the string
  Serial.println(like);
}

void loop() {

}
```

Result

```
I like coffee and cake
I like coffee
I like coffee and tea
```

The sketch works in the following way.

Creating and Printing the String

In the sketch given above, a new string is created and then printed for display in the Serial Monitor window.

Shortening the String

The string is shortened by replacing the 14th character in the string with a null terminating zero (2). This is element number 13 in the string array counting from 0.

When the string is printed, all the characters are printed up to the new null terminating zero. The other characters do not disappear; they still exist in the memory and the string array is still the same size. The only difference is that any function that works with strings will only see the string up to the first null terminator.

Changing a Word in the String

Finally, the sketch replaces the word "cake" with "tea" (3). It first has to replace the null terminator at like[13] with a space so that the string is restored to the originally created format.

Functions to Manipulate String Arrays

The previous sketch manipulated the string in a manual way by accessing individual characters in the string. To make it easier to manipulate string arrays, you can write your own functions to do so, or use some of the string functions from the **C** language library.

Given below is the list Functions to Manipulate String Arrays

The next sketch uses some C string functions.

Example

```
void setup() {
  char str[] = "This is my string"; // create a string
  char out_str[40]; // output from string functions placed here
  int num; // general purpose integer
  Serial.begin(9600);

  // (1) print the string
  Serial.println(str);

  // (2) get the length of the string (excludes null terminator)
  num = strlen(str);
  Serial.print("String length is: ");
  Serial.println(num);

  // (3) get the length of the array (includes null terminator)
  num = sizeof(str); // sizeof() is not a C string function
  Serial.print("Size of the array: ");
  Serial.println(num);

  // (4) copy a string
  strcpy(out_str, str);
  Serial.println(out_str);

  // (5) add a string to the end of a string (append)
  strcat(out_str, " sketch.");
  Serial.println(out_str);
  num = strlen(out_str);
  Serial.print("String length is: ");
  Serial.println(num);
  num = sizeof(out_str);
  Serial.print("Size of the array out_str[]: ");
  Serial.println(num);
}

void loop() {

}
```

Result

```
This is my string
String length is: 17
Size of the array: 18
This is my string
This is my string sketch.
String length is: 25
Size of the array out_str[]: 40
```

The sketch works in the following way.

Print the String

The newly created string is printed to the Serial Monitor window as done in previous sketches.

Get the Length of the String

The strlen() function is used to get the length of the string. The length of the string is for the printable characters only and does not include the null terminator.

The string contains 17 characters, so we see 17 printed in the Serial Monitor window.

Get the Length of the Array

The operator sizeof() is used to get the length of the array that contains the string. The length includes the null terminator, so the length is one more than the length of the string.

sizeof() looks like a function, but technically is an operator. It is not a part of the C string library, but was used in the sketch to show the difference between the size of the array and the size of the string (or string length).

Copy a String

The strcpy() function is used to copy the str[] string to the out_num[] array. The strcpy() function copies the second string passed to it into the first string. A copy of the string now exists in the out_num[] array, but only takes up 18 elements of the array, so we still have 22 free char elements in the array. These free elements are found after the string in memory.

The string was copied to the array so that we would have some extra space in the array to use in the next part of the sketch, which is adding a string to the end of a string.

Append a String to a String (Concatenate)

The sketch joins one string to another, which is known as concatenation. This is done using the strcat() function. The strcat() function puts the second string passed to it onto the end of the first string passed to it.

After concatenation, the length of the string is printed to show the new string length. The length of the array is then printed to show that we have a 25-character long string in a 40 element long array.

Remember that the 25-character long string actually takes up 26 characters of the array because of the null terminating zero.

Array Bounds

When working with strings and arrays, it is very important to work within the bounds of strings or arrays. In the example sketch, an array was created, which was 40 characters long, in order to allocate the memory that could be used to manipulate strings.

If the array was made too small and we tried to copy a string that is bigger than the array to it, the string would be copied over the end of the array. The memory beyond the end of the array could contain other important data used in the sketch, which would then be overwritten by our string. If the memory beyond the end of the string is overrun, it could crash the sketch or cause unexpected behavior.

ARDUINO - STRING OBJECT

What is an Object?

An object is a construct that contains both data and functions. A String object can be created just like a variable and assigned a value or string. The String object contains functions (which are called "methods" in object oriented programming (OOP)) which operate on the string data contained in the String object.

The following sketch and explanation will make it clear what an object is and how the String object is used.

Example

```
void setup() {
  String my_str = "This is my string.";
  Serial.begin(9600);

  // (1) print the string
  Serial.println(my_str);

  // (2) change the string to upper-case
  my_str.toUpperCase();
  Serial.println(my_str);

  // (3) overwrite the string
  my_str = "My new string.";
  Serial.println(my_str);

  // (4) replace a word in the string
  my_str.replace("string", "Arduino sketch");
  Serial.println(my_str);
  // (5) get the length of the string
  Serial.print("String length is: ");
  Serial.println(my_str.length());
}

void loop() {

}
```

Result

```
This is my string.
THIS IS MY STRING.
My new string.
My new Arduino sketch.
String length is: 22
```

A string object is created and assigned a value (or string) at the top of the sketch.

```
String my_str = "This is my string." ;
```

This creates a String object with the name **my_str** and gives it a value of "This is my string.".

This can be compared to creating a variable and assigning a value to it such as an integer –

```
int my_var = 102;
```

The sketch works in the following way.

Printing the String

The string can be printed to the Serial Monitor window just like a character array string.

Convert the String to Upper-case

The string object my_str that was created, has a number of functions or methods that can be operated on it. These methods are invoked by using the objects name followed by the dot operator (.) and then the name of the function to use.

```
my_str.toUpperCase();
```

The **toUpperCase()** function operates on the string contained in the **my_str** object which is of type String and converts the string data (or text) that the object contains to upper-case characters. A list of the functions that the String class contains can be found in the Arduino String reference. Technically, String is called a class and is used to create String objects.

Overwrite a String

The assignment operator is used to assign a new string to the **my_str** object that replaces the old string

```
my_str = "My new string." ;
```

Replacing a Word in the String

The replace() function is used to replace the first string passed to it by the second string passed to it. replace() is another function that is built into the String class and so is available to use on the String object my_str.

Getting the Length of the String

Getting the length of the string is easily done by using length(). In the example sketch, the result returned by length() is passed directly to Serial.println() without using an intermediate variable.

When to Use a String Object

A String object is much easier to use than a string character array. The object has built-in functions that can perform a number of operations on strings.

The main disadvantage of using the String object is that it uses a lot of memory and can quickly use up the Arduinos RAM memory, which may cause Arduino to hang, crash or behave unexpectedly. If a sketch on an Arduino is small and limits the use of objects, then there should be no problems.

Character array strings are more difficult to use and you may need to write your own functions to operate on these types of strings. The advantage is that you have control on the size of the string arrays that you make, so you can keep the arrays small to save memory.

You need to make sure that you do not write beyond the end of the array bounds with string arrays. The String object does not have this problem and will take care of the string bounds for you, provided there is enough memory for it to operate on. The String object can try to write to memory that does not exist when it runs out of memory, but will never write over the end of the string that it is operating on.

12.ARDUINO - TIME

Arduino provides four different time manipulation functions. They are −

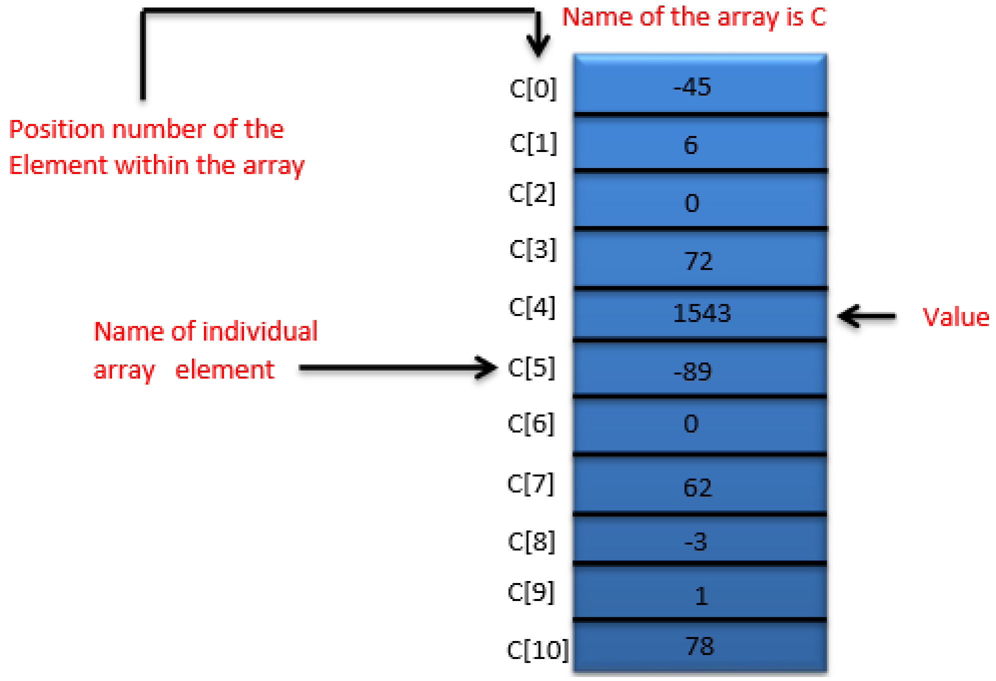
S.No.	Function & Description
1	delay () function The way the **delay()** function works is pretty simple. It accepts a single integer (or number) argument. This number represents the time (measured in milliseconds).
2	delayMicroseconds () function The **delayMicroseconds()** function accepts a single integer (or number) argument. There are a thousand microseconds in a millisecond, and a million microseconds in a second.
3	millis () function This function is used to return the number of milliseconds at the time, the Arduino board begins running the current program.
4	micros () function The micros() function returns the number of microseconds from the time, the Arduino board begins running the current program. This number overflows i.e. goes back to zero after approximately 70 minutes.

13.ARDUINO – ARRAYS

An array is a consecutive group of memory locations that are of the same type. To refer to a particular location or element in the array, we specify the name of the array and the position number of the particular element in the array.

The illustration given below shows an integer array called C that contains 11 elements. You refer to any one of these elements by giving the array name followed by the particular element's position number in square brackets ([]). The position number is more formally called a subscript or index (this number specifies the number of elements from the beginning of the array). The first element has subscript 0 (zero) and is sometimes called the zeros element.

Thus, the elements of array C are C[0] (pronounced "C sub zero"), C[1], C[2] and so on. The highest subscript in array C is 10, which is 1 less than the number of elements in the array (11). Array names follow the same conventions as other variable names.



A subscript must be an integer or integer expression (using any integral type). If a program uses an expression as a subscript, then the program evaluates the expression to determine the subscript. For example, if we assume that variable a is equal to 5 and that variable b is equal to 6, then the statement adds 2 to array element C[11].

A subscripted array name is an lvalue, it can be used on the left side of an assignment, just as non-array variable names can.

Let us examine array C in the given figure, more closely. The name of the entire array is C. Its 11 elements are referred to as C[0] to C[10]. The value of C[0] is -45, the value of C[1] is 6, the value of C[2] is 0, the value of C[7] is 62, and the value of C[10] is 78.

To print the sum of the values contained in the first three elements of array C, we would write −

```
Serial.print (C[ 0 ] + C[ 1 ] + C[ 2 ] );
```

To divide the value of C[6] by 2 and assign the result to the variable x, we would write −

```
x = C[ 6 ] / 2;
```

Declaring Arrays

Arrays occupy space in memory. To specify the type of the elements and the number of elements required by an array, use a declaration of the form −

```
type arrayName [ arraySize ] ;
```

The compiler reserves the appropriate amount of memory. (Recall that a declaration, which reserves memory is more properly known as a definition). The arraySize must be an integer constant greater than zero. For example, to tell the compiler to reserve 11 elements for integer array C, use the declaration −

```
int C[ 12 ]; // C is an array of 12 integers
```

Arrays can be declared to contain values of any non-reference data type. For example, an array of type string can be used to store character strings.

Examples Using Arrays

This section gives many examples that demonstrate how to declare, initialize and manipulate arrays.

Example 1: Declaring an Array and using a Loop to Initialize the Array’s Elements

The program declares a 10-element integer array **n**. Lines a–b use a **For** statement to initialize the array elements to zeros. Like other automatic variables, automatic arrays are not implicitly initialized to zero. The first output statement (line c) displays the column headings for the columns printed in the subsequent for statement (lines d–e), which prints the array in tabular format.

Example

```
int n[ 10 ] ; // n is an array of 10 integers

void setup () {

}

void loop () {
  for ( int i = 0; i < 10; ++i ) // initialize elements of array n to 0 {
    n[ i ] = 0; // set element at location i to 0
```

```
      Serial.print (i) ;
      Serial.print (‘\r’) ;
   }
   for ( int j = 0; j < 10; ++j ) // output each array element's value {
      Serial.print (n[j]) ;
      Serial.print (‘\r’) ;
   }
}
```

Result − It will produce the following result −

Element	Value
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Example 2: Initializing an Array in a Declaration with an Initializer List

The elements of an array can also be initialized in the array declaration by following the array name with an equal-to sign and a brace-delimited comma-separated list of initializers. The program uses an initializer list to initialize an integer array with 10 values (line a) and prints the array in tabular format (lines b–c).

Example

```
// n is an array of 10 integers
int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 } ;

void setup () {

}

void loop () {
   for ( int i = 0; i < 10; ++i ) {
      Serial.print (i) ;
      Serial.print (‘\r’) ;
   }
   for ( int j = 0; j < 10; ++j ) // output each array element's value {
      Serial.print (n[j]) ;
      Serial.print (‘\r’) ;
   }
}
```

Result − It will produce the following result −

Element	Value
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

Example 3: Summing the Elements of an Array

Often, the elements of an array represent a series of values to be used in a calculation. For example, if the elements of an array represent exam grades, a professor may wish to total the elements of the array and use that sum to calculate the class average for the exam. The program sums the values contained in the 10-element integer array **a**.

Example

```
const int arraySize = 10; // constant variable indicating size of array
int a[ arraySize ] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
int total = 0;

void setup () {

}
void loop () {
   // sum contents of array a
   for ( int i = 0; i < arraySize; ++i )
      total += a[ i ];
   Serial.print (“Total of array elements : ”) ;
   Serial.print(total) ;
}
```

Result − It will produce the following result −

```
Total of array elements: 849
```

Arrays are important to Arduino and should need a lot more attention. The following important concepts related to array should be clear to a Arduino −

S.NO.	Concept & Description
1	Passing Arrays to Functions To pass an array argument to a function, specify the name of the array without any brackets.
2	Multi-Dimensional Arrays Arrays with two dimensions (i.e., subscripts) often represent tables of values consisting of information arranged in rows and columns.

14.ARDUINO - I/O FUNCTIONS

The pins on the Arduino board can be configured as either inputs or outputs. We will explain the functioning of the pins in those modes. It is important to note that a majority of Arduino analog pins, may be configured, and used, in exactly the same manner as digital pins.

Pins Configured as INPUT

Arduino pins are by default configured as inputs, so they do not need to be explicitly declared as inputs with **pinMode()** when you are 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 megaohm in front of the pin.

This means that it takes very little current to switch the input pin from one state to another. This makes the pins useful for such tasks as implementing a capacitive touch sensor or reading an LED as a photodiode.

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

Pull-up Resistors

Pull-up resistors are often useful to steer an input pin to a known state if no input is present. This can be done by adding a pull-up resistor (to +5V), or a pull-down resistor (resistor to ground) on the input. A 10K resistor is a good value for a pull-up or pull-down resistor.

Using Built-in Pull-up Resistor with Pins Configured as Input

There are 20,000 pull-up resistors built into the Atmega chip that can be accessed from software. These built-in pull-up resistors are accessed by setting the **pinMode()** as INPUT_PULLUP. This effectively inverts the behavior of the INPUT mode, where HIGH means the sensor is OFF and LOW means the sensor is ON. The value of this pull-up depends on the microcontroller used. On most AVR-based boards, the value is guaranteed to be between 20kΩ and 50kΩ. On the Arduino Due, it is between 50kΩ and 150kΩ. For the exact value, consult the datasheet of the microcontroller on your board.

When connecting a sensor to a pin configured with INPUT_PULLUP, the other end should be connected to the ground. In case of a simple switch, this causes the pin to read HIGH when the switch is open and LOW when the switch is pressed. The pull-up resistors provide enough current to light an LED dimly connected to a pin configured as an input. If LEDs in a project seem to be working, but very dimly, this is likely what is going on.

Same registers (internal chip memory locations) that control whether a pin is HIGH or LOW control the pull-up resistors. Consequently, a pin that is configured to have pull-up resistors turned on when the pin is in INPUTmode, will have the pin configured as HIGH if the pin is then switched to an OUTPUT mode with pinMode(). This works in the other direction as well, and an output pin that is left in a HIGH state will have the pull-up resistor set if switched to an input with pinMode().

Example

```
pinMode(3,INPUT) ; // set pin to input without using built in pull up resistor
pinMode(5,INPUT_PULLUP) ; // set pin to input using built in pull up resistor
```

Pins Configured as OUTPUT

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. This is enough current to brightly light up an LED (do not forget the series resistor), or run many sensors but not enough current to run relays, solenoids, or motors.

Attempting to run high current devices from the output pins, can damage or destroy the output transistors in the pin, or damage the entire Atmega chip. Often, this results in a "dead" pin in the microcontroller but the remaining chips still function adequately. For this reason, it is a good idea to connect the OUTPUT pins to other devices through 470Ω or 1k resistors, unless maximum current drawn from the pins is required for a particular application.

pinMode() Function

The pinMode() function is used to configure a specific pin to behave either as an input or an output. It is possible to enable the internal pull-up resistors with the mode INPUT_PULLUP. Additionally, the INPUT mode explicitly disables the internal pull-ups.

pinMode() Function Syntax

```
Void setup () {
  pinMode (pin , mode);
}
```

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

Example

```
int button = 5 ; // button connected to pin 5
int LED = 6; // LED connected to pin 6

void setup () {
  pinMode(button , INPUT_PULLUP);
  // set the digital pin as input with pull-up resistor
  pinMode(button , OUTPUT); // set the digital pin as output
}

void setup () {
  If (digitalRead(button ) == LOW) // if button pressed {
    digitalWrite(LED,HIGH); // turn on led
    delay(500); // delay for 500 ms
    digitalWrite(LED,LOW); // turn off led
    delay(500); // delay for 500 ms
  }
}
```

digitalWrite() Function

The **digitalWrite()** function is used to write a HIGH or a LOW value to a digital pin. If the pin has been configured as an OUTPUT with pinMode(), its voltage will be set to the corresponding value: 5V (or 3.3V on 3.3V boards) for HIGH, 0V (ground) for LOW. If the pin is configured as an INPUT, digitalWrite() will enable (HIGH) or disable (LOW) the internal pullup on the input pin. It is recommended to set the pinMode() to INPUT_PULLUP to enable the internal pull-up resistor.

If you do not set the pinMode() to OUTPUT, and connect an LED to a pin, when calling digitalWrite(HIGH), the LED may appear dim. Without explicitly setting pinMode(), digitalWrite() will have enabled the internal pull-up resistor, which acts like a large current-limiting resistor.

digitalWrite() Function Syntax

```
Void loop() {
  digitalWrite (pin ,value);
}
```

- **pin** − the number of the pin whose mode you wish to set
- **value** − HIGH, or LOW.

Example

```
int LED = 6; // LED connected to pin 6

void setup () {
  pinMode(LED, OUTPUT); // set the digital pin as output
}

void setup () {
  digitalWrite(LED,HIGH); // turn on led
  delay(500); // delay for 500 ms
  digitalWrite(LED,LOW); // turn off led
  delay(500); // delay for 500 ms
}
```

analogRead( ) function

Arduino is able to detect whether there is a voltage applied to one of its pins and report it through the digitalRead() function. There is a difference between an on/off sensor (which detects the presence of an object) and an analog sensor, whose value continuously changes. In order to read this type of sensor, we need a different type of pin.

In the lower-right part of the Arduino board, you will see six pins marked “Analog In”. These special pins not only tell whether there is a voltage applied to them, but also its value. By using the **analogRead()** function, we can read the voltage applied to one of the pins.

This function returns a number between 0 and 1023, which represents voltages between 0 and 5 volts. For example, if there is a voltage of 2.5 V applied to pin number 0, analogRead(0) returns 512.

analogRead() function Syntax

```
analogRead(pin);
```

- **pin** − the number of the analog input pin to read from (0 to 5 on most boards, 0 to 7 on the Mini and Nano, 0 to 15 on the Mega)

Example

```
int analogPin = 3;//potentiometer wiper (middle terminal)
  // connected to analog pin 3
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
}
```

ARDUINO - ADVANCED I/O FUNCTION

In this chapter, we will learn some advanced Input and Output Functions.

analogReference() Function

Configures the reference voltage used for analog input (i.e. the value used as the top of the input range). The options are −

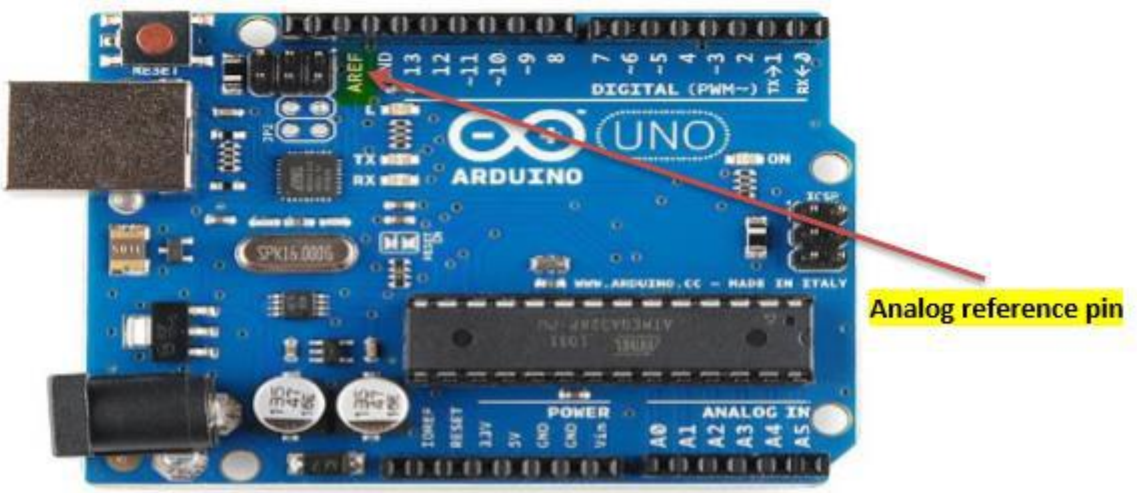
- **DEFAULT** − The default analog reference of 5 volts (on 5V Arduino boards) or 3.3 volts (on 3.3V Arduino boards)
- **INTERNAL** − An built-in reference, equal to 1.1 volts on the ATmega168 or ATmega328 and 2.56 volts on the ATmega8 (not available on the Arduino Mega)
- **INTERNAL1V1** − A built-in 1.1V reference (Arduino Mega only)
- **INTERNAL2V56** − A built-in 2.56V reference (Arduino Mega only)
- **EXTERNAL** − The voltage applied to the AREF pin (0 to 5V only) is used as the reference

analogReference() Function Syntax

analogReference (type);

type − can use any type of the follow (DEFAULT, INTERNAL, INTERNAL1V1, INTERNAL2V56, EXTERNAL)

Do not use anything less than 0V or more than 5V for external reference voltage on the AREF pin. If you are using an external reference on the AREF pin, you must set the analog reference to EXTERNAL before calling the **analogRead()** function. Otherwise, you will short the active reference voltage (internally generated) and the AREF pin, possibly damaging the microcontroller on your Arduino board.



Analog reference pin

Alternatively, you can connect the external reference voltage to the AREF pin through a 5K resistor, allowing you to switch between external and internal reference voltages.

Note that the resistor will alter the voltage that is used as the reference because there is an internal 32K resistor on the AREF pin. The two act as a voltage divider. For example, 2.5V applied through the resistor will yield 2.5 * 32 / (32 + 5) = ~2.2V at the AREF pin.

Example

```
int analogPin = 3;// potentiometer wiper (middle terminal) connected to analog pin 3
int val = 0; // variable to store the read value

void setup() {
  Serial.begin(9600); // setup serial
  analogReference(EXTERNAL); // the voltage applied to the AREF pin (0 to 5V only)
    // is used as the reference.
}

void loop() {
  val = analogRead(analogPin); // read the input pin
  Serial.println(val); // debug value
}
```

15.ARDUINO - CHARACTER FUNCTIONS

All data is entered into computers as characters, which includes letters, digits and various special symbols. In this section, we discuss the capabilities of C++ for examining and manipulating individual characters.

The character-handling library includes several functions that perform useful tests and manipulations of character data. Each function receives a character, represented as an int, or EOF as an argument. Characters are often manipulated as integers.

Remember that EOF normally has the value –1 and that some hardware architectures do not allow negative values to be stored in char variables. Therefore, the character-handling functions manipulate characters as integers.

The following table summarizes the functions of the character-handling library. When using functions from the character-handling library, include the **<cctype>** header.

S.No.	Prototype & Description
1	**int isdigit(int c)** Returns 1 if c is a digit and 0 otherwise.
2	**int isalpha(int c)** Returns 1 if c is a letter and 0 otherwise.
3	**int isalnum(int c)** Returns 1 if c is a digit or a letter and 0 otherwise.
4	**int isxdigit(int c)** Returns 1 if c is a hexadecimal digit character and 0 otherwise. (See Appendix D, Number Systems, for a detailed explanation of binary, octal, decimal and numbers.)
5	**int islower(int c)** Returns 1 if c is a lowercase letter and 0 otherwise.

6	**int isupper(int c)** Returns 1 if c is an uppercase letter; 0 otherwise.
7	**int isspace(int c)** Returns 1 if c is a white-space character—newline ('\n'), space (' '), form feed ('\f'), carriage return ('\r'), horizontal tab ('\t'), or vertical tab ('\v')—and 0 otherw
8	**int iscntrl(int c)** Returns 1 if c is a control character, such as newline ('\n'), form feed ('\f'), carriage return ('\r'), ('\t'), vertical tab ('\v'), alert ('\a'), or backspace ('\b')—and 0 otherwise.
9	**int ispunct(int c)** Returns 1 if c is a printing character other than a space, a digit, or a letter and 0 otherwise.
10	**int isprint(int c)** Returns 1 if c is a printing character including space (' ') and 0 otherwise.
11	**int isgraph(int c)** Returns 1 if c is a printing character other than space (' ') and 0 otherwise.

Examples

The following example demonstrates the use of the functions **isdigit, isalpha, isalnum** and **isxdigit**. Function **isdigit** determines whether its argument is a digit (0–9). The function **isalpha** determines whether its argument is an uppercase letter (A-Z) or a lowercase letter (a–z). The function **isalnum** determines whether its argument is an uppercase, lowercase letter or a digit. Function **isxdigit** determines whether its argument is a hexadecimal digit (A–F, a–f, 0–9).

Example 1

```
void setup () {
   Serial.begin (9600);
   Serial.print ("According to isdigit:\r");
   Serial.print (isdigit( '8' ) ? "8 is a": "8 is not a");
   Serial.print (" digit\r" );
   Serial.print (isdigit( '8' ) ?"# is a": "# is not a") ;
   Serial.print (" digit\r");
   Serial.print ("\rAccording to isalpha:\r" );
   Serial.print (isalpha('A' ) ?"A is a": "A is not a");
   Serial.print (" letter\r");
   Serial.print (isalpha('A' ) ?"b is a": "b is not a");
   Serial.print (" letter\r");
   Serial.print (isalpha('A') ?"& is a": "& is not a");
   Serial.print (" letter\r");
   Serial.print (isalpha( 'A' ) ?"4 is a":"4 is not a");
   Serial.print (" letter\r");
   Serial.print ("\rAccording to isalnum:\r");
   Serial.print (isalnum( 'A' ) ?"A is a" : "A is not a" );

   Serial.print (" digit or a letter\r" );
   Serial.print (isalnum( '8' ) ?"8 is a" : "8 is not a" ) ;
   Serial.print (" digit or a letter\r");
   Serial.print (isalnum( '#' ) ?"# is a" : "# is not a" );
   Serial.print (" digit or a letter\r");
   Serial.print ("\rAccording to isxdigit:\r");
   Serial.print (isxdigit( 'F' ) ?"F is a" : "F is not a" );
   Serial.print (" hexadecimal digit\r" );
   Serial.print (isxdigit( 'J' ) ?"J is a" : "J is not a" ) ;
   Serial.print (" hexadecimal digit\r" );
   Serial.print (isxdigit( '7' ) ?"7 is a" : "7 is not a" ) ;

   Serial.print (" hexadecimal digit\r" );
   Serial.print (isxdigit( '$' ) ? "$ is a" : "$ is not a" );
   Serial.print (" hexadecimal digit\r" );
   Serial.print (isxdigit( 'f' ) ? “f is a" : "f is not a");

}

void loop () {

}
```

Result

```
According to isdigit:
8 is a digit
# is not a digit
According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter
According to isalnum:
A is a digit or a letter

8 is a digit or a letter
# is not a digit or a letter
According to isxdigit:
```

```
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit

$ is not a hexadecimal digit
f is a hexadecimal digit
```

We use the conditional operator **(?:)** with each function to determine whether the string " is a " or the string " is not a " should be printed in the output for each character tested. For example, line **a** indicates that if '8' is a digit—i.e., if **isdigit** returns a true (nonzero) value—the string "8 is a " is printed. If '8' is not a digit (i.e., if **isdigit** returns 0), the string " 8 is not a " is printed.

Example 2

The following example demonstrates the use of the functions **islower** and **isupper**. The function **islower** determines whether its argument is a lowercase letter (a–z). Function **isupper** determines whether its argument is an uppercase letter (A–Z).

```
int thisChar = 0xA0;

void setup () {
   Serial.begin (9600);
   Serial.print ("According to islower:\r") ;
   Serial.print (islower( 'p' ) ? "p is a" : "p is not a" );
   Serial.print ( " lowercase letter\r" );
   Serial.print ( islower( 'P') ? "P is a" : "P is not a") ;
   Serial.print ("lowercase letter\r");
   Serial.print (islower( '5' ) ? "5 is a" : "5 is not a" );
   Serial.print ( " lowercase letter\r" );
   Serial.print ( islower( '!' )? "! is a" : "! is not a") ;
   Serial.print ("lowercase letter\r");

   Serial.print ("\rAccording to isupper:\r") ;
   Serial.print (isupper ( 'D' ) ? "D is a" : "D is not an" );
   Serial.print ( " uppercase letter\r" );
   Serial.print ( isupper ( 'd' )? "d is a" : "d is not an") ;
   Serial.print ( " uppercase letter\r" );
   Serial.print (isupper ( '8' ) ? "8 is a" : "8 is not an" );
   Serial.print ( " uppercase letter\r" );
   Serial.print ( islower( '$' )? "$ is a" : "$ is not an") ;
   Serial.print ("uppercase letter\r ");
}

void setup () {

}
```

Result

According to islower:

```
p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter
```

Example 3

The following example demonstrates the use of functions **isspace, iscntrl, ispunct, isprint** and **isgraph**.

- The function **isspace** determines whether its argument is a white-space character, such as space (' '), form feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t') or vertical tab ('\v').
- The function **iscntrl** determines whether its argument is a control character such as horizontal tab ('\t'), vertical tab ('\v'), form feed ('\f'), alert ('\a'), backspace ('\b'), carriage return ('\r') or newline ('\n').
- The function **ispunct** determines whether its argument is a printing character other than a space, digit or letter, such as $, #, (,), [,], {, }, ;, : or %.
- The function **isprint** determines whether its argument is a character that can be displayed on the screen (including the space character).

```
void setup () {
  Serial.begin (9600);
  Serial.print ( " According to isspace:\rNewline ") ;
  Serial.print (isspace( '\n' )? " is a" : " is not a" );
  Serial.print ( " whitespace character\rHorizontal tab") ;
  Serial.print (isspace( '\t' )? " is a" : " is not a" );
  Serial.print ( " whitespace character\n") ;
  Serial.print (isspace('%')? " % is a" : " % is not a" );

  Serial.print ( " \rAccording to iscntrl:\rNewline") ;
  Serial.print ( iscntrl( '\n' )?"is a" : " is not a" ) ;
  Serial.print (" control character\r");
  Serial.print (iscntrl( '$' ) ? " $ is a" : " $ is not a" );
  Serial.print (" control character\r");
  Serial.print ("\rAccording to ispunct:\r");
  Serial.print (ispunct(';' ) ?"; is a" : "; is not a" ) ;
  Serial.print (" punctuation character\r");
  Serial.print (ispunct('Y' ) ?"Y is a" : "Y is not a" ) ;
  Serial.print ("punctuation character\r");
  Serial.print (ispunct('#' ) ?"# is a" : "# is not a" ) ;
  Serial.print ("punctuation character\r");
```

```
   Serial.print ( "\r According to isprint:\r");
   Serial.print (isprint('$' ) ?"$ is a" : "$ is not a" );
   Serial.print (" printing character\rAlert ");
   Serial.print (isprint('\a' ) ?" is a" : " is not a" );
   Serial.print (" printing character\rSpace ");
   Serial.print (isprint(' ' ) ?" is a" : " is not a" );
   Serial.print (" printing character\r");

   Serial.print ("\r According to isgraph:\r");
   Serial.print (isgraph ('Q' ) ?"Q is a" : "Q is not a" );
   Serial.print ("printing character other than a space\rSpace ");
   Serial.print (isgraph (' ') ?" is a" : " is not a" );
   Serial.print ("printing character other than a space ");
}

void loop () {

}
```

Result

```
According to isspace:
Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character
According to iscntrl:
Newline is a control character
$ is not a control character
According to ispunct:
; is a punctuation character
Y is not a punctuation character
# is a punctuation character
According to isprint:
$ is a printing character
Alert is not a printing character
Space is a printing character
According to isgraph:
Q is a printing character other than a space
Space is not a printing character other than a space
```

16.ARDUINO - MATH LIBRARY

The Arduino Math library (math.h) includes a number of useful mathematical functions for manipulating floating-point numbers.

Library Macros

Following are the macros defined in the header math.h −

Given below is the list of macros defined in the header math.h

Library Functions

The following functions are defined in the header **math.h** −

Given below is the list of functions are defined in the header math.h

Example

The following example shows how to use the most common math.h library functions −

```
double double__x = 45.45 ;
double double__y = 30.20 ;

void setup() {
   Serial.begin(9600);
   Serial.print("cos num = ");
   Serial.println (cos (double__x) ); // returns cosine of x
   Serial.print("absolute value of num = ");
   Serial.println (fabs (double__x) ); // absolute value of a float
   Serial.print("floating point modulo = ");
   Serial.println (fmod (double__x, double__y)); // floating point modulo
   Serial.print("sine of num = ");
   Serial.println (sin (double__x) ) ;// returns sine of x
   Serial.print("square root of num : ");
   Serial.println ( sqrt (double__x) );// returns square root of x
   Serial.print("tangent of num : ");
   Serial.println ( tan (double__x) ); // returns tangent of x
   Serial.print("exponential value of num : ");
   Serial.println ( exp (double__x) ); // function returns the exponential value of x.
   Serial.print("cos num : ");

   Serial.println (atan (double__x) ); // arc tangent of x
   Serial.print("tangent of num : ");
   Serial.println (atan2 (double__y, double__x) );// arc tangent of y/x
   Serial.print("arc tangent of num : ");
   Serial.println (log (double__x) ) ; // natural logarithm of x
   Serial.print("cos num : ");
   Serial.println ( log10 (double__x)); // logarithm of x to base
}

void loop() {

}
```

Result

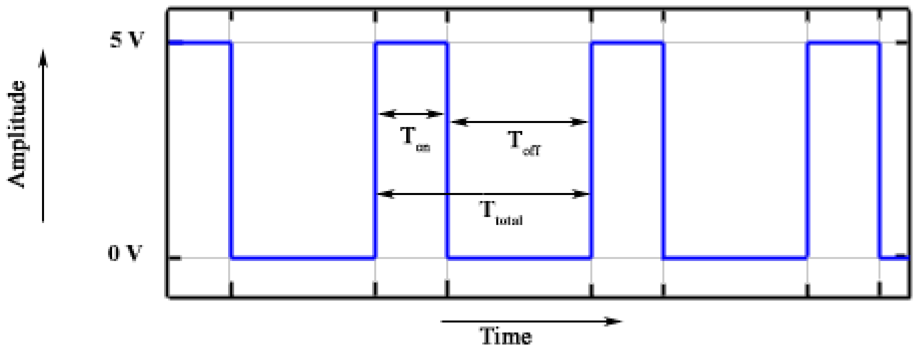
```
cos num = 0.10
absolute value of num = 45.45
floating point modulo =15.25
sine of num = 0.99
square root of num : 6.74
tangent of num : 9.67
exponential value of num : ovf
cos num : 1.55
tangent of num : 0.59
arc tangent of num : 3.82
cos num : 1.66
logarithm of num to base 10 : inf
power of num : 2065.70
```

17.ARDUINO - PULSE WIDTH MODULATION

Pulse Width Modulation or PWM is a common technique used to vary the width of the pulses in a pulse-train. PWM has many applications such as controlling servos and speed controllers, limiting the effective power of motors and LEDs.

Basic Principle of PWM

Pulse width modulation is basically, a square wave with a varying high and low time. A basic PWM signal is shown in the following figure.



There are various terms associated with PWM −

- **On-Time −** Duration of time signal is high.
- **Off-Time −** Duration of time signal is low.
- **Period −** It is represented as the sum of on-time and off-time of PWM signal.

- **Duty Cycle −** It is represented as the percentage of time signal that remains on during the period of the PWM signal.

Duty Cycle

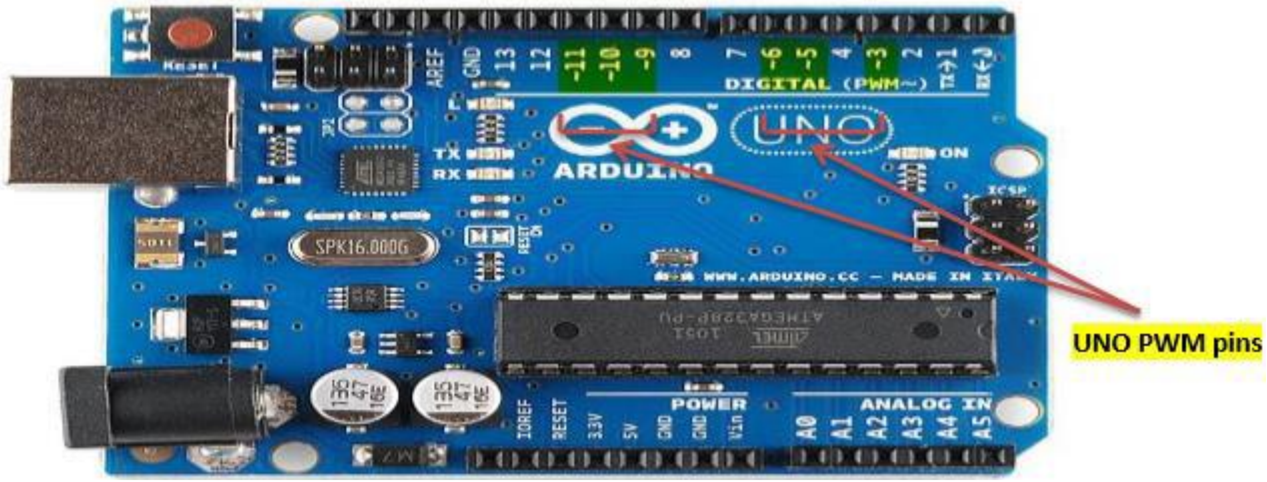
Duty cycle is calculated as the on-time of the period of time. Using the period calculated above, duty cycle is calculated as −

$$D = \frac{T_{on}}{T_{on}+T_{off}} = \frac{T_{on}}{T_{total}}$$

analogWrite() Function

The **analogWrite()** function writes an analog value (PWM wave) to a pin. It can be used to light a LED at varying brightness or drive a motor at various speeds. After a call of the analogWrite() function, the pin will generate a steady square wave of the specified duty cycle until the next call to analogWrite() or a call to digitalRead() or digitalWrite() on the same pin. The frequency of the PWM signal on most pins is approximately 490 Hz. On the Uno and similar boards, pins 5 and 6 have a frequency of approximately 980 Hz. Pins 3 and 11 on the Leonardo also run at 980 Hz.

On most Arduino boards (those with the ATmega168 or ATmega328), this function works on pins 3, 5, 6, 9, 10, and 11. On the Arduino Mega, it works on pins 2 - 13 and 44 - 46. Older Arduino boards with an ATmega8 only support **analogWrite()** on pins 9, 10, and 11.



The Arduino Due supports **analogWrite()** on pins 2 through 13, and pins DAC0 and DAC1. Unlike the PWM pins, DAC0 and DAC1 are Digital to Analog converters, and act as true analog outputs.

You do not need to call pinMode() to set the pin as an output before calling analogWrite().

analogWrite() Function Syntax

analogWrite (pin , value) ;

value − the duty cycle: between 0 (always off) and 255 (always on).

Example

```
int ledPin = 9; // LED connected to digital pin 9
int analogPin = 3; // potentiometer connected to analog pin 3
int val = 0; // variable to store the read value

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

void loop() {
  val = analogRead(analogPin); // read the input pin
  analogWrite(ledPin, (val / 4)); // analogRead values go from 0 to 1023,
    // analogWrite values from 0 to 255
}
```

18.ARDUINO - RANDOM NUMBERS

To generate random numbers, you can use Arduino random number functions. We have two functions −

- randomSeed(seed)
- random()

randomSeed (seed)

The function randomSeed(seed) resets Arduino's pseudorandom number generator. Although the distribution of the numbers returned by random() is essentially random, the sequence is predictable. You should reset the generator to some random value. If you have an unconnected analog pin, it might pick up random noise from the surrounding environment. These may be radio waves, cosmic rays, electromagnetic interference from cell phones, fluorescent lights and so on.

Example

```
randomSeed(analogRead(5)); // randomize using noise from analog pin 5
```

random( )

The random function generates pseudo-random numbers. Following is the syntax.

random( ) Statements Syntax

long random(max) // it generate random numbers from 0 to max
long random(min, max) // it generate random numbers from min to max

Example

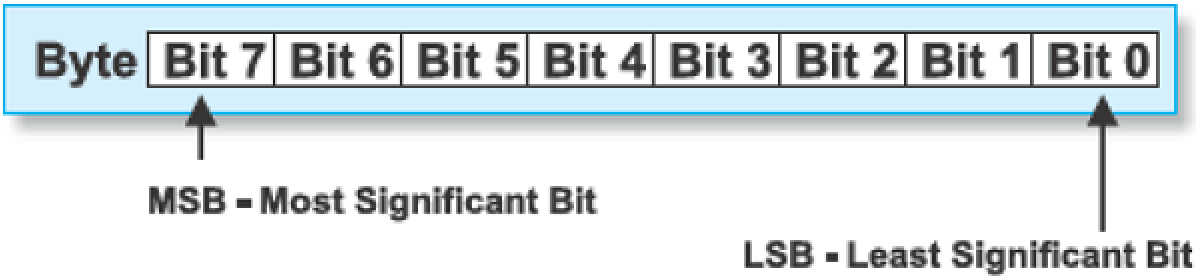
```
long randNumber;

void setup() {
  Serial.begin(9600);
  // if analog input pin 0 is unconnected, random analog
  // noise will cause the call to randomSeed() to generate
  // different seed numbers each time the sketch runs.
  // randomSeed() will then shuffle the random function.
  randomSeed(analogRead(0));
}

void loop() {
  // print a random number from 0 to 299
  Serial.print("random1=");
  randNumber = random(300);
  Serial.println(randNumber); // print a random number from 0to 299
  Serial.print("random2=");
  randNumber = random(10, 20);// print a random number from 10 to 19
  Serial.println (randNumber);
  delay(50);
}
```

Let us now refresh our knowledge on some of the basic concepts such as bits and bytes.

Bits

A bit is just a binary digit.

- The binary system uses two digits, 0 and 1.
- Similar to the decimal number system, in which digits of a number do not have the same value, the 'significance' of a bit depends on its position in the binary number. For example, digits in the decimal number 666 are the same, but have different values.



Bytes

A byte consists of eight bits.

- If a bit is a digit, it is logical that bytes represent numbers.
- All mathematical operations can be performed upon them.
- The digits in a byte do not have the same significance either.
- The leftmost bit has the greatest value called the Most Significant Bit (MSB).

- The rightmost bit has the least value and is therefore, called the Least Significant Bit (LSB).
- Since eight zeros and ones of one byte can be combined in 256 different ways, the largest decimal number that can be represented by one byte is 255 (one combination represents a zero).

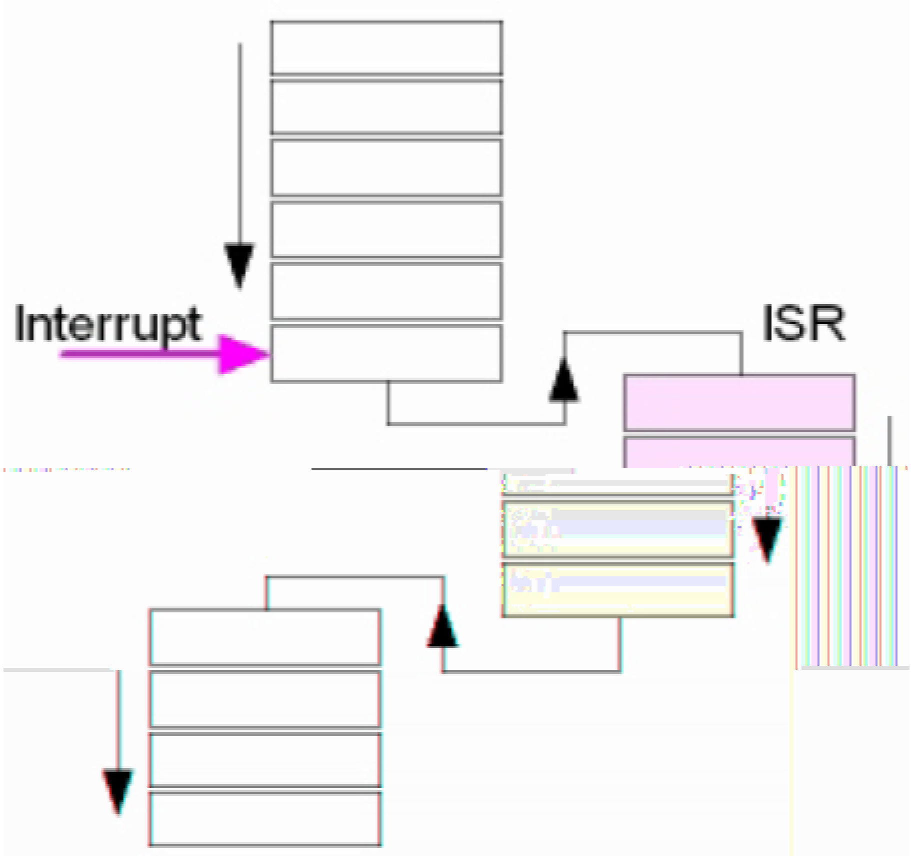
ARDUINO - INTERRUPTS

Interrupts stop the current work of Arduino such that some other work can be done.

Suppose you are sitting at home, chatting with someone. Suddenly the telephone rings. You stop chatting, and pick up the telephone to speak to the caller. When you have finished your telephonic conversation, you go back to chatting with the person before the telephone rang.

Similarly, you can think of the main routine as chatting to someone, the telephone ringing causes you to stop chatting. The interrupt service routine is the process of talking on the telephone. When the telephone conversation ends, you then go back to your main routine of chatting. This example explains exactly how an interrupt causes a processor to act.

The main program is running and performing some function in a circuit. However, when an interrupt occurs the main program halts while another routine is carried out. When this routine finishes, the processor goes back to the main routine again.



Important features

Here are some important features about interrupts −

- Interrupts can come from various sources. In this case, we are using a hardware interrupt that is triggered by a state change on one of the digital pins.
- Most Arduino designs have two hardware interrupts (referred to as "interrupt0" and "interrupt1") hard-wired to digital I/O pins 2 and 3, respectively.
- The Arduino Mega has six hardware interrupts including the additional interrupts ("interrupt2" through "interrupt5") on pins 21, 20, 19, and 18.
- You can define a routine using a special function called as “Interrupt Service Routine” (usually known as ISR).
- You can define the routine and specify conditions at the rising edge, falling edge or both. At these specific conditions, the interrupt would be serviced.
- It is possible to have that function executed automatically, each time an event happens on an input pin.

Types of Interrupts

There are two types of interrupts −

- **Hardware Interrupts −** They occur in response to an external event, such as an external interrupt pin going high or low.

- **Software Interrupts** − They occur in response to an instruction sent in software. The only type of interrupt that the “Arduino language” supports is the attachInterrupt() function.

Using Interrupts in Arduino

Interrupts are very useful in Arduino programs as it helps in solving timing problems. A good application of an interrupt is reading a rotary encoder or observing a user input. Generally, an ISR should be as short and fast as possible. Other interrupts will be executed after the current one finishes in an order that depends on the priority they have.

Typically, global variables are used to pass data between an ISR and the main program. To make sure variables shared between an ISR and the main program are updated correctly, declare them as volatile.

attachInterrupt Statement Syntax

```
attachInterrupt(digitalPinToInterrupt(pin),ISR,mode);//recommended for arduino board
attachInterrupt(pin, ISR, mode) ; //recommended Arduino Due, Zero only
//argument pin: the pin number
//argument ISR: the ISR to call when the interrupt occurs;
  //this function must take no parameters and return nothing.
```

The following three constants are predefined as valid values −

- **LOW** to trigger the interrupt whenever the pin is low.
- **CHANGE** to trigger the interrupt whenever the pin changes value.
- **FALLING** whenever the pin goes from high to low.

Example

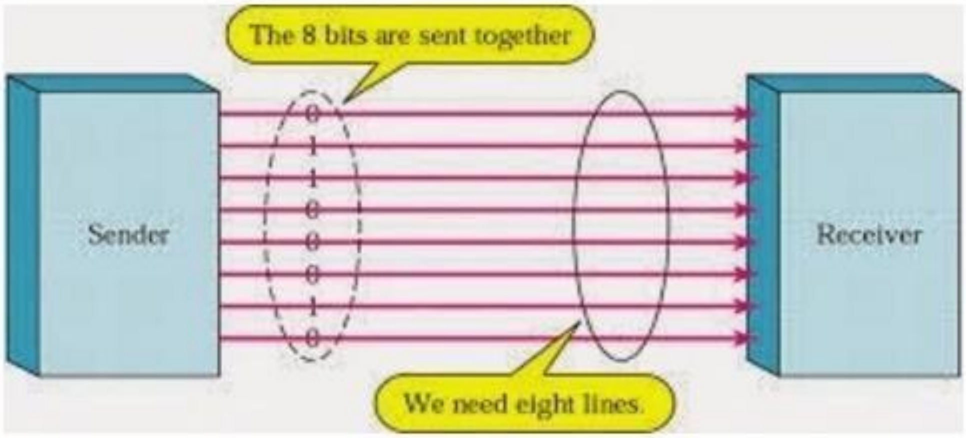
```
int pin = 2; //define interrupt pin to 2
volatile int state = LOW; // To make sure variables shared between an ISR
//the main program are updated correctly,declare them as volatile.

void setup() {
  pinMode(13, OUTPUT); //set pin 13 as output
  attachInterrupt(digitalPinToInterrupt(pin), blink, CHANGE);
  //interrupt at pin 2 blink ISR when pin to change the value
}
void loop() {
  digitalWrite(13, state); //pin 13 equal the state value
}

void blink() {
  //ISR function
  state = !state; //toggle the state when the interrupt occurs
}
```

19.ARDUINO - COMMUNICATION

Hundreds of communication protocols have been defined to achieve this data exchange. Each protocol can be categorized into one of the two categories: parallel or serial.

Parallel Communication

Parallel connection between the Arduino and peripherals via input/output ports is the ideal solution for shorter distances up to several meters. However, in other cases when it is necessary to establish communication between two devices for longer distances it is not possible to use parallel connection. Parallel interfaces transfer multiple bits at the same time. They usually require buses of data - transmitting across eight, sixteen, or more wires. Data is transferred in huge, crashing waves of 1's and 0's.



Advantages and Drawbacks of Parallel Communication

Parallel communication certainly has its advantages. It is faster than serial, straightforward, and relatively easy to implement. However, it requires many input/output (I/O) ports and lines. If you have ever had to move a project from a basic Arduino Uno to a Mega, you know that the I/O lines on a microprocessor can be precious and few. Therefore, we prefer serial communication, sacrificing potential speed for pin real estate.

Serial Communication Modules

Today, most Arduino boards are built with several different systems for serial communication as standard equipment.

Which of these systems are used depends on the following factors −

- How many devices the microcontroller has to exchange data with?
- How fast the data exchange has to be?
- What is the distance between these devices?
- Is it necessary to send and receive data simultaneously?

One of the most important things concerning serial communication is the **Protocol**, which should be strictly observed. It is a set of rules, which must be applied such that the devices can correctly interpret data they mutually exchange. Fortunately, Arduino automatically takes care of this, so that the work of the programmer/user is reduced to simple write (data to be sent) and read (received data).

Types of Serial Communications

Serial communication can be further classified as −

- **Synchronous** − Devices that are synchronized use the same clock and their timing is in synchronization with each other.
- **Asynchronous** − Devices that are asynchronous have their own clocks and are triggered by the output of the previous state.

It is easy to find out if a device is synchronous or not. If the same clock is given to all the connected devices, then they are synchronous. If there is no clock line, it is asynchronous.

For example, UART (Universal Asynchronous Receiver Transmitter) module is asynchronous.

The asynchronous serial protocol has a number of built-in rules. These rules are nothing but mechanisms that help ensure robust and error-free data transfers. These mechanisms, which we get for eschewing the external clock signal, are −

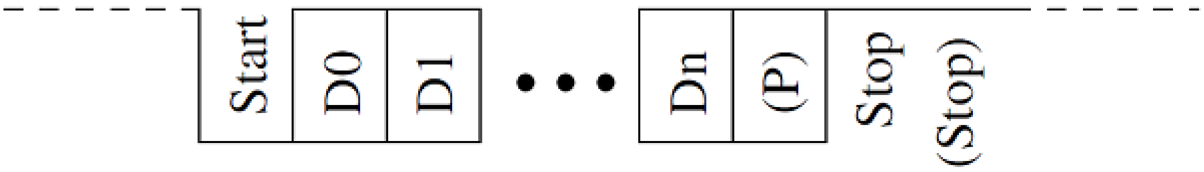
- Synchronization bits
- Data bits
- Parity bits
- Baud rate

Synchronization Bits

The synchronization bits are two or three special bits transferred with each packet of data. They are the start bit and the stop bit(s). True to their name, these bits mark the beginning and the end of a packet respectively.

There is always only one start bit, but the number of stop bits is configurable to either one or two (though it is normally left at one).

The start bit is always indicated by an idle data line going from 1 to 0, while the stop bit(s) will transition back to the idle state by holding the line at 1.



Data Bits

The amount of data in each packet can be set to any size from 5 to 9 bits. Certainly, the standard data size is your basic 8-bit byte, but other sizes have their uses. A 7-bit data packet can be more efficient than 8, especially if you are just transferring 7-bit ASCII characters.

Parity Bits

The user can select whether there should be a parity bit or not, and if yes, whether the parity should be odd or even. The parity bit is 0 if the number of 1's among the data bits is even. Odd parity is just the opposite.

Baud Rate

The term baud rate is used to denote the number of bits transferred per second [bps]. Note that it refers to bits, not bytes. It is usually required by the protocol that each byte is transferred along with several control bits. It means that one byte in serial data stream may consist of 11 bits. For example, if the baud rate is 300 bps then maximum 37 and minimum 27 bytes may be transferred per second.

Arduino UART

The following code will make Arduino send hello world when it starts up.

```
void setup() {
  Serial.begin(9600); //set up serial library baud rate to 9600
  Serial.println("hello world"); //print hello world
}

void loop() {

}
```

After the Arduino sketch has been uploaded to Arduino, open the Serial monitor 🔍 at the top right section of Arduino IDE.

Type anything into the top box of the Serial Monitor and press send or enter on your keyboard. This will send a series of bytes to the Arduino.

The following code returns whatever it receives as an input.

The following code will make Arduino deliver output depending on the input provided.

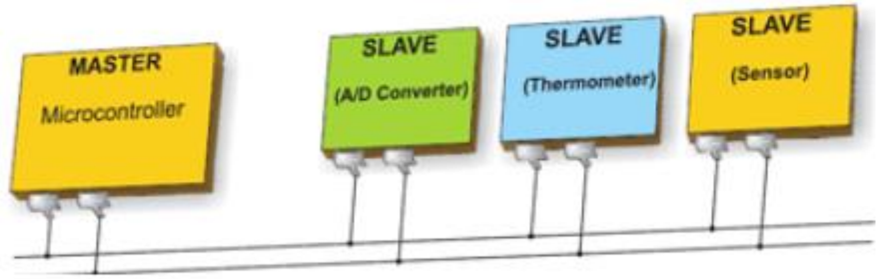
```
void setup() {
  Serial.begin(9600); //set up serial library baud rate to 9600
}

void loop() {
  if(Serial.available()) //if number of bytes (characters) available for reading from {
    serial port
    Serial.print("I received:"); //print I received
    Serial.write(Serial.read()); //send what you read
  }
}
```

Notice that **Serial.print** and **Serial.println** will send back the actual ASCII code, whereas **Serial.write** will send back the actual text. See ASCII codes for more information.

ARDUINO - INTER INTEGRATED CIRCUIT

Inter-integrated circuit (I2C) is a system for serial data exchange between the microcontrollers and specialized integrated circuits of a new generation. It is used when the distance between them is short (receiver and transmitter are usually on the same printed board). Connection is established via two conductors. One is used for data transfer and the other is used for synchronization (clock signal).

As seen in the following figure, one device is always a master. It performs addressing of one slave chip before the communication starts. In this way, one microcontroller can communicate with 112 different devices. Baud rate is usually 100 Kb/sec (standard mode) or 10 Kb/sec (slow baud rate mode). Systems with the baud rate of 3.4 Mb/sec have recently appeared. The distance between devices, which communicate over an I2C bus is limited to several meters.



Board I2C Pins

The I2C bus consists of two signals − SCL and SDA. SCL is the clock signal, and SDA is the data signal. The current bus master always generates the clock signal. Some slave devices may force the clock low at times to delay the master sending more data (or to require more time to prepare data before the master attempts to clock it out). This is known as “clock stretching”.

Following are the pins for different Arduino boards −

- Uno, Pro Mini A4 (SDA), A5 (SCL)
- Mega, Due 20 (SDA), 21 (SCL)
- Leonardo, Yun 2 (SDA), 3 (SCL)

Arduino I2C

We have two modes - master code and slave code - to connect two Arduino boards using I2C. They are −

- Master Transmitter / Slave Receiver
- Master Receiver / Slave Transmitter

Master Transmitter / Slave Receiver

Let us now see what is master transmitter and slave receiver.

Master Transmitter

The following functions are used to initialize the Wire library and join the I2C bus as a master or slave. This is normally called only once.

- **Wire.begin(address)** − Address is the 7-bit slave address in our case as the master is not specified and it will join the bus as a master.
- **Wire.beginTransmission(address)** − Begin a transmission to the I2C slave device with the given address.
- **Wire.write(value)** − Queues bytes for transmission from a master to slave device (in-between calls to beginTransmission() and endTransmission()).
- **Wire.endTransmission()** − Ends a transmission to a slave device that was begun by beginTransmission() and transmits the bytes that were queued by wire.write().

Example

```
#include <Wire.h> //include wire library

void setup() //this will run only once {
   Wire.begin(); // join i2c bus as master
}

short age = 0;

void loop() {
   Wire.beginTransmission(2);
   // transmit to device #2
   Wire.write("age is = ");
   Wire.write(age); // sends one byte
   Wire.endTransmission(); // stop transmitting
   delay(1000);
}
```

Slave Receiver

The following functions are used −

- **Wire.begin(address)** − Address is the 7-bit slave address.

- **Wire.onReceive(received data handler) −** Function to be called when a slave device receives data from the master.
- **Wire.available() −** Returns the number of bytes available for retrieval with Wire.read().This should be called inside the Wire.onReceive() handler.

Example

```
#include <Wire.h> //include wire library

void setup() {  //this will run only once
   Wire.begin(2); // join i2c bus with address #2
   Wire.onReceive(receiveEvent); // call receiveEvent when the master send any thing
   Serial.begin(9600); // start serial for output to print what we receive
}

void loop() {
   delay(250);
}

//-----this function will execute whenever data is received from master-----//

void receiveEvent(int howMany) {
   while (Wire.available()>1) // loop through all but the last {
      char c = Wire.read(); // receive byte as a character
      Serial.print(c); // print the character
   }
}
```

Master Receiver / Slave Transmitter

Let us now see what is master receiver and slave transmitter.

Master Receiver

The Master, is programmed to request, and then read bytes of data that are sent from the uniquely addressed Slave Arduino.

The following function is used −

Wire.requestFrom(address,number of bytes) − Used by the master to request bytes from a slave device. The bytes may then be retrieved with the functions wire.available() and wire.read() functions.

Example

```
#include <Wire.h> //include wire library void setup() {
   Wire.begin(); // join i2c bus (address optional for master)
   Serial.begin(9600); // start serial for output
}

void loop() {
   Wire.requestFrom(2, 1); // request 1 bytes from slave device #2
   while (Wire.available()) // slave may send less than requested {
      char c = Wire.read(); // receive a byte as character
      Serial.print(c); // print the character
   }
   delay(500);
}
```

Slave Transmitter

The following function is used.

Wire.onRequest(handler) − A function is called when a master requests data from this slave device.

Example

```
#include <Wire.h>

void setup() {
   Wire.begin(2); // join i2c bus with address #2
   Wire.onRequest(requestEvent); // register event
}

Byte x = 0;

void loop() {
   delay(100);
}

// function that executes whenever data is requested by master
// this function is registered as an event, see setup()

void requestEvent() {
   Wire.write(x); // respond with message of 1 bytes as expected by master
   x++;
}
```

20.ARDUINO - BLINKING LED

LEDs are small, powerful lights that are used in many different applications. To start, we will work on blinking an LED, the Hello World of microcontrollers. It is as simple as turning a light on and off. Establishing this important baseline will give you a solid foundation as we work towards experiments that are more complex.

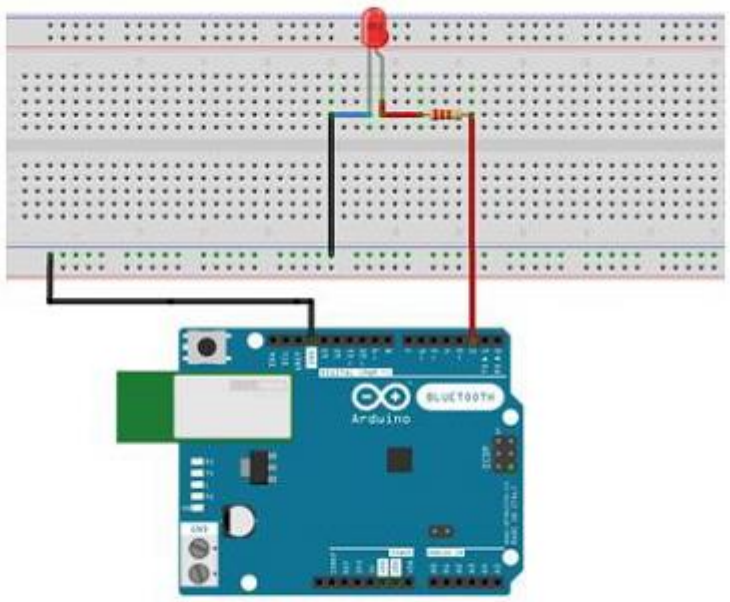
Components Required

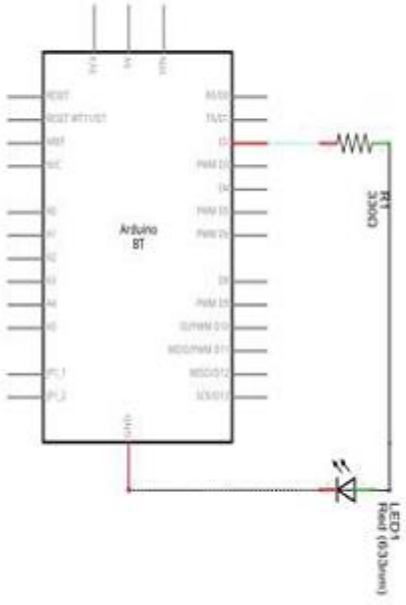
You will need the following components −

- 1 × Breadboard
- 1 × Arduino Uno R3
- 1 × LED
- 1 × 330Ω Resistor
- 2 × Jumper

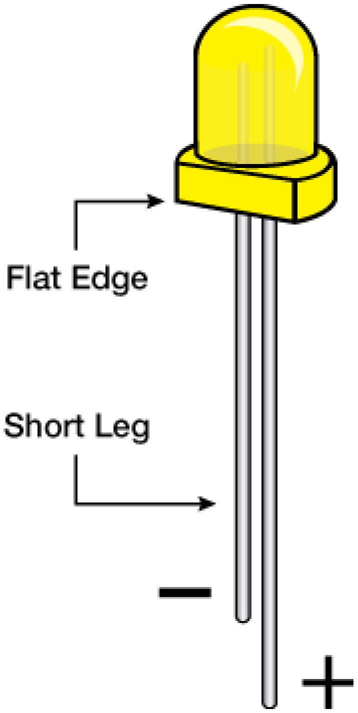
Procedure

Follow the circuit diagram and hook up the components on the breadboard as shown in the image given below.

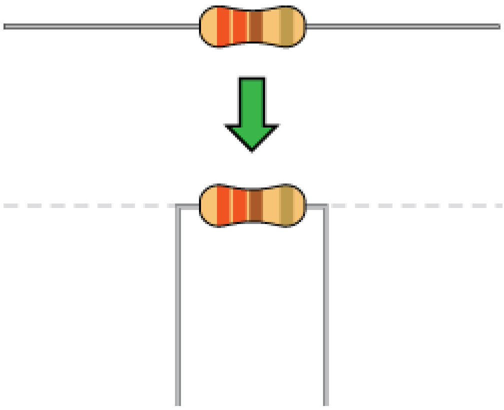




Note − To find out the polarity of an LED, look at it closely. The shorter of the two legs, towards the flat edge of the bulb indicates the negative terminal.

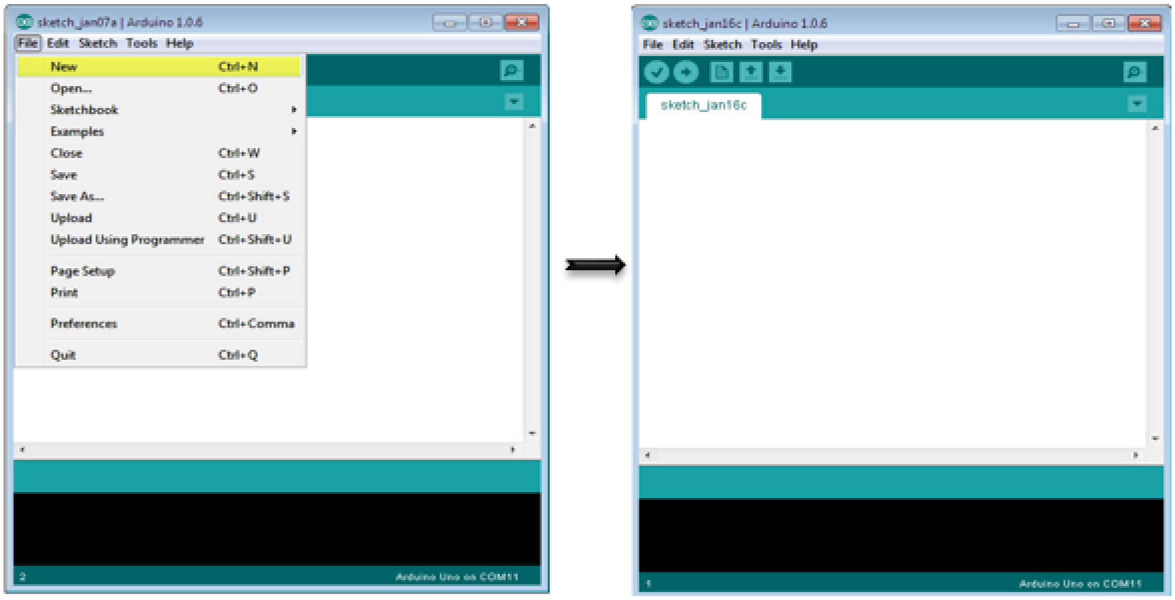


Components like resistors need to have their terminals bent into 90° angles in order to fit the breadboard sockets properly. You can also cut the terminals shorter.



Sketch

Open the Arduino IDE software on your computer. Coding in the Arduino language will control your circuit. Open the new sketch File by clicking New.



Arduino Code

```
/*
   Blink
   Turns on an LED on for one second, then off for one second, repeatedly.
*/

// the setup function runs once when you press reset or power the board

void setup() {  // initialize digital pin 13 as an output.
   pinMode(2, OUTPUT);
}

// the loop function runs over and over again forever

void loop() {
   digitalWrite(2, HIGH); // turn the LED on (HIGH is the voltage level)
   delay(1000); // wait for a second
   digitalWrite(2, LOW); // turn the LED off by making the voltage LOW
   delay(1000); // wait for a second
}
```

Code to Note

pinMode(2, OUTPUT) − Before you can use one of Arduino's pins, you need to tell Arduino Uno R3 whether it is an INPUT or OUTPUT. We use a built-in "function" called pinMode() to do this.

21.ARDUINO - FADING LED

This example demonstrates the use of the analogWrite() function in fading an LED off. AnalogWrite uses pulse width modulation (PWM), turning a digital pin on and off very quickly with different ratios between on and off, to create a fading effect.

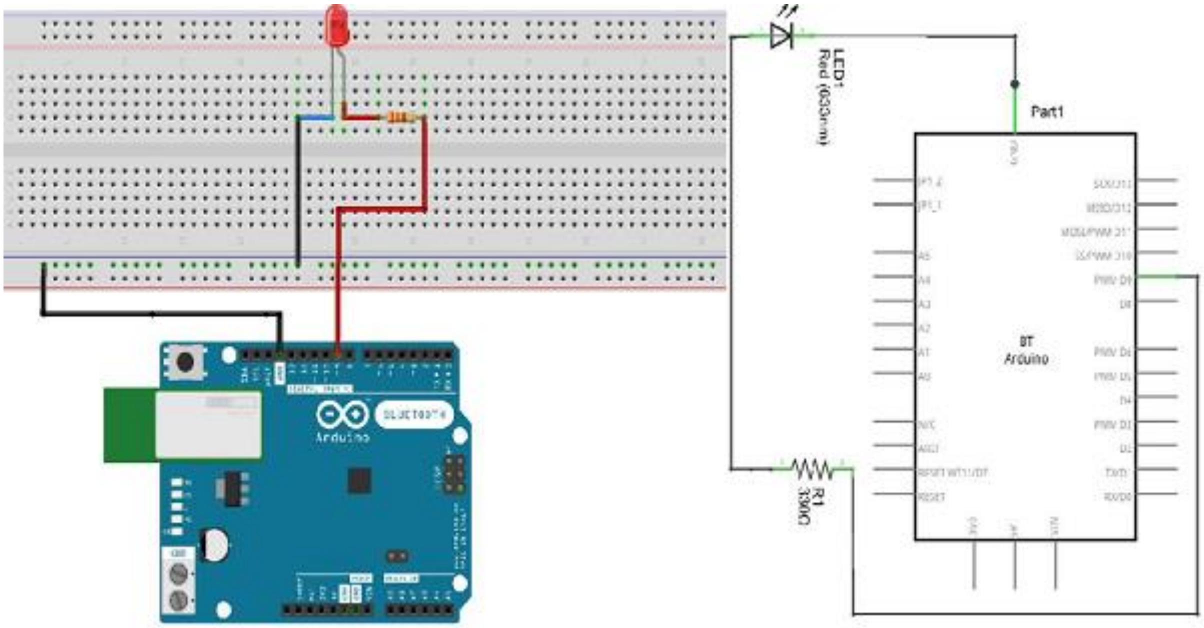
Components Required

You will need the following components −

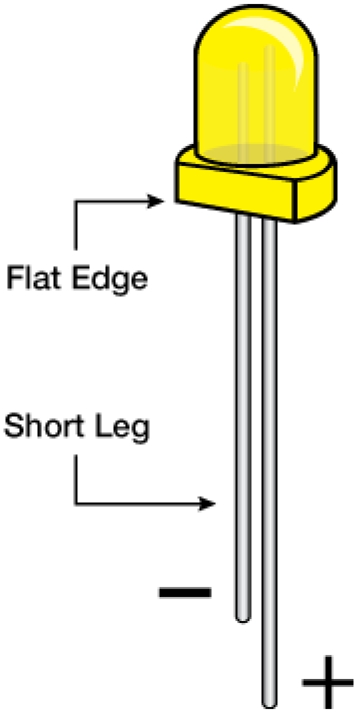
- 1 × Breadboard
- 1 × Arduino Uno R3
- 1 × LED
- 1 × 330Ω Resistor
- 2 × Jumper

Procedure

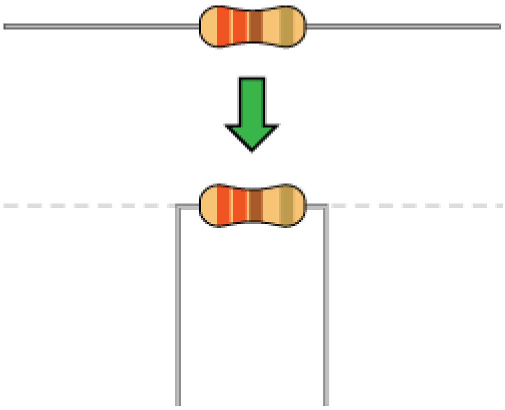
Follow the circuit diagram and hook up the components on the breadboard as shown in the image given below.



Note − To find out the polarity of an LED, look at it closely. The shorter of the two legs, towards the flat edge of the bulb indicates the negative terminal.

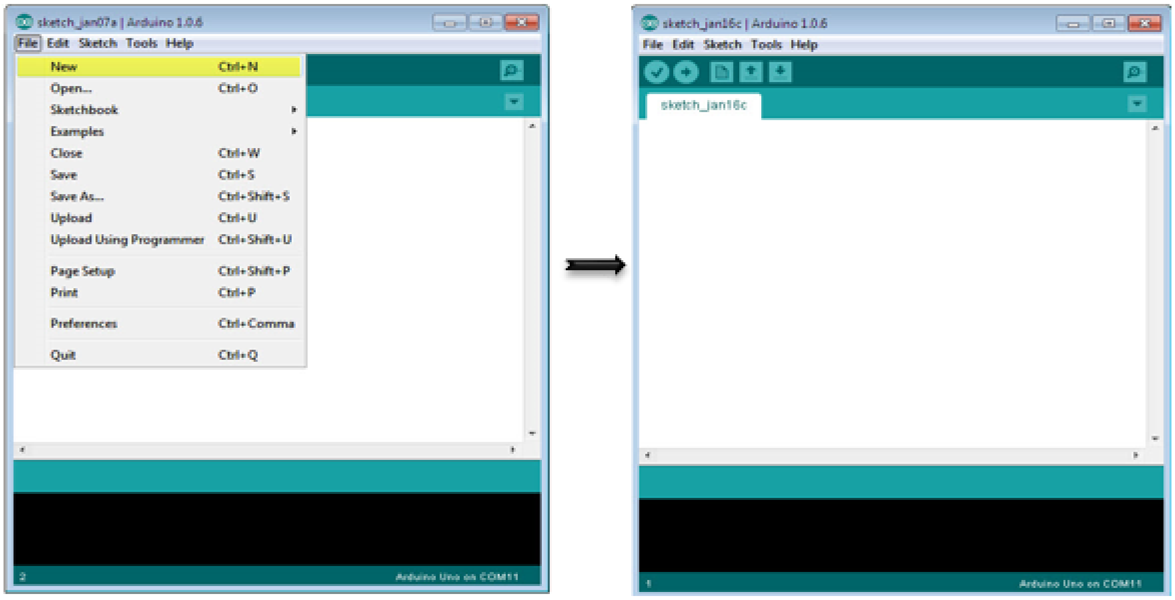


Components like resistors need to have their terminals bent into 90° angles in order to fit the breadboard sockets properly. You can also cut the terminals shorter.



Sketch

Open the Arduino IDE software on your computer. Coding in the Arduino language will control your circuit. Open the new sketch File by clicking New.



Arduino Code

```
/*
  Fade
  This example shows how to fade an LED on pin 9 using the analogWrite() function.

  The analogWrite() function uses PWM, so if you want to change the pin you're using, be
  sure to use another PWM capable pin. On most Arduino, the PWM pins are identified with
  a "~" sign, like ~3, ~5, ~6, ~9, ~10 and ~11.
*/

int led = 9; // the PWM pin the LED is attached to
int brightness = 0; // how bright the LED is
int fadeAmount = 5; // how many points to fade the LED by
// the setup routine runs once when you press reset:

void setup() {
  // declare pin 9 to be an output:
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:

void loop() {
  // set the brightness of pin 9:
  analogWrite(led, brightness);
  // change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;
  // reverse the direction of the fading at the ends of the fade:
  if (brightness == 0 || brightness == 255) {
    fadeAmount = -fadeAmount ;
  }
  // wait for 30 milliseconds to see the dimming effect
  delay(300);
}
```

Code to Note

After declaring pin 9 as your LED pin, there is nothing to do in the setup() function of your code. The analogWrite() function that you will be using in the main loop of your code requires two arguments: One, telling the function which pin to write to and the other indicating what PWM value to write.

In order to fade the LED off and on, gradually increase the PWM values from 0 (all the way off) to 255 (all the way on), and then back to 0, to complete the cycle. In the sketch given above, the PWM value is set using a variable called brightness. Each time through the loop, it increases by the value of the variable **fadeAmount**.

If brightness is at either extreme of its value (either 0 or 255), then fadeAmount is changed to its negative. In other words, if fadeAmount is 5, then it is set to -5. If it is -5, then it is set to 5. The next time through the loop, this change causes brightness to change direction as well.

analogWrite() can change the PWM value very fast, so the delay at the end of the sketch controls the speed of the fade. Try changing the value of the delay and see how it changes the fading effect.

Result

You should see your LED brightness change gradually.

22.ARDUINO - HUMIDITY SENSOR

In this section, we will learn how to interface our Arduino board with different sensors. We will discuss the following sensors −

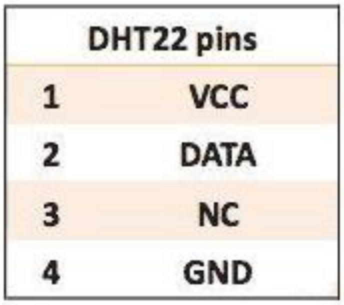
- Humidity sensor (DHT22)
- Temperature sensor (LM35)
- Water detector sensor (Simple Water Trigger)
- PIR SENSOR
- ULTRASONIC SENSOR
- GPS

Humidity Sensor (DHT22)

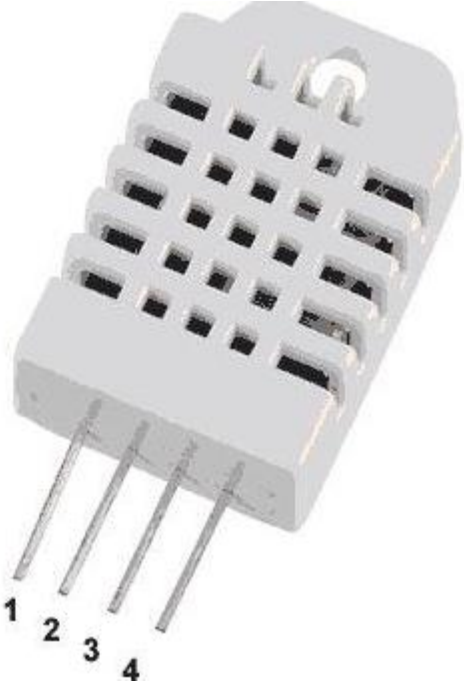
The DHT-22 (also named as AM2302) is a digital-output, relative humidity, and temperature sensor. It uses a capacitive humidity sensor and a thermistor to measure the surrounding air, and sends a digital signal on the data pin.

In this example, you will learn how to use this sensor with Arduino UNO. The room temperature and humidity will be printed to the serial monitor.

The DHT-22 Sensor



DHT22 pins	
1	VCC
2	DATA
3	NC
4	GND



The connections are simple. The first pin on the left to 3-5V power, the second pin to the data input pin and the right-most pin to the ground.

Technical Details

- **Power** − 3-5V
- **Max Current** − 2.5mA
- **Humidity** − 0-100%, 2-5% accuracy
- **Temperature** − 40 to 80°C, ±0.5°C accuracy

Components Required

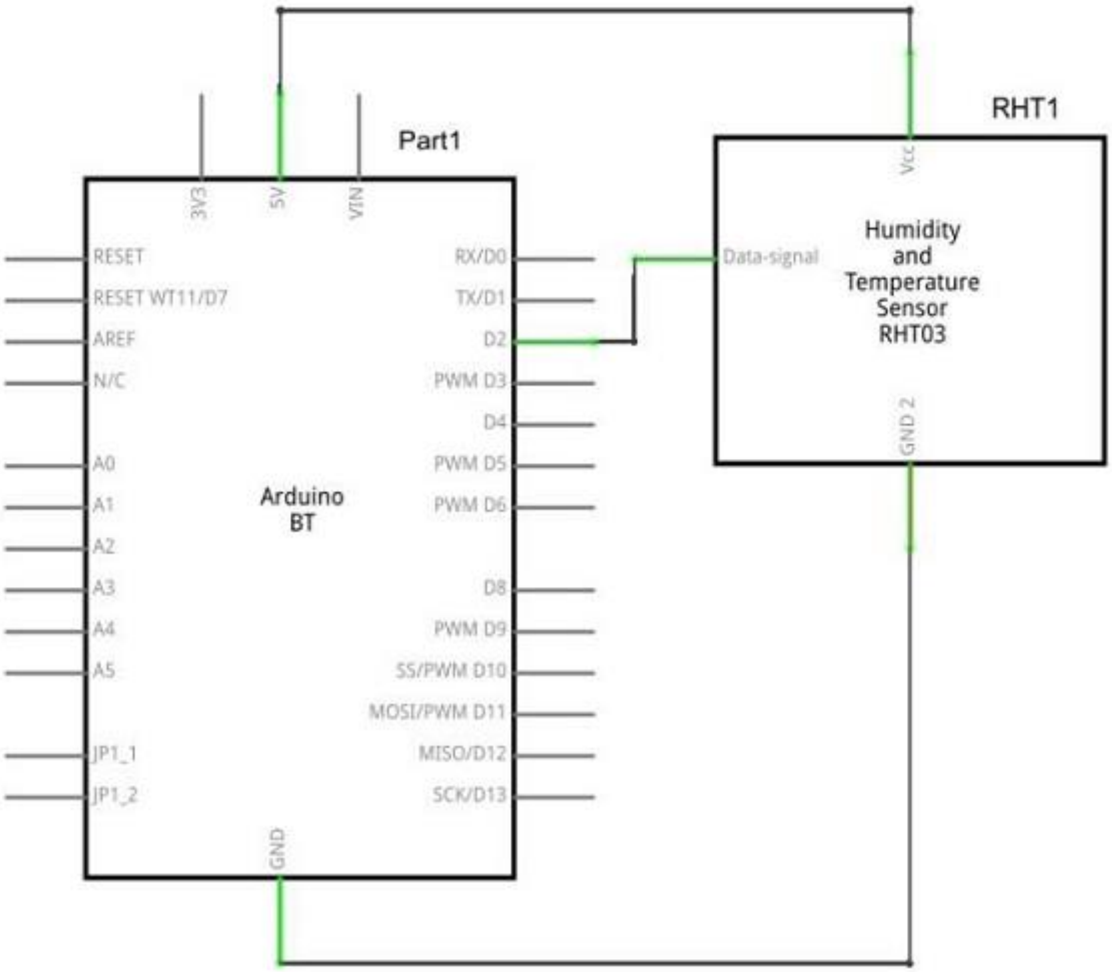
You will need the following components −

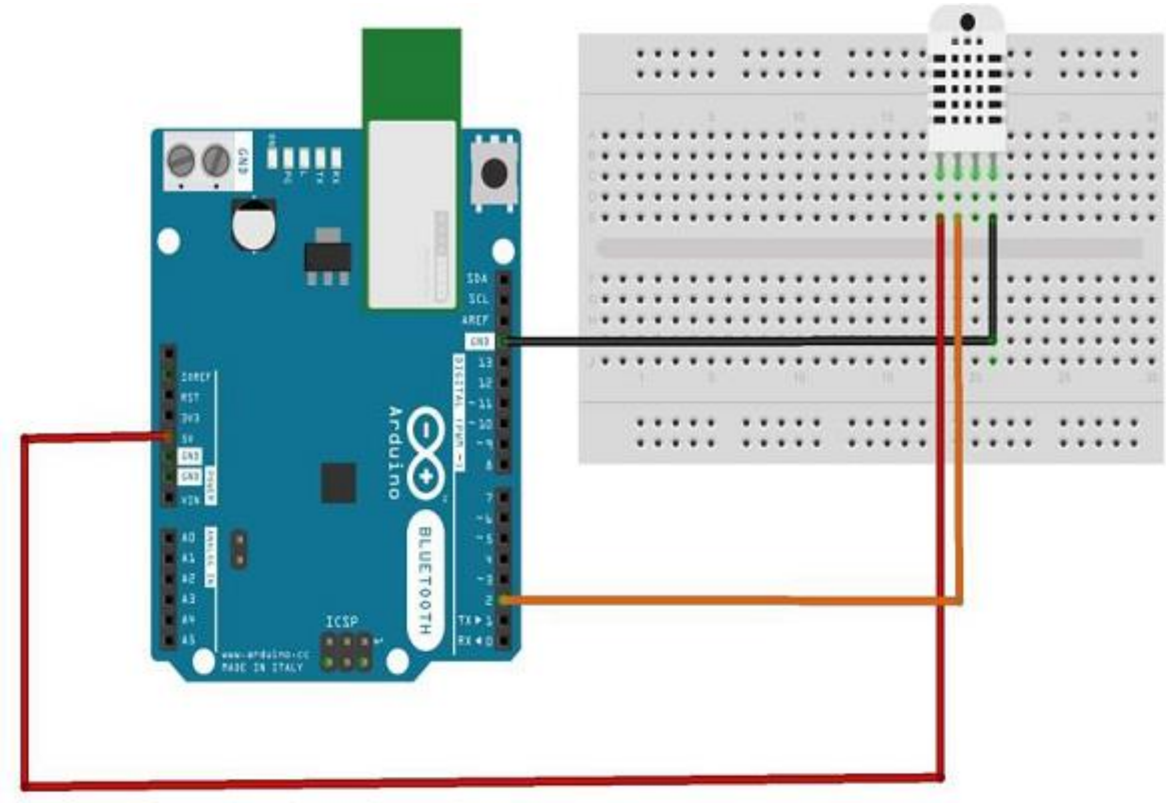
- 1 × Breadboard
- 1 × Arduino Uno R3
- 1 × DHT22

- 1 × 10K ohm resistor

Procedure

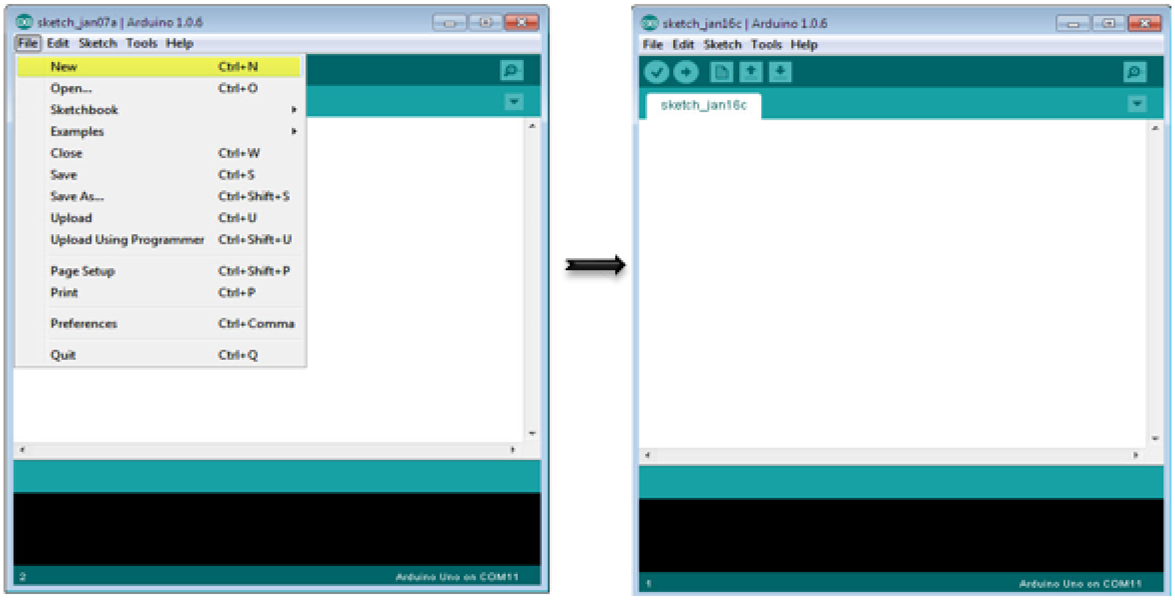
Follow the circuit diagram and hook up the components on the breadboard as shown in the image below.





Sketch

Open the Arduino IDE software on your computer. Coding in the Arduino language will control your circuit. Open a new sketch File by clicking New.



Arduino Code

```
// Example testing sketch for various DHT humidity/temperature sensors
```

```
#include "DHT.h"
#define DHTPIN 2 // what digital pin we're connected to
// Uncomment whatever type you're using!
//#define DHTTYPE DHT11 // DHT 11
#define DHTTYPE DHT22 // DHT 22 (AM2302), AM2321
//#define DHTTYPE DHT21 // DHT 21 (AM2301)
// Connect pin 1 (on the left) of the sensor to +5V
// NOTE: If using a board with 3.3V logic like an Arduino Due connect pin 1
// to 3.3V instead of 5V!
// Connect pin 2 of the sensor to whatever your DHTPIN is
// Connect pin 4 (on the right) of the sensor to GROUND
// Connect a 10K resistor from pin 2 (data) to pin 1 (power) of the sensor
// Initialize DHT sensor.
// Note that older versions of this library took an optional third parameter to
// tweak the timings for faster processors. This parameter is no longer needed
// as the current DHT reading algorithm adjusts itself to work on faster procs.
DHT dht(DHTPIN, DHTTYPE);

void setup() {
  Serial.begin(9600);
  Serial.println("DHTxx test!");
  dht.begin();
}

void loop() {
  delay(2000); // Wait a few seconds between measurements
  float h = dht.readHumidity();
  // Reading temperature or humidity takes about 250 milliseconds!
  float t = dht.readTemperature();
  // Read temperature as Celsius (the default)
  float f = dht.readTemperature(true);
  // Read temperature as Fahrenheit (isFahrenheit = true)
  // Check if any reads failed and exit early (to try again).
  if (isnan(h) || isnan(t) || isnan(f)) {
    Serial.println("Failed to read from DHT sensor!");
    return;
  }

  // Compute heat index in Fahrenheit (the default)
  float hif = dht.computeHeatIndex(f, h);
  // Compute heat index in Celsius (isFahreheit = false)
  float hic = dht.computeHeatIndex(t, h, false);
  Serial.print ("Humidity: ");
  Serial.print (h);
  Serial.print (" %\t");
  Serial.print ("Temperature: ");
  Serial.print (t);
  Serial.print (" *C ");
  Serial.print (f);
  Serial.print (" *F\t");
  Serial.print ("Heat index: ");
  Serial.print (hic);
  Serial.print (" *C ");
  Serial.print (hif);
  Serial.println (" *F");
}
```

Code to Note

DHT22 sensor has four terminals (V_{cc}, DATA, NC, GND), which are connected to the board as follows −

- DATA pin to Arduino pin number 2
- V_{cc} pin to 5 volt of Arduino board
- GND pin to the ground of Arduino board
- We need to connect 10k ohm resistor (pull up resistor) between the DATA and the V_{cc} pin

Once hardware connections are done, you need to add DHT22 library to your Arduino library file as described earlier.

Result

You will see the temperature and humidity display on serial port monitor which is updated every 2 seconds.

23.ARDUINO - WATER DETECTOR / SENSOR

Water sensor brick is designed for water detection, which can be widely used in sensing rainfall, water level, and even liquid leakage.

Connecting a water sensor to an Arduino is a great way to detect a leak, spill, flood, rain, etc. It can be used to detect the presence, the level, the volume and/or the absence of water. While this could be used to remind you to water your plants, there is a better Grove sensor for that. The sensor has an array of exposed traces, which read LOW when water is detected.

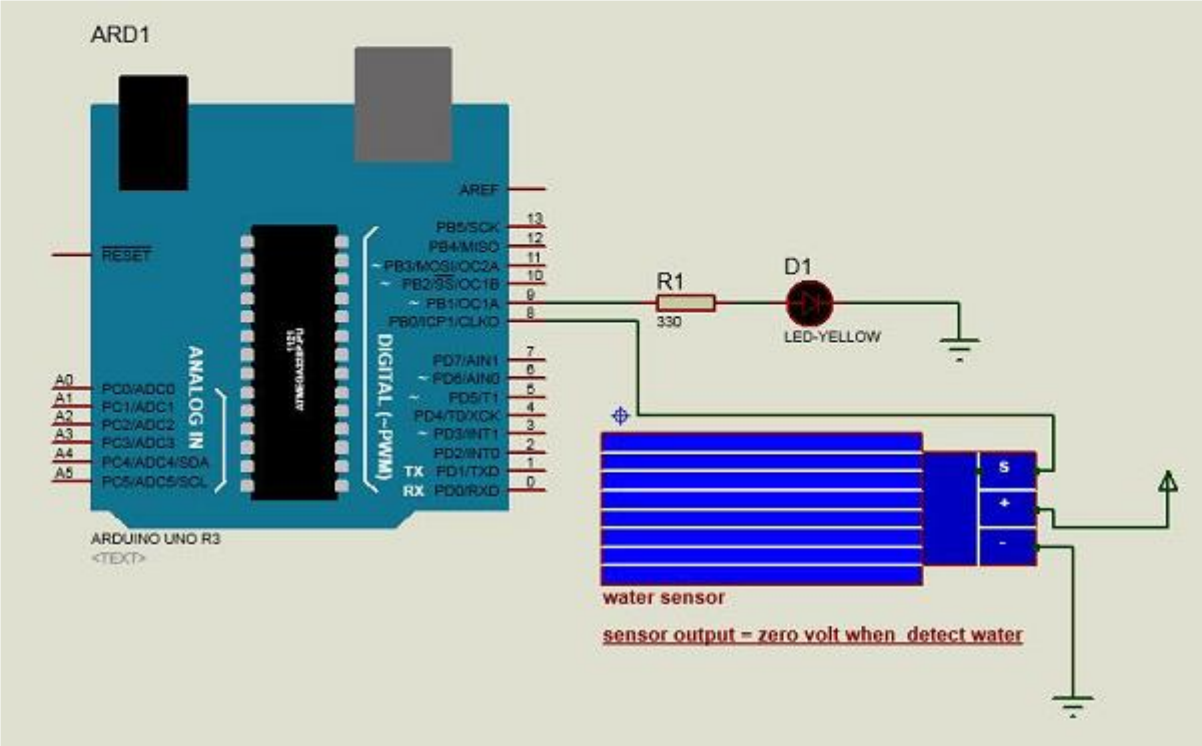
Components Required

You will need the following components −

- 1 × Breadboard
- 1 × Arduino Uno R3
- 1 × Water Sensor
- 1 × led
- 1 × 330 ohm resistor

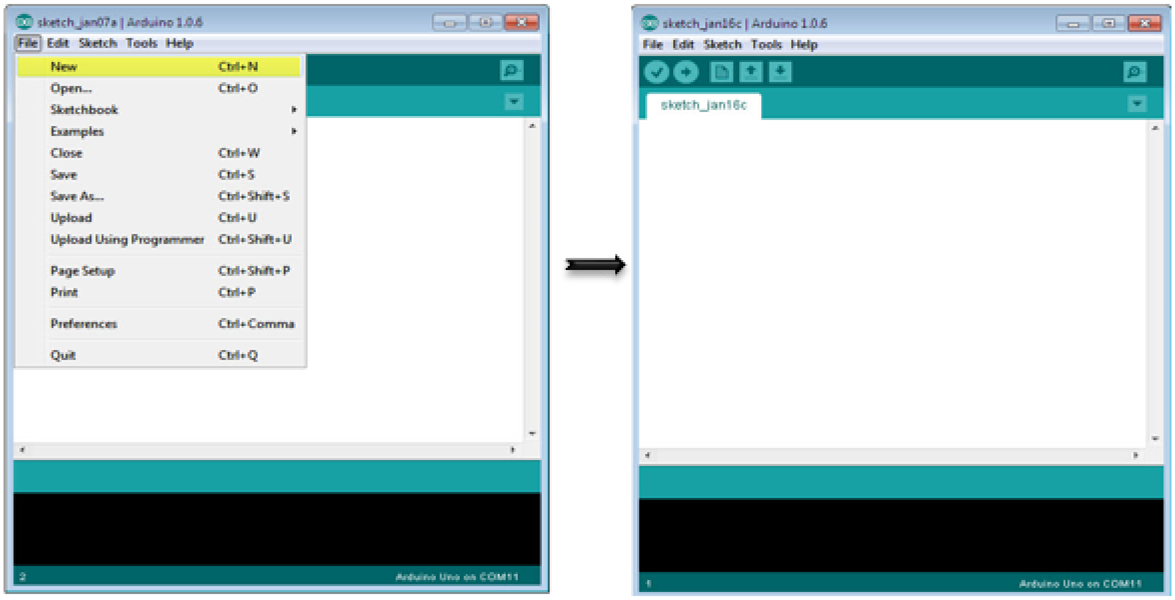
Procedure

Follow the circuit diagram and hook up the components on the breadboard as shown in the image given below.



Sketch

Open the Arduino IDE software on your computer. Coding in the Arduino language will control your circuit. Open a new sketch File by clicking on New.



Arduino Code

```
#define Grove_Water_Sensor 8 // Attach Water sensor to Arduino Digital Pin 8
#define LED 9 // Attach an LED to Digital Pin 9 (or use onboard LED)
```

```
void setup() {
  pinMode(Grove_Water_Sensor, INPUT); // The Water Sensor is an Input
  pinMode(LED, OUTPUT); // The LED is an Output
}

void loop() {
  /* The water sensor will switch LOW when water is detected.
  Get the Arduino to illuminate the LED and activate the buzzer
  when water is detected, and switch both off when no water is present */
  if( digitalRead(Grove_Water_Sensor) == LOW) {
    digitalWrite(LED,HIGH);
  }else {
    digitalWrite(LED,LOW);
  }
}
```

Code to Note

Water sensor has three terminals - S, V_{out}(+), and GND (-). Connect the sensor as follows –

- Connect the +V_s to +5v on your Arduino board.
- Connect S to digital pin number 8 on Arduino board.
- Connect GND with GND on Arduino.
- Connect LED to digital pin number 9 in Arduino board.

When the sensor detects water, pin 8 on Arduino becomes LOW and then the LED on Arduino is turned ON.

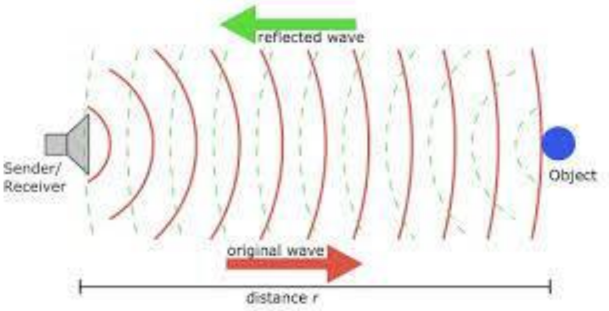
Result

You will see the indication LED turn ON when the sensor detects water.

24.ARDUINO - ULTRASONIC SENSOR

The HC-SR04 ultrasonic sensor uses SONAR to determine the distance of an object just like the bats do. It offers excellent non-contact range detection with high accuracy and stable readings in an easy-to-use package from 2 cm to 400 cm or 1” to 13 feet.

The operation is not affected by sunlight or black material, although acoustically, soft materials like cloth can be difficult to detect. It comes complete with ultrasonic transmitter and receiver module.



Technical Specifications

- Power Supply − +5V DC
- Quiescent Current − <2mA
- Working Current − 15mA

- Effectual Angle − <15°
- Ranging Distance − 2cm – 400 cm/1″ – 13ft
- Resolution − 0.3 cm
- Measuring Angle − 30 degree

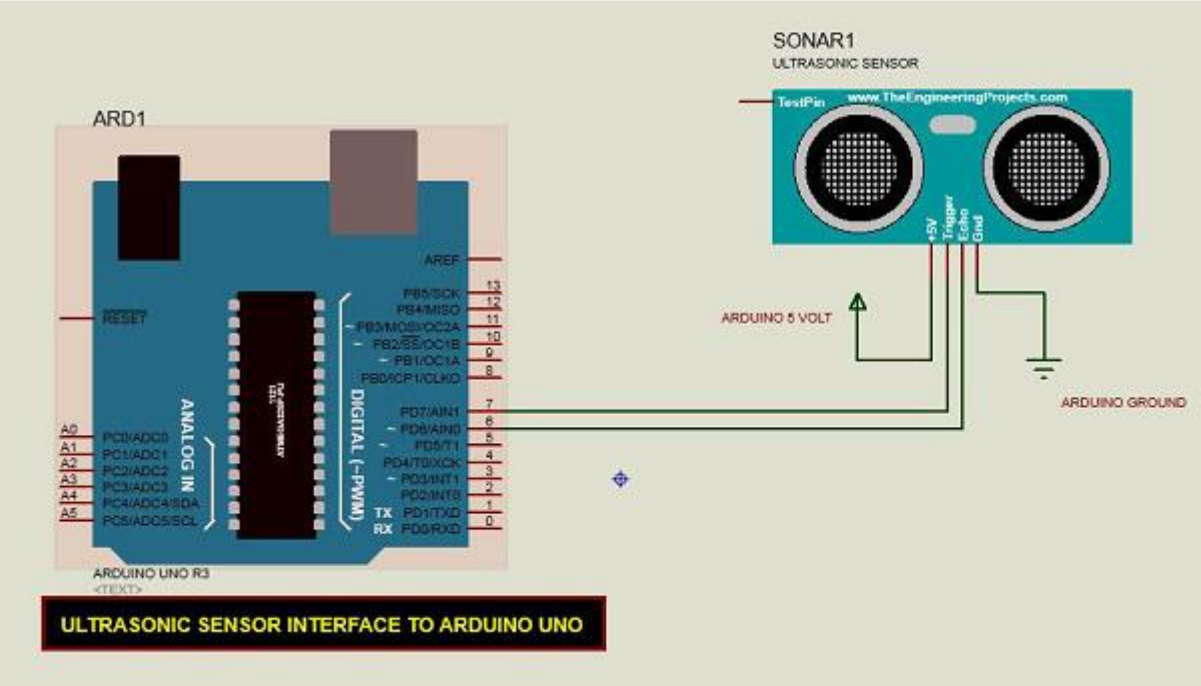
Components Required

You will need the following components −

- 1 × Breadboard
- 1 × Arduino Uno R3
- 1 × ULTRASONIC Sensor (HC-SR04)

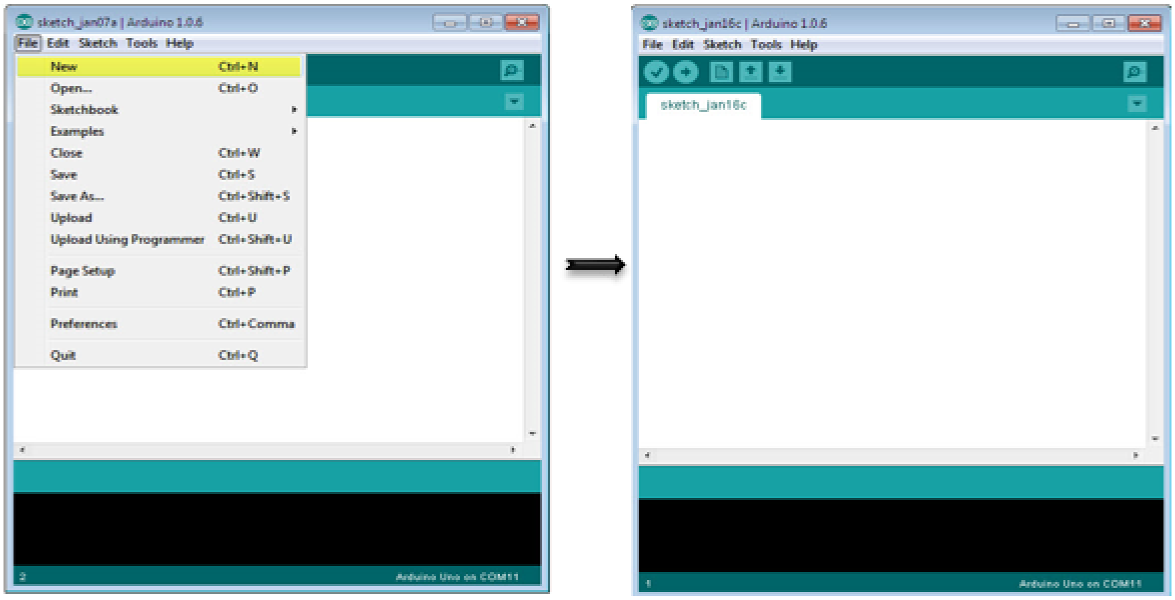
Procedure

Follow the circuit diagram and make the connections as shown in the image given below.



Sketch

Open the Arduino IDE software on your computer. Coding in the Arduino language will control your circuit. Open a new sketch File by clicking New.


sketch_jan07a | Arduino 1.0.6
File Edit Sketch Tools Help
New Ctrl+N
Open... Ctrl+O
Sketchbook
Examples
Close Ctrl+W
Save Ctrl+S
Save As... Ctrl+Shift+S
Upload Ctrl+U
Upload Using Programmer Ctrl+Shift+U
Page Setup Ctrl+Shift+P
Print Ctrl+P
Preferences Ctrl+Comma
Quit Ctrl+Q
2
Arduino Uno on COM11
sketch_jan16c | Arduino 1.0.6
File Edit Sketch Tools Help
sketch_jan16c
1
Arduino Uno on COM11

Arduino Code

```
int distance;
int duration;

void setup() {
 Serial.begin(9600);

pinMode(9,OUTPUT);//trigger pin
pinMode(10,OUTPUT);//echo pin

}

void loop() {
  // put your main code here, to run repeatedly:
digitalWrite(9,LOW);
delay(2);
digitalWrite(9,HIGH);
delay(10);
digitalWrite(10,LOW);
duration=pulseIn(10,HIGH);
distance=duration*0.034/2;
Serial.println(distance);
delay(100);
}
```

Code to Note

The Ultrasonic sensor has four terminals - +5V, Trigger, Echo, and GND connected as follows −

- Connect the +5V pin to +5v on your Arduino board.
- Connect Trigger to digital pin 7 on your Arduino board.
- Connect Echo to digital pin 6 on your Arduino board.
- Connect GND with GND on Arduino.

In our program, we have displayed the distance measured by the sensor in inches and cm via the serial port.

Result

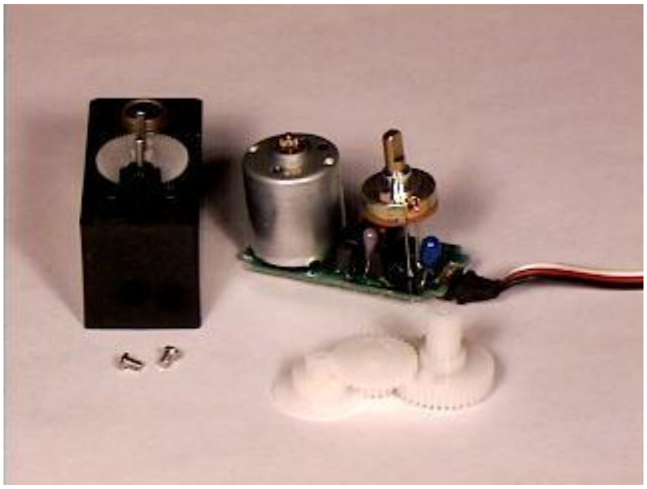
You will see the distance measured by sensor in inches and cm on Arduino serial monitor.

25. ARDUINO - SERVO MOTOR

A Servo Motor is a small device that has an output shaft. This shaft can be positioned to specific angular positions by sending the servo a coded signal. As long as the coded signal exists on the input line, the servo will maintain the angular position of the shaft. If the coded signal changes, the angular position of the shaft changes. In practice, servos are used in radio-controlled airplanes to position control surfaces like the elevators and rudders. They are also used in radio-controlled cars, puppets, and of course, robots.

Servos are extremely useful in robotics. The motors are small, have built-in control circuitry, and are extremely powerful for their size. A standard servo such as the Futaba S-148 has 42 oz/inches of torque, which is strong for its size. It also draws power proportional to the mechanical load. A lightly loaded servo, therefore, does not consume much energy.

The guts of a servo motor is shown in the following picture. You can see the control circuitry, the motor, a set of gears, and the case. You can also see the 3 wires that connect to the outside world. One is for power (+5volts), ground, and the white wire is the control wire.



Working of a Servo Motor

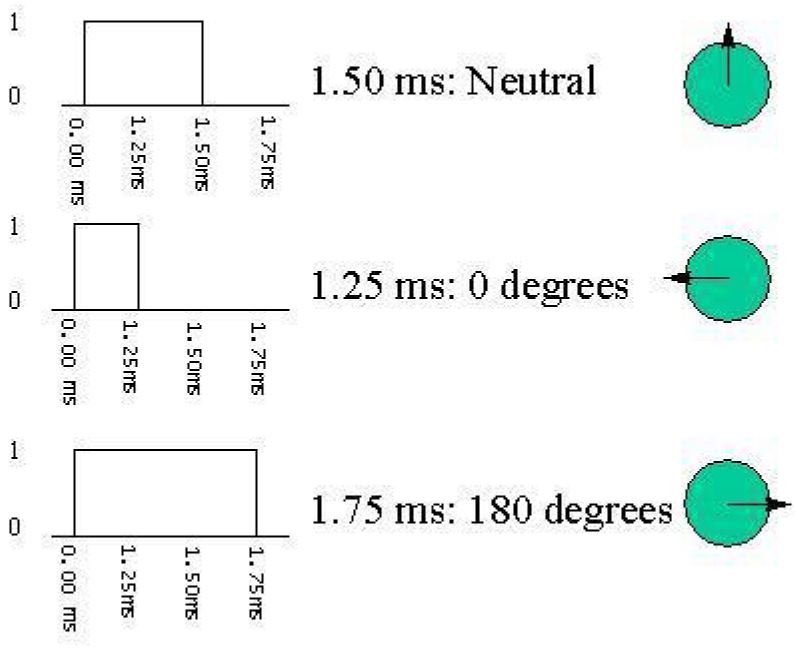
The servo motor has some control circuits and a potentiometer (a variable resistor, aka pot) connected to the output shaft. In the picture above, the pot can be seen on the right side of the circuit board. This pot allows the control circuitry to monitor the current angle of the servo motor.

If the shaft is at the correct angle, then the motor shuts off. If the circuit finds that the angle is not correct, it will turn the motor until it is at a desired angle. The output shaft of the servo is capable of traveling somewhere around 180 degrees. Usually, it is somewhere in the 210-degree range, however, it varies depending on the manufacturer. A normal servo is used to control an angular motion of 0 to 180 degrees. It is mechanically not capable of turning any farther due to a mechanical stop built on to the main output gear.

The power applied to the motor is proportional to the distance it needs to travel. So, if the shaft needs to turn a large distance, the motor will run at full speed. If it needs to turn only a small amount, the motor will run at a slower speed. This is called **proportional control**.

How Do You Communicate the Angle at Which the Servo Should Turn?

The control wire is used to communicate the angle. The angle is determined by the duration of a pulse that is applied to the control wire. This is called **Pulse Coded Modulation**. The servo expects to see a pulse every 20 milliseconds (.02 seconds). The length of the pulse will determine how far the motor turns. A 1.5 millisecond pulse, for example, will make the motor turn to the 90-degree position (often called as the neutral position). If the pulse is shorter than 1.5 milliseconds, then the motor will turn the shaft closer to 0 degrees. If the pulse is longer than 1.5 milliseconds, the shaft turns closer to 180 degrees.



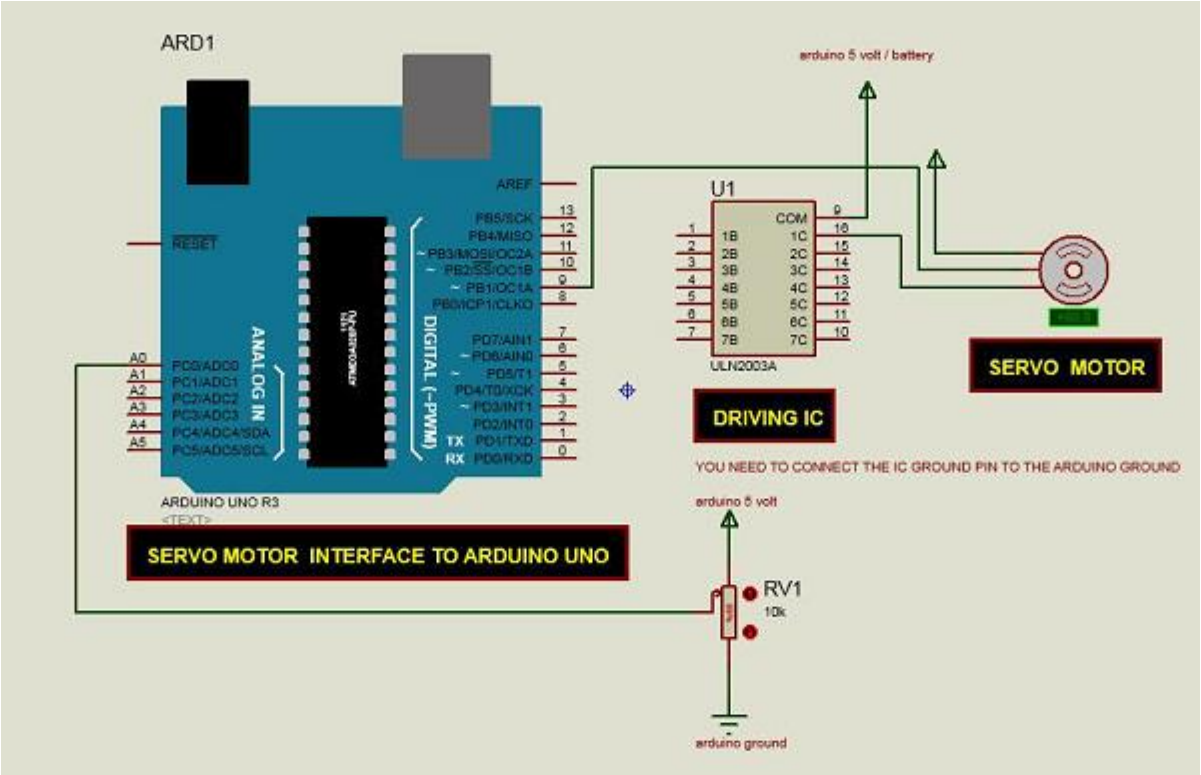
Components Required

You will need the following components −

- 1 × Arduino UNO board
- 1 × Servo Motor
- 1 × ULN2003 driving IC
- 1 × 10 KΩ Resistor

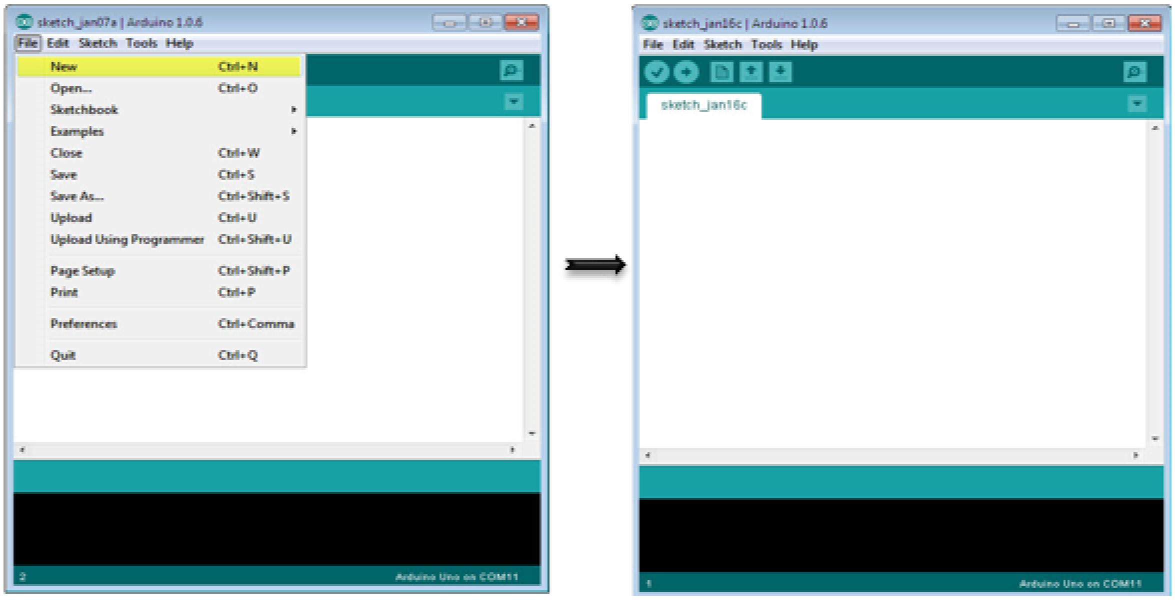
Procedure

Follow the circuit diagram and make the connections as shown in the image given below.



Sketch

Open the Arduino IDE software on your computer. Coding in the Arduino language will control your circuit. Open a new sketch File by clicking on New.



Arduino Code

```
/* Controlling a servo position using a potentiometer (variable
resistor) */
```

```
#include <Servo.h>
   Servo myservo; // create servo object to control a servo
   int potpin = 0; // analog pin used to connect the potentiometer
   int val; // variable to read the value from the analog pin

void setup() {
   myservo.attach(9); // attaches the servo on pin 9 to the servo
object
}

void loop() {
   val = analogRead(potpin);
   // reads the value of the potentiometer (value between 0 and
1023)
   val = map(val, 0, 1023, 0, 180);
   // scale it to use it with the servo (value between 0 and 180)
   myservo.write(val); // sets the servo position according to the
scaled value
   delay(15);
}
```

Code to Note

Servo motors have three terminals - power, ground, and signal. The power wire is typically red, and should be connected to the 5V pin on the Arduino. The ground wire is typically black or brown and should be connected to one terminal of ULN2003 IC (10 - 16). To protect your Arduino board from damage, you will need some driver IC to do that. Here we have used ULN2003 IC to drive the servo motor. The signal pin is typically yellow or orange and should be connected to Arduino pin number 9.