Innovative Experiments Using Open Source for UG Physics Students.

May 2021

Lecture 1: Getting Started with Arduino

Mukesh Kumar, K Basu, P V Rajesh, R Raut, S S Ghugre *UGC DAE CSR, KC*

Note : This manuscript attempts to use the Open Source tools in setting up innovative experiments, for UG & PG Physics course.

Disclaimer: These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside the intended audience only with the permission of the author. The material has been developed following the discussions with our colleagues, during the workshops that have been conducted in collaboration with

- Department of Physics, Asutosh College, Kolkata : November $7^{th} - 8^{th}$, 2017
- Department of Physics & Computer Science, Bethune College, Kolkata, July $19^{th} - 20^{th}$ 2018
- R. P. Gogate College of Arts & Science, Ratnagiri, December $18^{th} - 19^{th}$ 2018
- S. H. Kelkar College, Devgad, December $22^{nd} - 23^{rd}$ 2018
- Department of Physics, Government Holkar Science College, Indore, September $17^{th} - 18^{th}$ 2019
- Department of Physics, R.D & S.H National College, Mumbai, February $6^{th} - 7^{th}$ 2020
- Department of Physics, Victoria Institution (College), Kolkata, July $6^{th} - 10^{th}$ 2020

Contents

[0]Email : ssg@alpha.iuc.res.in & ssg.iuc@gmail.com

1.1 Introduction

UGC-DAE Consortium for Scientific Research, Kolkata Centre, is in the process of developing a range of innovative, low cost experiments based on the routinely available resources for the undergraduates. These experiments are expected to be illustrative and contribute in their understanding of the basics of the subject, apart from rejuvenating the fun factor in the learning process. And all this with an accompanying rigor on the extracted numbers.

For these set of innovative experiments the data acquisition is performed using routinely available resources *viz.*

1. The sound card an integral part of the personal computer.
2. Arduino, an open source microcontroller.

The the data visualisation and analysis is performed using the Open Source toolkits (packages) *viz.* Octave and Python.

1.2 ARDUINO Microcontroller

A micro-controller is self contained system with processor, memory & peripherals combined on a single hardware real estate, which is an open-source physical computing platform based on a simple i/o board and a development environment

Innovative micro-controller based experiments can be developed which would allow for open-ended experiments in the conventional laboratory.

Why ARDUINO ?? The choice of *Arduino* (Fig. 1.1) was essentially due to

1. It being an *Open Source*, both in terms of hardware and software. It is available from several vendors across the country, unlike some of the other kits which are available only through a select vendors, which goes against the ethos of GPL.
2. The hardware is cheap and can also be built from the components using the information available in public domain.
3. It can be powered from either a USB or a standalone DC power
4. It can run standalone from a computer (chip is programmable) and it has a decent memory.
5. It can work with both Digital and Analogue signals, Sensors and Actuators

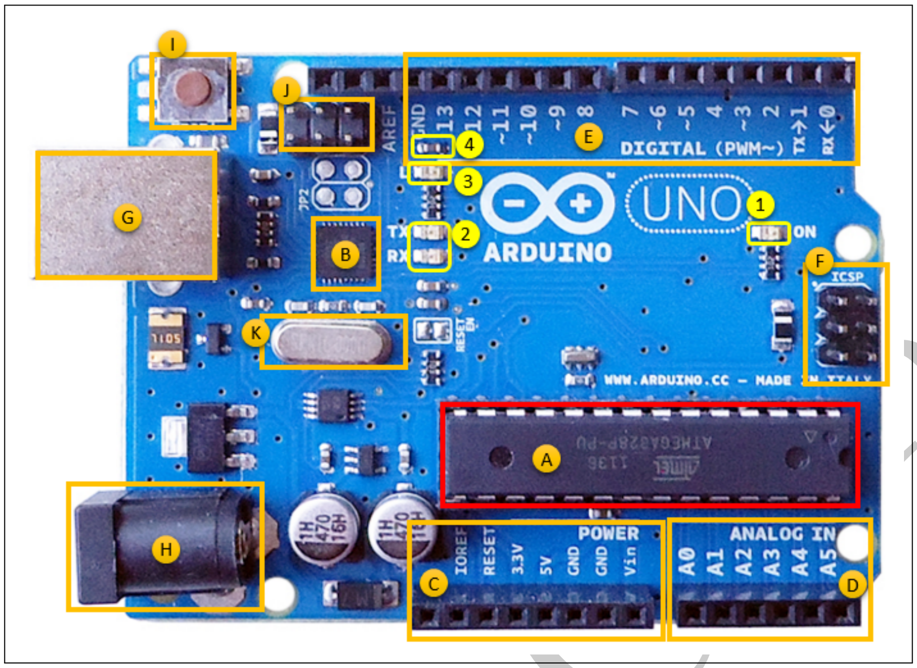


Figure 1.1: Arduino Mircocontroller.

6. It can communicate with the computer via
   (a) Serial , Bluetooth , Ethernet
7. Various shields, *ie.* boards that can be plugged on top on the Arduino or it's variants, that enhance it's capabilities are easily available.
8. Ready made boards such as IR transmitter and receiver, Ultra sonic distance sensor, to name a few are available which can be easily integrated with the Arduino microcontroller, allowing us the flexibility to configure experiments as per user requirement and specifications.

1.3 The Hardware Real Estate

(A) CPU
And at the heart of the Arduino development board is the *Atmel ARV Atmega 328P (A)* , a modified Harvard architecture 8-bit RISC single chip microcontroller which was developed by Atmel in 1996. It can be identified as a **prominent black rectangular chip with 28 pins**. The components of relevance to us are

(i) 32 KB Flash memory, onto which the programs are stored. They survive a power recycle, until over written.
(ii) 2KB of RAM, which is the transparent run time memory.
(iii) CPU, which is the heart or brain of the microcontroller.

(iv) A non volatile Electrically Erasable Programmable Read Only Memory (EEPROM) of 1KB which keeps the data even after device restart and reset.

(B) USB Interface Chip
The ATMega 16U2 programmable USB to UART (Universal Asynchronous Receiver-Transmitter : variable speed for serial communication), converts signals in the USB level to a level compatible with the Arduino UNO board.

(C) Power pins
Power Pins the 5v & 3v3 provide 5 and 3.3 volt dc supply to external components. The GND pins are used to close the electrical circuit and provide a common logic reference level throughout the circuit. The external circuit and the Arduino board should have a **Common ground**.

(D) Analog pins
The Arduino Uno has 6 analog pins, which utilize a **10 bit** ADC (Analog to Digital converter). They can accept a **maximum** voltage of 5 volt. They are labelled as Pins A0-A5, These pins just measure voltage and not the current because they have very high internal resistance. Hence, only a small amount of current flows through these pins.

(E) Digital pins
Pins labelled 0-13 of the Arduino Uno serve as digital input/output pins. Pin 13 of the Arduino Uno is connected to the built-in LED.
When digital pins are used as output pins, they supply 40 milliamps of current at 5 volts, however, it is recommended to limit the current to 20 milliamps.
In the Arduino Uno - pins 3,5,6,9,10,11 labelled by ( ) have Pule Width Modulation capability which simulate an analog like output.

(F) ICSP header ICSP stands for In-Circuit Serial Programming, which at times are used to program the microcontroller using dedicated programming device.

(G) USB Connector This is a printer USB port used to load a program from the Arduino IDE onto the Arduino board. The board can also be powered through this port.

(H) Power Port The Arduino board can be powered through an AC-to-DC adapter or a battery using a conventional **2.1mm center-positive plug** into the power jack of the board.

(I) Reset Switch A short press would restart the program, whereas a long press would reset it to factory settings.

(J)

(K) Crystal Oscillator This is a quartz crystal oscillator which ticks 16 million times a second. On each tick, the microcontroller performs one operation, for example, addition, subtraction, to name a few.

1. Power ON LED Indicates that the board has been powered ON.

2. TX / RX LED **TX** stands for transmit, and **RX** for receive. These are indicator LEDs which blink whenever the UNO board is transmitting or receiving data *ie* the board is communicating with the host computer.
3. SMD Components
4. On Board LED The on board LED is connected to digital pin-13.

1.3.1 Variants of the Arduino

The Arduino micro-controller comes in several variants, some of the most popular being

1. Uno
2. Mega
3. Nano

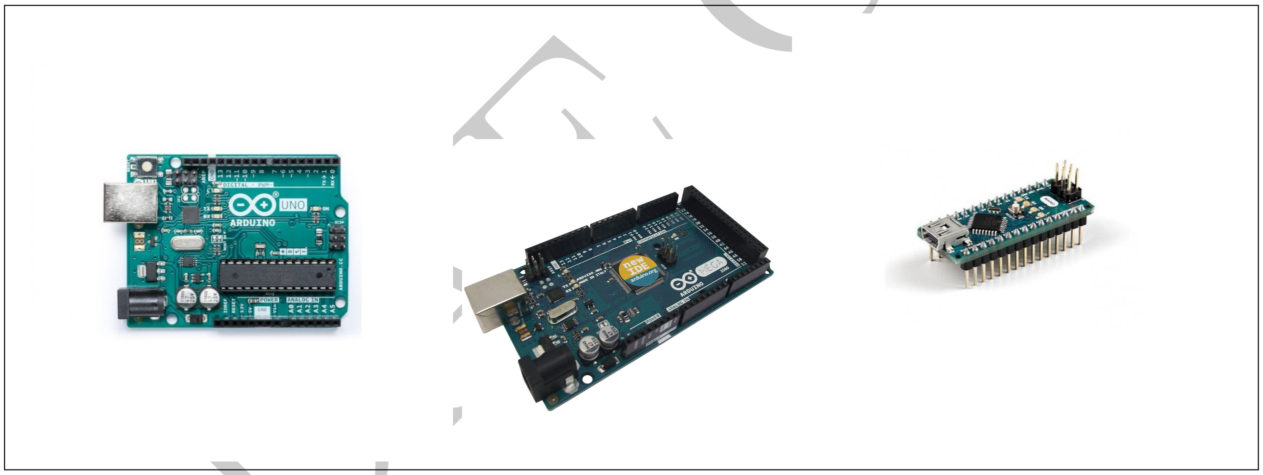


Figure 1.2: Arduino Uno, Mega and Nano .

Original Nano boards are getting rare, and one usually ends up with a clone. **If so the device drivers for the communication chip have to be manually updated before one can use these boards**. The present manuscript utilizes the **Arduino Uno** boards. No support can be extended to the **Nano Clones**.

1.3.2 Essential Accessories

To effectively use the Arduino, we require the following

1. USB printer cable, Fig.1.3

2. Breadboards
   (a) **Mini Breadboard**
   (b) **Small Breadboard**
3. Breadboard Power supply, this has to be used with a **small breadboard**.
4. Connectors
5. Breakout Boards
6. Components

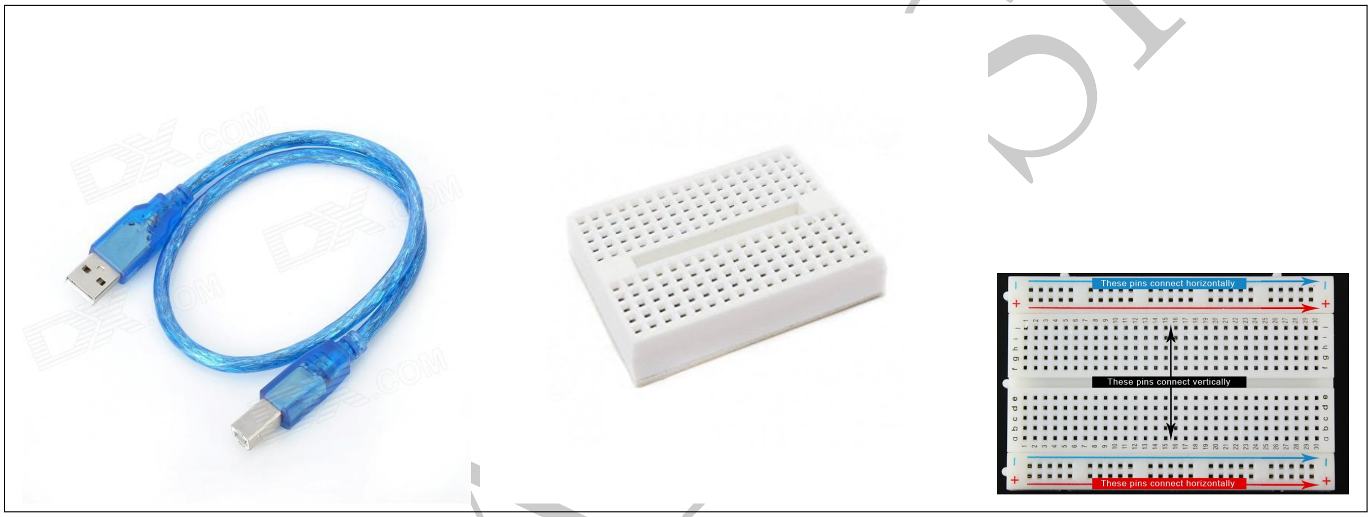

These pins connect horizontally
These pins connect vertically
These pins connect horizontally

Figure 1.3: USB cable, Mini Breadboard and Small Breadboard .

Breakout boards or prototype shields (Fig.1.6), are also available which help us integrate the boards with external electrical circuitry for our specific experiments.

1.4 Arduino & Computer

The Arduino Software (Integrated Development Environment) allows us to write programs and upload them on to the board, in a user friendly and transparent manner.

The program for Arduino are referred to as Sketch, and is a **cousin** of the familiar C programming language.

The first step in programming the Arduino board is downloading and installing the Arduino IDE. The open source Arduino IDE is available for Windows, Mac OS X, and Linux.

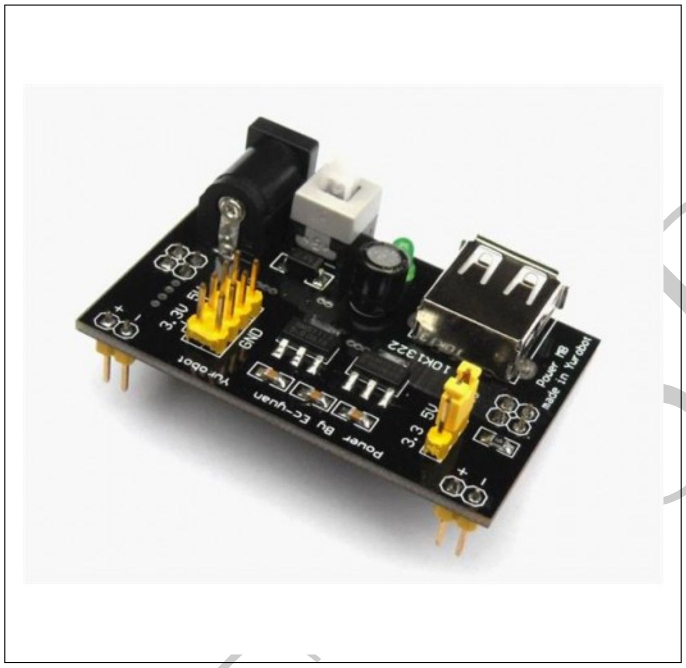


Figure 1.4: Breadboard power supply .

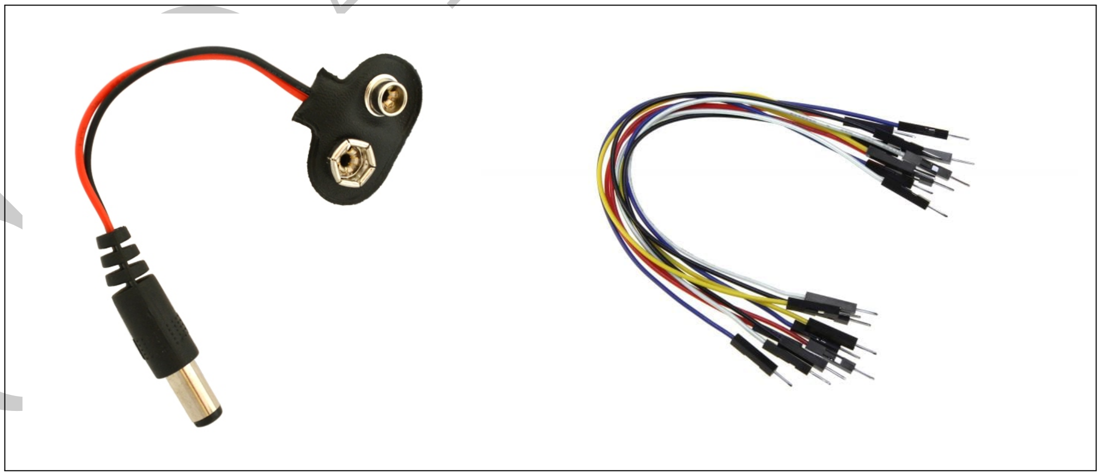


Figure 1.5: 9V snap connector and Single strand connectors .

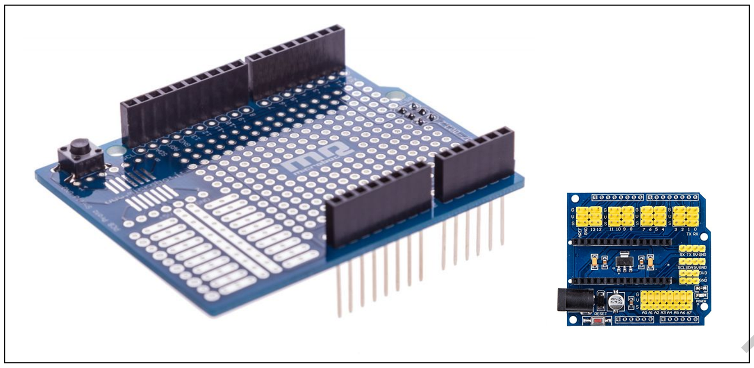
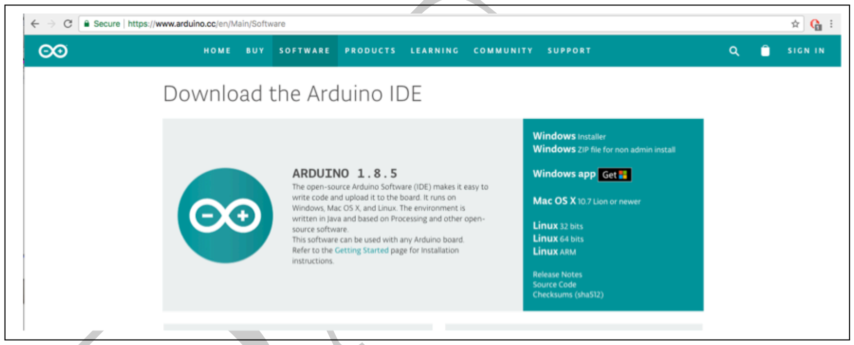


Figure 1.6: Prototype expansion boards for Arduino Uno, and Nano .

The details for getting started with Arduino on Windows is provided at

http://arduino.cc/en/Guide/Windows



Download the Arduino IDE

ARDUINO 1.8.5

The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. It runs on Windows, Mac OS X, and Linux. The environment is written in Java and based on Processing and other open-source software.
This software can be used with any Arduino board.
Refer to the Getting Started page for Installation instructions.

Windows Installer
Windows ZIP file for non admin install
Windows app Get
Mac OS X 10.7 Lion or newer
Linux 32 bits
Linux 64 bits
Linux ARM
Release Notes
Source Code
Checksums (sha512)

Figure 1.7: Download the IDE from arduino.cc/en/Main/Software.

1.4.1 Initial Configuration of IDE

Now all we need to do is connect the microcontroller to the system via a USB (Fig.1.8) and we are all set.

We need to configure the development environment, by providing, the board details as well as the port (Fig.1.9) to which it is connected. Most of the time the software will prompt you with the correct options :

Now, we are all set to venture into the world of microcontrollers.

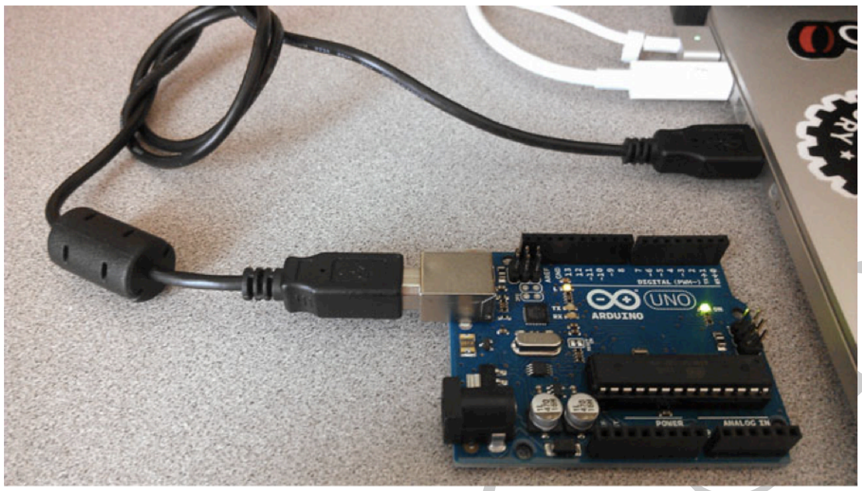
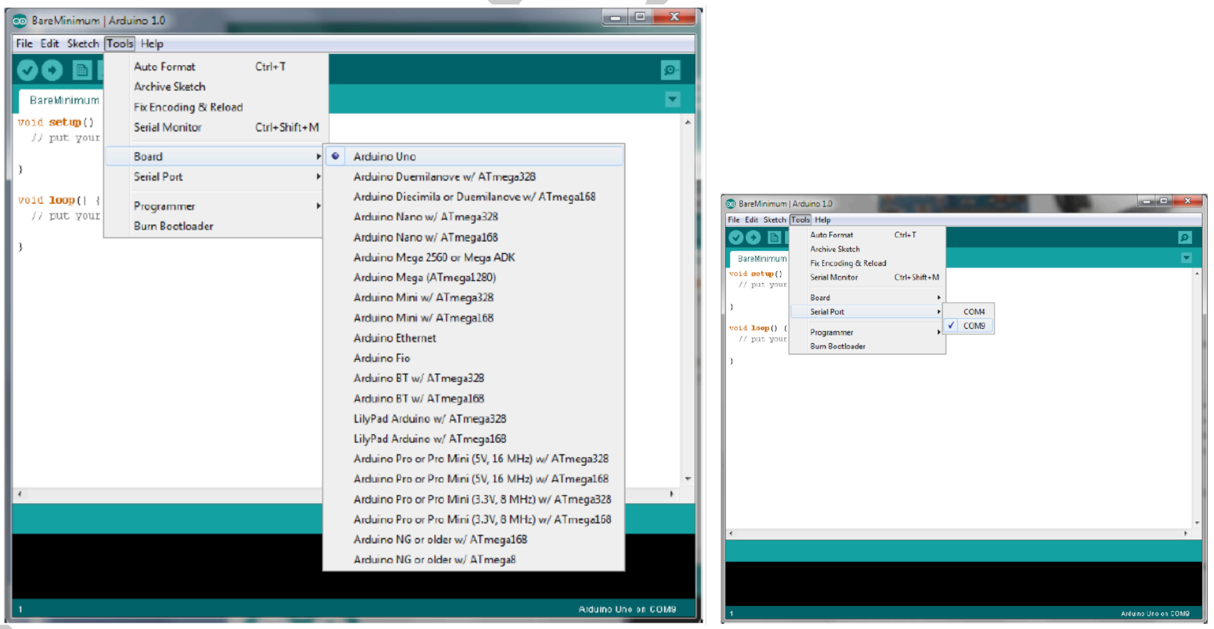


Figure 1.8: Connect the Arduino to the host computer.


BareMinimum | Arduino 1.0
File Edit Sketch Tools Help
BareMinimum
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Board
Serial Port
Programmer
Burn Bootloader
● 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 Mini w/ ATmega328
Arduino Mini w/ ATmega168
Arduino Ethernet
Arduino Fio
Arduino BT w/ ATmega328
Arduino BT w/ ATmega168
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
COM4
✓ COM9
Arduino Uno on COM9

```
void setup() {
  // put your
}

void loop() {
  // put your
}
```

Figure 1.9: Selection of the Board and Com port.

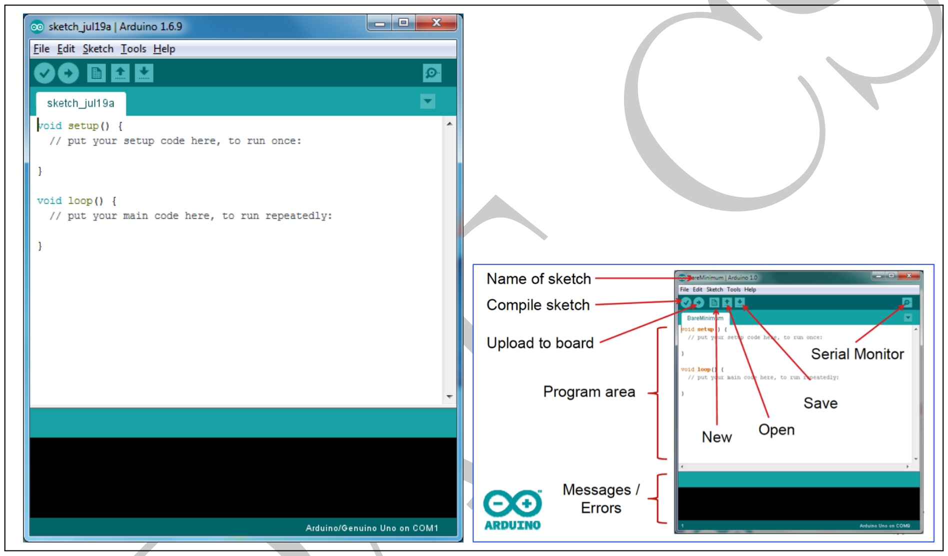
1.5 First Progam : Blink LED

A program in Arduino is referred to as *Sketch*.

The anatomy of a sketch is

void setup() $\rightarrow$ initialization
executed only when the program begins
void loop() $\rightarrow$ code executed continously

On activating (double click the Arduino icon on the desktop), we would be presented with the following window (Fig.1.10) :



```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Figure 1.10: Start writing your sketch.

The summary of the various buttons on the screen are enumerated in the figure above.

Having done this we are all set to write our first program for the micro-controller which is the equivalent of **Hello World**, the first program conventionally written in any language.

We know that we have an on-board LED connected to digital pin number 13, and if we were sequentially make the pin HIGH and LOW, we would observe the LED to blink.

The code which causes the LEB to blink is

```
 1
 2 // the setup function runs once when you press reset or power the board
 3 void setup() {
 4   // initialize digital pin LED_BUILTIN as an output.
 5   pinMode(LED_BUILTIN, OUTPUT);
 6 }
 7
 8 // the loop function runs over and over again forever
 9 void loop() {
10   digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
11   delay(1000);                       // wait for a second
12   digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
13   delay(1000);                       // wait for a second
14 }
```

Blink On Board LED

In the initialization routine, we define (Line -4) the digital pin number 13 as an **output** pin. Here, we have taken recourse to the built-in *constant* , LED_BUILTIN , which refers to digital pin number 13.

In the **main loop** of the program, we first set this pin HIGH Line -10) then we wait for a *preset* time delay(1000), where the argument is in **milli-seconds**.

Following this delay, we then set this pin to a LOW state, and again wait for $1000 milli - second$ (Line -12-13)

Now, we are ready to Compile the code and then Upload the code to the microcontroller.

The procedure to transfer this code onto the Arduino is as follows (Fig.1.11) :

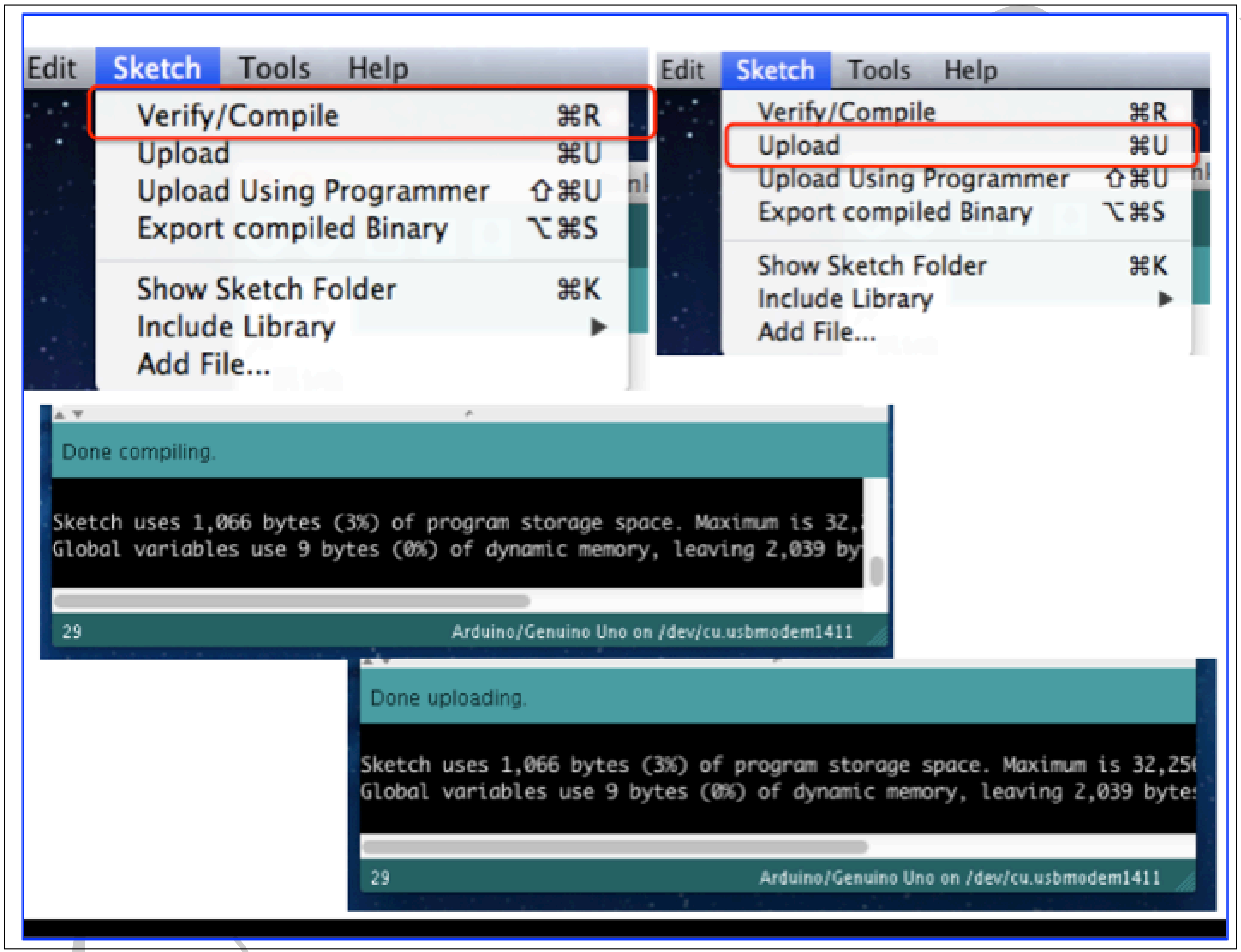
1. The sketch, which is case sensitive, is typed in the program area.

2. Press the Compile button, for compilation and any errors are identified in this process and need to be rectified before the code is compiled.

3. Press the *Upload* button, to program the Arduino board with the sketch.

4. During the uploading, the *TX/RX* LED will flash, indicating a communication between the Arduino and the PC.

Once the code gets uploaded on to micro-controller, the on-board LED which is connected to *pin 13* will blink (Fig.1.12). The blinking rate can be controlled by modifying the option parameter for *delay*. The parameter is set in milli-seconds. Play around with value and observe when the LED appears to have a constant illumination, which will be due to **persistence of vision**.



```
Done compiling.

Sketch uses 1,066 bytes (3%) of program storage space. Maximum is 32,
Global variables use 9 bytes (0%) of dynamic memory, leaving 2,039 by

29                                  Arduino/Genuino Uno on /dev/cu.usbmodem1411
```

```
Done uploading.

Sketch uses 1,066 bytes (3%) of program storage space. Maximum is 32,25
Global variables use 9 bytes (0%) of dynamic memory, leaving 2,039 byte

29                                  Arduino/Genuino Uno on /dev/cu.usbmodem1411
```

Figure 1.11: Compile and upload the sketch.

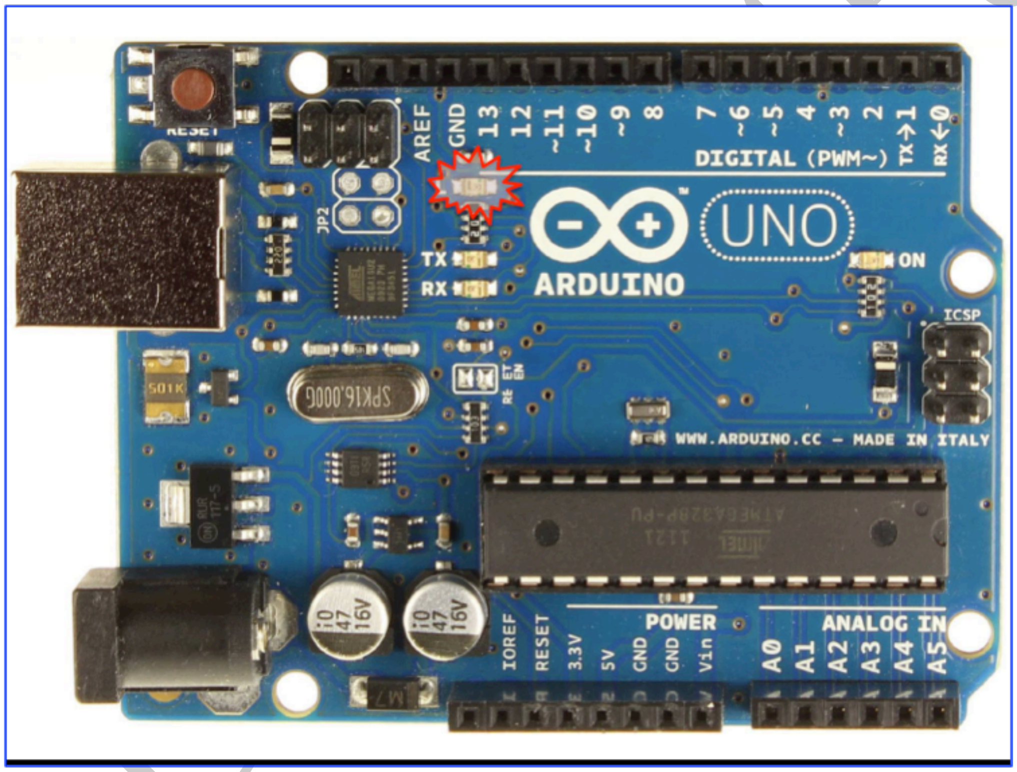


Figure 1.12: Blinking the on-board LED.

1.6 Record Analog Voltage

1.6.1 ADC on ARDUINO

An **A**nalog-to- **D**igital **C**onvertor, converts the analog voltage which is provided as input into an equivalent digital number.

We classify (specify) the given ADC based upon the number of bits.

For example if we have a *10* bit ADC (present onboard in Arduino), then it will map V_{min} to $V_{max} \rightarrow 2^{10} = 1024$ digital numbers.

Arduino accepts analog signals up to 5 volt and hence has a resolution of about 5 milli-Volt.

We know that, the ADC after converting the input analog voltage would return an integer value (channel number).

The inbuilt function analogRead(pin_number), would help us obtain the ADC value for the given input analog voltage.

Since, the ADC channel number is an integer, the variable which stores this value has to be declared as **integer**.

```
int SensorValue = analogRead(A0);
```

The above command, would store the ADC channel number (as an integer) corresponding to the analog voltage connected to Analog Pin Number 0.

If we were to connect the onboard 5V supply to this pin, the variable SensorValue would have a value of 1023, where as if we were to connect the 3.3V supply it would have a value of 673

Now, the value obtained would have to be displayed to the user. A simple solution would be to use the Serial Monitor for this purpose.

Hence, in the initialization routine we need to set up the communication with the Serial Monitor. The parameter of relevance would be the baud rate, the rate at which the data is communicated. The command below sets the value at 9600 bits per second.

```
Serial.begin(9600)
```

```
// the setup routine runs once when you press reset://‘

void setup() {
// initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
}

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

void loop() {
// read the input on analog pin 0:
  int sensorValue = analogRead(A0);
// print out the value you read:
  Serial.println(sensorValue);
// delay in the milli-seconds in between reads for stability
  delay(1000);        // delay in between reads for stability
}
```

Read Analog Value

Once the program is compiled and uploaded you can view the results in the Serial Monitor, the commands of relevance are shown below, and the results are presented in the Fig.1.13.

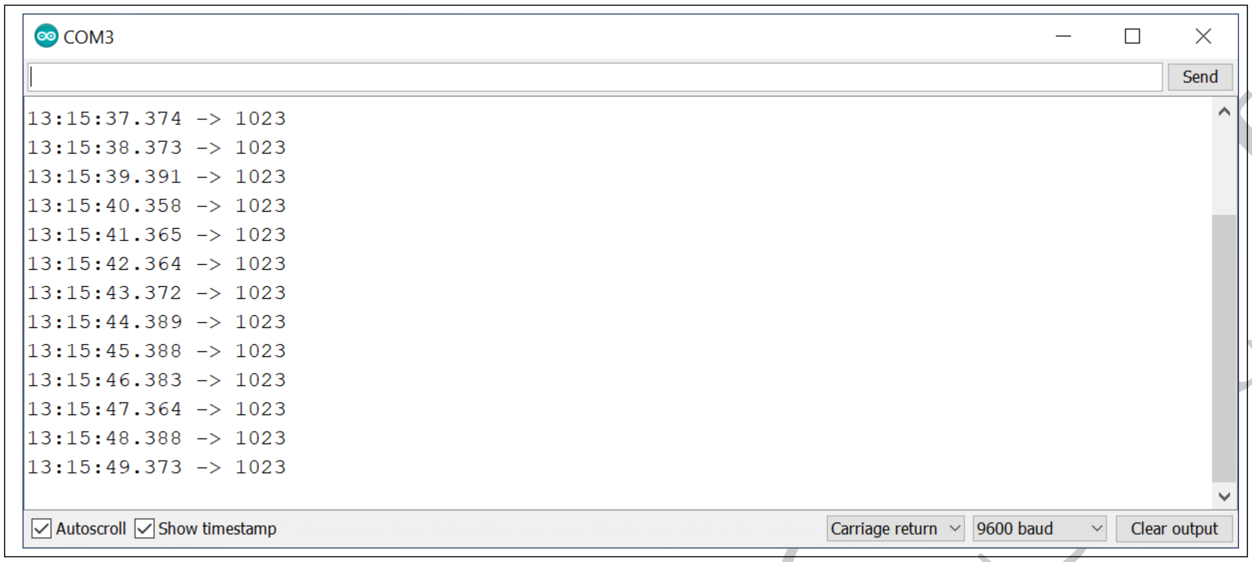
$$\begin{tikzcd}
\text{Tools} \arrow[r] & \text{Serial Monitor} \\
\arrow[r] & \text{Autoscroll} \\
\arrow[r] & \text{Show timestamp} \\
\arrow[r] & \text{Carriage return} \\
\arrow[r] & \text{9600 baud}
\end{tikzcd}$$

Since we have used the on-board 5V supply, we have the ADC channel number as 1023.

1.6.2 Read Analog Voltage

We know that the ADC converts the input into a channel number, and we then need to map it or convert it to the quantity of our relevance. This is referred to as Calibration.

This is possible since **ADC** reports a ratiometric value *ie*, there is a direct one-is-to-one correlation between the input value and the digitized ADC channel number.



```
13:15:37.374 -> 1023
13:15:38.373 -> 1023
13:15:39.391 -> 1023
13:15:40.358 -> 1023
13:15:41.365 -> 1023
13:15:42.364 -> 1023
13:15:43.372 -> 1023
13:15:44.389 -> 1023
13:15:45.388 -> 1023
13:15:46.383 -> 1023
13:15:47.364 -> 1023
13:15:48.388 -> 1023
13:15:49.373 -> 1023
```

Figure 1.13: View the results on the Serial Monitor.

For example, using the **on-board 10 bit ADC**, we have the following relation

$$\begin{aligned} 0\ V &\longrightarrow 0 \text{ channel} \\ 5\ V &\longrightarrow 1023 \text{ channel} \\ \frac{volt}{channel} &= \frac{5}{1023} \\ \text{unknown voltage} &= \frac{5}{1023} \times \text{channel} \end{aligned}$$

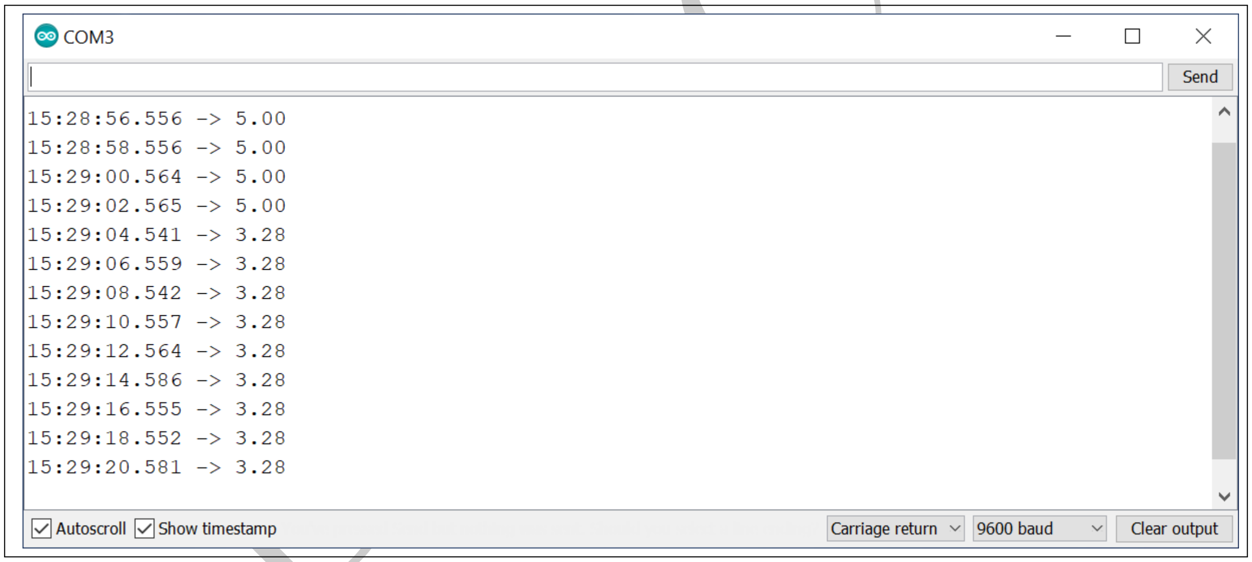
Offcourse we have to bear in mind, that while the ADC channel is an integer, the mapped voltage would be a real number.

```
// read the input on analog pin 0:
int sensorValue = analogRead(A0);

// Convert the analog reading (which goes from 0 - 1023) to a voltage (0 - 5V):
float voltage = sensorValue * (5.0 / 1023.0);
```

The code below allows us to measure a voltage using the **Analog** input pin, which then is displayed on the Serial Monitor (Fig.1.14). Offcourse, we have to bear in mind **this voltage cannot exceed 5 volt**.

```
// the setup routine runs once when you press reset:
void setup() {
// initialize serial communication at 9600 bits per second:
Serial.begin(9600);
}

// the loop routine runs over and over again forever:
void loop() {
// read the input on analog pin 0:
int sensorValue = analogRead(A0);
// Convert the analog reading (which goes from 0 - 1023) to a voltage (0 - 5V):
float voltage = sensorValue * (5.0 / 1023.0);
// print out the value you read:
Serial.println(voltage);
delay(1000);
}
```

Read Analog Voltage



```
15:28:56.556 -> 5.00
15:28:58.556 -> 5.00
15:29:00.564 -> 5.00
15:29:02.565 -> 5.00
15:29:04.541 -> 3.28
15:29:06.559 -> 3.28
15:29:08.542 -> 3.28
15:29:10.557 -> 3.28
15:29:12.564 -> 3.28
15:29:14.586 -> 3.28
15:29:16.555 -> 3.28
15:29:18.552 -> 3.28
15:29:20.581 -> 3.28
```

Figure 1.14: Measured voltage on the Serial Monitor.

1.6.3 Time Stamped Data

Many a times, we may have to record, a time stamped data, *ie*, we need to record the time along with the data. However, we are not interested in the **absolute** time, but we need the relative time information between the recorded data.

We could use the inbuilt millis() function, which records the number of milliseconds that have elapsed since the program started running. It is expected to overflow after several days. The value returned is an unsigned long. The pseudo code for doing so would be :

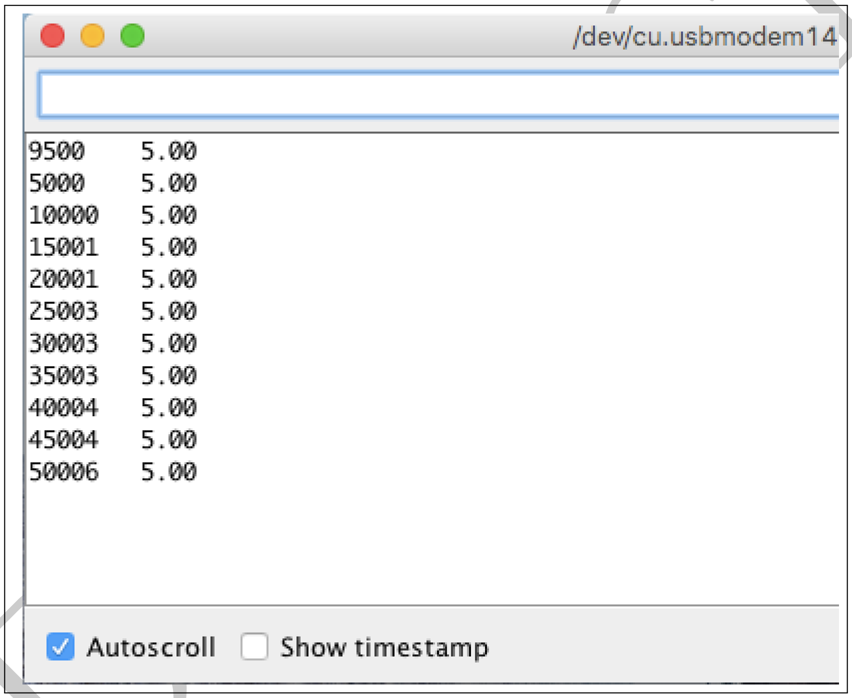
start_tim = millis() in the initialization routine
current_tim = millis() in the main loop
elapsed_tim = current_tim - start_tim in the main loop

The Sketch to achieve is presented blow, and the results are illustrated in Fig.1.15.

```
unsigned long startMillis;
unsigned long currentMillis;
unsigned long now;
void setup() {
// initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
// initialize start time :
  startMillis = millis();
}

// the loop routine runs over and over again forever:
void loop() {
  currentMillis = millis();
  now = currentMillis - startMillis;
  int sensorValue = analogRead(A0);
  float voltage = sensorValue * (5.0 / 1023.0);
  Serial.print(now);
  Serial.print("\t");
  Serial.println(voltage);
  delay(5000);
}
```

Time Stamped Data

The first few readings (maximum two-three), may have to be ignored due to settling down time, which can be circumvented by a *delay(1000)* command in the initialization routine.



```
9500    5.00
5000    5.00
10000   5.00
15001   5.00
20001   5.00
25003   5.00
30003   5.00
35003   5.00
40004   5.00
45004   5.00
50006   5.00
```

Figure 1.15: Time Stamped Data on the Serial Monitor.

1.7 Temperature Measurement using LM35

1.7.1 LM35 Temperature Sensor

The LM35 is precision integrated-circuit temperature device whose output voltage is linearly proportional to the Centigrade temperature. It has a fairly linear calibration of $\sim 10\ mV/^0C$. The pin configuration is depicted in Fig. 1.16.

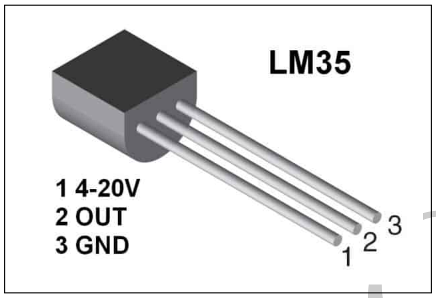


Figure 1.16: Pin Configuration for the LM 35.

1.7.2 Circuit Diagram

The LM35 can derive the power directly from the Arduino, and the output pin is connected to one of the analog input pins, Fig. 1.17.

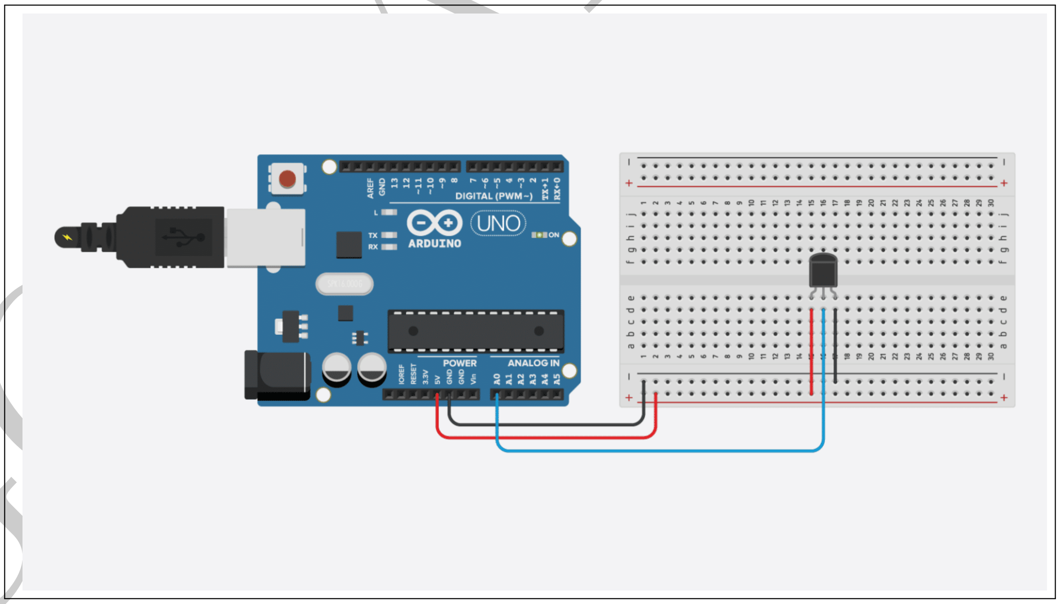


Figure 1.17: Connecting the LM 35 to the Arduino.

1.7.3 Sketch for temperature measurement

The pseudo-code to perform a temperature measurement with LM35 is :

```
pinMode(PWM_out_pin, OUTPUT);
// Define one of the Analog pins as the INPUT pin in the initialization routine
#define sensorPin A0;

// Read the ADC value for the pre defined input pin
int reading = analogRead(sensorPin);

// Convert the reading into voltage:
float voltage = reading * (5000 / 1024.0);

// Convert the voltage into the temperature in degree Celsius:
float temperature = voltage / 10;
```

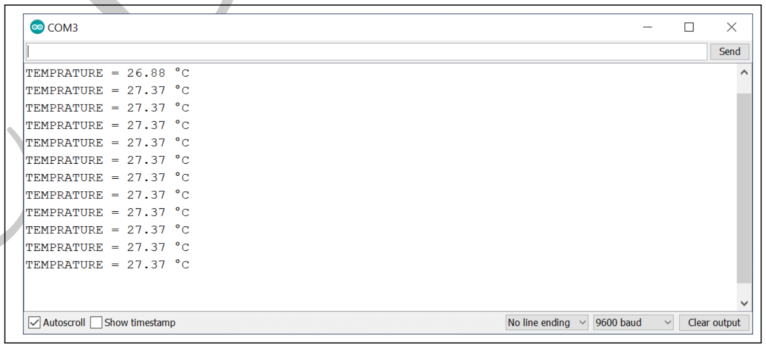
The code below (Ref. https://www.makerguides.com) allows us to measure the temperature using LM35 and display the results on the Serial Monitor.

```
// Define to which pin of the Arduino the output of the LM35 is connected:
#define sensorPin A0

void setup() {
  // Begin serial communication at a baud rate of 9600:
  Serial.begin(9600);
}

void loop() {
  // Get a reading from the temperature sensor:
  int reading = analogRead(sensorPin);

  // Convert the reading into voltage:
  float voltage = reading * (5000 / 1023.0);

  // Convert the voltage into the temperature in degree Celsius:
  float temperature = voltage / 10;

  // Print the temperature in the Serial Monitor:
  Serial.print(temperature);
  Serial.print(" \xC2\xB0"); // shows degree symbol
  Serial.println("C");

  delay(1000); // wait a second between readings
}
```

Temperature Measurement with LM35

1.7.4 Results

The results are illustrated pictorially in Fig. 1.18, which are consistent with the results of measurements performed using other temperature recording equippment.



```
TEMPRATURE = 26.88 °C
TEMPRATURE = 27.37 °C
TEMPRATURE = 27.37 °C
TEMPRATURE = 27.37 °C
TEMPRATURE = 27.37 °C
TEMPRATURE = 27.37 °C
TEMPRATURE = 27.37 °C
TEMPRATURE = 27.37 °C
TEMPRATURE = 27.37 °C
TEMPRATURE = 27.37 °C
TEMPRATURE = 27.37 °C
TEMPRATURE = 27.37 °C
```

Figure 1.18: Display of the Temperarure on the Serial Monitor.

1.8 Temperature Measurement Using DHT 11

1.8.1 Introduction

The DHT11 module has an inbuilt temperature (a negative temperature coefficient thermistor) & humidity sensor module. The onboard components measures and processes the analog signal with stored calibration coefficients, and converts the analog signal to the corresponding digital value and outputs the signal which encodes the temperature and humidity.The shield has all the necessary components, so that we can integrate it with the Arduino, without any additional components. The pin details are presented in Fig. 1.19.

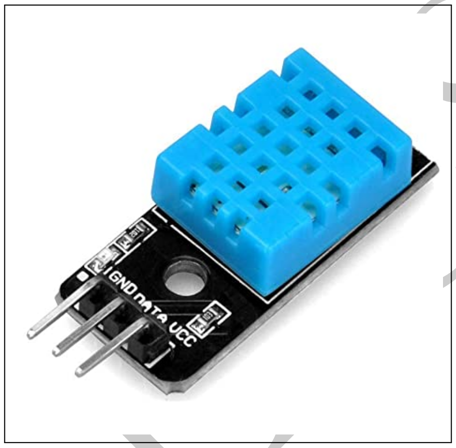


Figure 1.19: Pin configuration of the DHT11 module (The configuration is vendor dependent.)

1.8.2 Installing the DHT11 Library

The DHT11 has a failt simple connection :

$$\begin{aligned} Vcc &\longrightarrow + \ 5\ V \text{ from Arduino} \\ Gnd &\longrightarrow \text{Ground from Arduino} \\ Data &\longrightarrow \text{Pin 8} \end{aligned}$$

The connections are illustrated in Fig. 1.20.

1.8.3 Installing the DHTLib

DHT11 sensors have their own single wire protocol for transferring the data. This protocol requires precise timing. Fortunately, DHT Library was written to hide away all the complexities so that the various operations are transparent to the user, who could issue the commands to read the temperature and humidity data, without getting caught in the protocols.

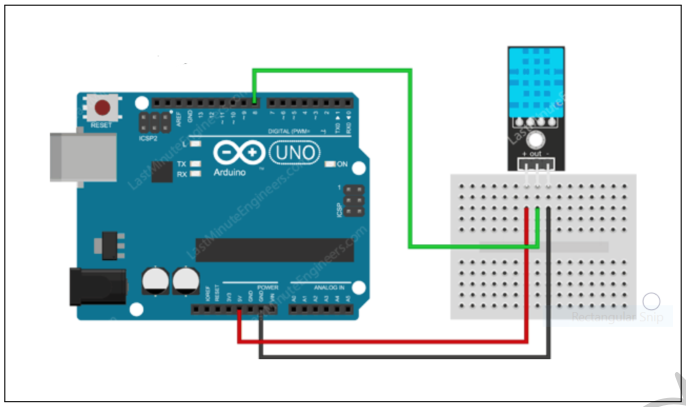
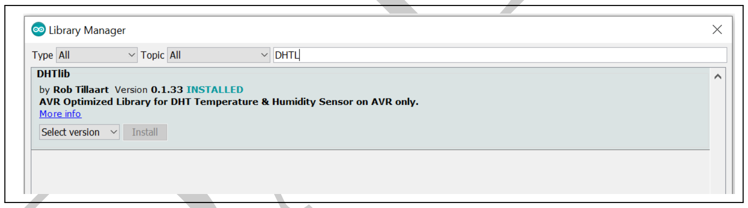


Figure 1.20: Connecting the DTH11 module to the Arduino.

The procedure to add a library is to use the inbuilt library manager, which can be activated, via the Tools ⟶ Manage Libraries, and type DHTLib, and if the library is not installed it would prompt you to install the same. This is presented in Fig. 1.21.


Library Manager

Type All Topic All DHTL

DHTlib
by **Rob Tillaart** Version **0.1.33** INSTALLED
AVR Optimized Library for DHT Temperature & Humidity Sensor on AVR only.
More info

Figure 1.21: Installing the library for the DTH11 module.

1.8.4 Sketch for DHT11

```
#include <dht.h>

dht DHT;

#define DHT11_PIN 8

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

void loop(){
  int chk = DHT.read11(DHT11_PIN);
  Serial.print("Temperature = ");
  Serial.println(DHT.temperature);
  Serial.print("Humidity = ");
  Serial.println(DHT.humidity);
  delay(1000);
}
```

Temperature Measurement with DHT 11

1.9 Temperature Measurement Using Thermistor

1.9.1 Thermistors

Since we need to automate the temperature measurement, we would be using a combination of

- Negative Temperature Coefficient (NTC) thermistors. These are essentially resistors with a negative temperature coefficient, *ie.* following an **increase** in the *temperature* correspondingly the *resistance* **decreases**.

- Microcontroller However, a microcontroller does not have a built-in provision / facility of an *ohmmeter*. It can only read in an *Analog voltage* due to the inbuilt Analog-Digital-Converter. Hence, we need to employ a circuit wherein the **measured voltage** is related to the **resistance**. This is achieved using the familiar voltage divider circuit, presented in Fig.1.22.

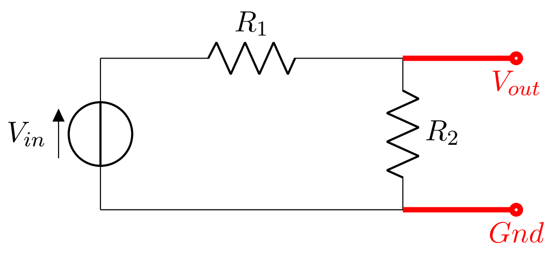


Figure 1.22: Voltage Divider.

Since, for a voltage divider the relation between V_{in} & V_{out} is

$$V_{out} = V_{in} \cdot \frac{R_2}{R_1 + R_2}$$

If any of the above three quantities are known, the fourth can be determined. Therefore, to convert the resistance of the *NTC, thermistor* into a corresponding voltage, and we connect it in series as R_2 with another known resistance, say $R_1 = 10,000\ \Omega$. We then measure the voltage in the middle, and as the resistance changes, the voltage changes too, in accordance with the above voltage-divider equation.

- The above procedure helps obtain the value of the resistance of the *NTC thermistor*, which offcourse is a function of the temperature. However, the final aim is to obtain / measure the **temperature**. The Steinhart-Hart equation, which to attempts to model the **thermistor resistance** as a function of **temperature**, and is given by

$$\begin{aligned} \frac{1}{T} &= A + Bln(R) + C\left[ln(R)\right]^3 \\ T &: \text{ temperature (in Kelvin)} \\ R &: \text{ resistance at T (in ohms)} \\ A,\ B\ \&\ C &: \text{ are the Steinhart Hart coefficients} \end{aligned}$$

This equation at times for the NTC thermistors also be characterised with the B or β parameter equation given by

$$\begin{aligned} \frac{1}{T} &= \frac{1}{T_0} + \frac{1}{B} ln\left[\frac{R}{R_0}\right] \\ T_0 &: \text{ room temperature (in Kelvin)} = 25^0\ C = 298.15K \\ R_0 &: \text{ resistance at room temperature } \sim 10K\Omega \\ B &: \text{ in this case } \sim \text{ 3950coefficient of the thermistor} \end{aligned}$$

1.9.2 Circuit

1.9.3 Sketch to record temperature

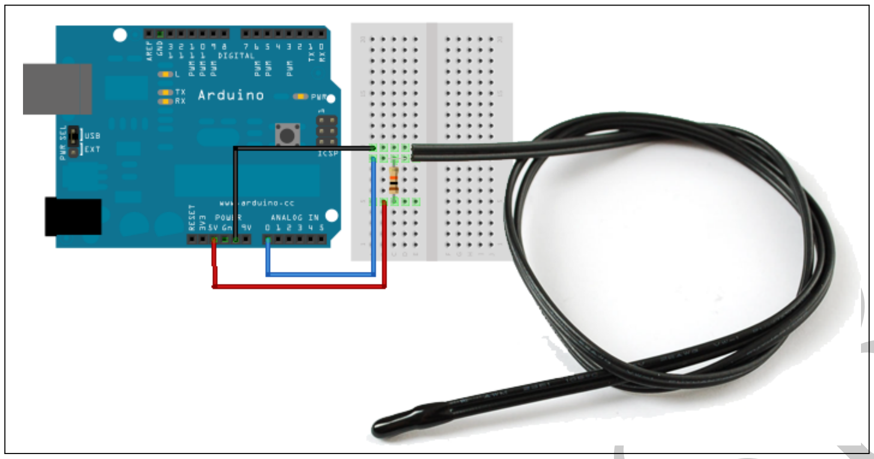


Figure 1.23: Circuit for connecting the thermistor to the Arduino.

```
byte NTCPin = A0;
#define SERIESRESISTOR 10000
#define NOMINAL_RESISTANCE 10000
#define NOMINAL_TEMPERATURE 25
#define BCOEFFICIENT 3950

void setup()
{
Serial.begin(9600);
}
void loop()
{
float ADCvalue;
float Resistance;
ADCvalue = analogRead(NTCPin);
Serial.print("Analoge ");
Serial.print(ADCvalue);
Serial.print(" = ");
//convert value to resistance
Resistance = (1023 / ADCvalue) - 1;
Resistance = SERIESRESISTOR / Resistance;
Serial.print(Resistance);
Serial.println(" Ohm");

float steinhart;
steinhart = Resistance / NOMINAL_RESISTANCE; // (R/Ro)
steinhart = log(steinhart); // ln(R/Ro)
steinhart /= BCOEFFICIENT; // 1/B * ln(R/Ro)
steinhart += 1.0 / (NOMINAL_TEMPERATURE + 273.15); // + (1/To)
steinhart = 1.0 / steinhart; // Invert
steinhart -= 273.15; // convert to C

Serial.print("Temperature ");
Serial.print(steinhart);
Serial.print(" \xC2\xB0");
Serial.println("C");
delay(10000);
}
```

Temperature Measurement with Thermistor

Since we know that in the above code two readings are spaced by 10 second time interval, we should be able to save this data to a file and then plot the temperature variation as a function of time.

1.10 PWM : Fade a LED

1.10.1 Introduction

Pulse Width Modulation, or PWM, is a technique for **mimicking** analog results using digital signals. Digital control is used to create a square wave, a signal switched continuously, between the ON, HIGH and OFF, LOW states. This repetitive *on-off* pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the portion of the time the signal spends ON versus the time that the signal spends OFF.

Offcourse, the response of the load which receives this signal is slower than the frequency of the pulse. For an LED, using PWM causes the light to be turned on and off at frequency than our eyes **cannot detect**. We simply perceive the light as brighter or dimmer depending on the widths of the pulses in the PWM output.

The various terms of relevance for PWM are

1. **On-Time**, the duration for which the signal is HIGH τ_o.
2. **Period**, the time taken for one complete oscillation.
3. **Duty-Cycle**, the percentage of time for which the signal remains ON, during the *period* of the signal, τ_o/τ_c.

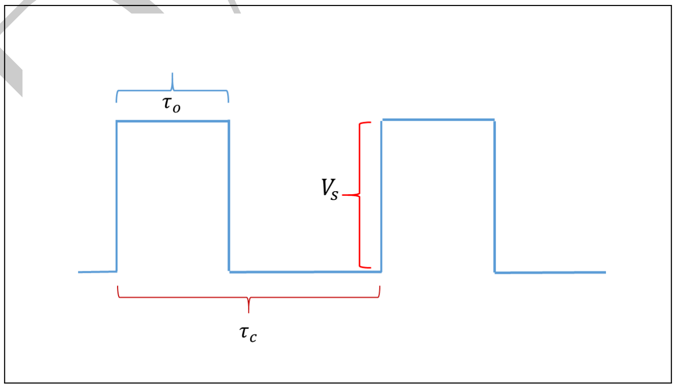


Figure 1.24: Nomenclature for PWM .

The output of a PWM channel is available on digital I/O pins 3, 5, 6, 9, 10 and 11. These pins can be identified as they have the symbol before the corresponding pin numbers. When this is supplied to a load whose response is slower than that of the frequency of the pulse, this appears to the load as an *effective* voltage of

$$V_{eff} = V_s \frac{\tau_0}{\tau_c}$$

Thus the output voltage is a function of the duty cycle, for a 100% duty cycle, the output, would be 5 volt, whereas a 50% duty cycle would result in an output of 2.5 volt.

Now, the analogWrite (PWM pin no, level) function can be used to achieve this, where the parameter level is an 8 bit integer, which can be set to obtain the desired duty cycle, which in turn governs the *effective voltage* as experienced by the slow load.

We know that since the parameter is a 8 bit integer, it's maximum value is 255. Thus we can use the scaling of 5 $V \longrightarrow 255$, hence the desired voltage could be set as $\frac{V_o}{5} \times 255$, where V_o is the desired effective analog voltage.

The pseudo-code to achieve this is

```
pinMode(PWM_out_pin, OUTPUT);
// Define one of the PWM pins as the OUTPUT pin in the initialization routine

analogWrite( PWM_out_pin, level);
// Set the duty cycle to the integer corresponding to the parameter level
```

The code below allows us to fade an LED, connected to pin number 9.

```
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(30);
}
```

Fade a LED using PWM

1.11 Charging of Capacitor

1.11.1 Introduction

A *capacitor* is made of a dielectric material within two parallel plates. When a battery is connected to a capacitor, positive charge collects on one plate and negative charge collects on the other plate until the potential difference between the two is equal to the voltage of the battery.

The *capacitance* is defined as the ratio between Q and V, where Q is the charge imbalance needed to produce a given voltage V across the capacitor.

$$C = \frac{Q}{V}$$

Some electrical circuits contain both resistors and capacitors. These are called *capacitor* RC circuit (Fig.1.25). If the initial voltage applied to an RC circuit is V_0, and the capacitance of the capacitor is C, and the resistance of the resistor is R, then the amount of time, t, it takes for the capacitor to reach the charge, Q, is given by:

$$Q = C \cdot V_0 \left(1 - e^{\left[\frac{t}{RC}\right]}\right)$$

$$V(t) = V_0 \left(1 - e^{\left[\frac{t}{RC}\right]}\right)$$

$V(t)$ is the voltage across the capacitor at time t

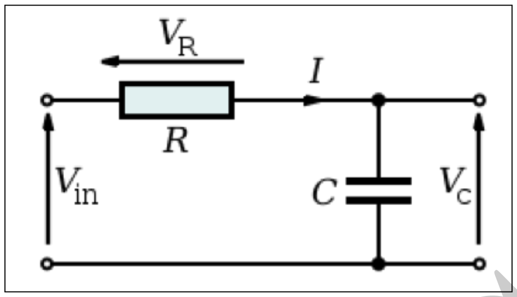


Figure 1.25: Series RC circuit.

1.11.2 Circuit

The circuit diagram, for investigating the charging of a capacitor, using magenta is presented in Fig.1.26.

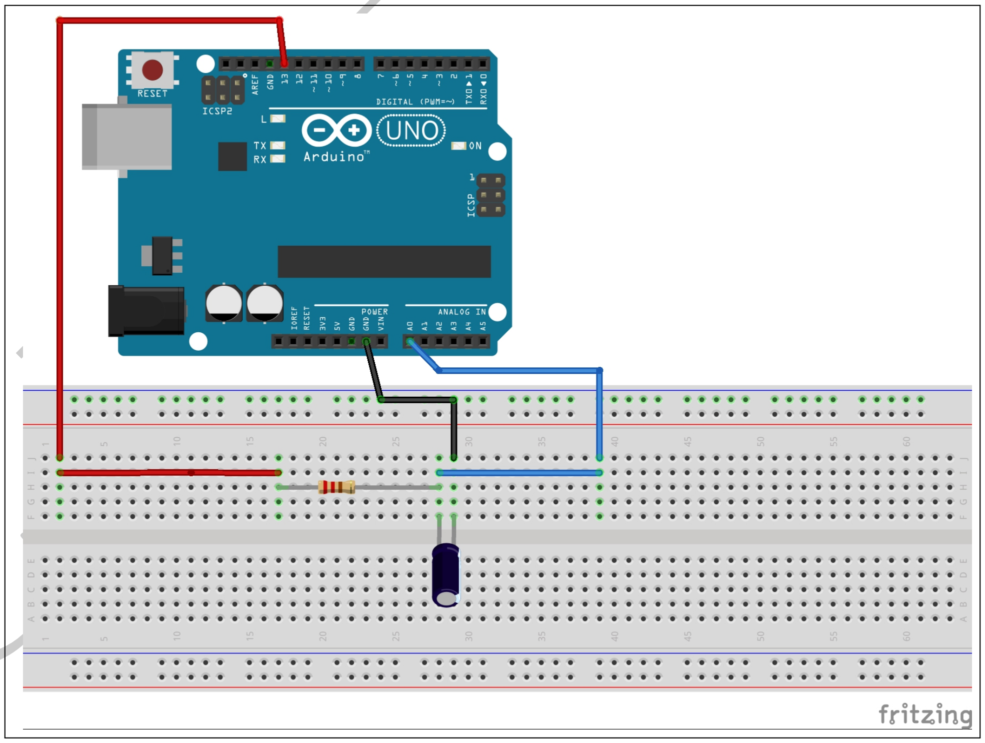


Figure 1.26: Circuit diagram.

We will use one of the *digital outputs* of Arduino to start the charging of the capacitor, say **Pin**

13.

Before restarting a new run, discharge the capacitor, by removing it from the circuit and **shorting the two terminals manually**

1.11.3 Sketch for Acquiring Data

```
 1 int SensorValue = 0 ;
 2 int led = 13 ;
 3 int data;
 4 float Voltage1 = 0.0;
 5 float Voltage2 = 0.0;
 6 long unsigned now = 0.0 ;
 7 long unsigned then = 0.0 ;
 8 long unsigned elapsed = 0.0 ;
 9
10 void setup() {
11   // put your setup code here, to run once:
12   pinMode(LED_BUILTIN, OUTPUT);
13   Serial.begin(9600);
14   digitalWrite(led, LOW);
15   delay(10000);
16   digitalWrite(led, HIGH);
17   then = millis();
18 }
19
20 void loop() {
21   // put your main code here, to run repeatedly:
22 SensorValue = analogRead(A0);
23 now = millis();
24 Voltage1 = SensorValue * ( 5.0/1023.0);
25 elapsed = now - then ;
26 Serial.print(elapsed) ;
27 Serial.print("\t");
28 // Serial.print(",");
29 Serial.println(Voltage1);
30 delay(1000);
31 }
```

Study the Charging of the Capacitor

1.11.4 Procedure

Once we start the charging, by setting the above pin to high state, (Lines 15), prior to doing so, we wait sufficiently long enough (Line 14) for us to fire up the serial terminal, on which we output the result.

We need to **disable the auto scroll** as well as **set the baud rate to 9600** the same value we had set in our code.

Since, we have a 10 bit ADC, it has $2^{10} = 1024$ channels, which are used to store a maximum of 5 Volt. Using this information we can map the ADC voltage into the corresponding voltage, which ten is displayed on the screen.

On the monitor we observe the time and the voltage across the capacitor. These values are seperated either by a tab (Line 26), or by a comma, if we were to activate the next line, and comment the previous line. Accordingly in the analysis routine, we will have to set the delimiter.

These values are to be copied to an ASCII file, which is achieved, by

1. Use the command CNTR+A, which *selects all*
2. Copy the selected contents using the familiar CNTR+C command.
3. Open the Python GUI
4. Select the File option and the New File sub-menu
5. In the poped up empty window CNTR+V to copy the contents
6. Save the file as a **.txt** file by choosing the option as *txt* in the *Save As* sub-menu. You may store the file in any appropriate sub directory.
7. Scroll down to the end of the file and ensure that the last line is complete. Delete any half written file.

The program is terminated by **closing the serial monitor**

The IDE does not allow us to store the values into a file, hence the round about way to copy-paste the values from the serial monitor into an ASCII file, which can be later analyzed using Python, or any other toolkit or software.

1.11.4.1 Analysis Of The Acquired Data

We can read back the data stored in the file, and using the value of the time calculate

1. Voltage across the resistor at that instance (Line-21).
2. The theoretical value of the voltage across the capacitor (Line 25).

The following code is used for the analysis of the stored data, and the results are presented in Fig.1.27.

```python
import numpy as np
import matplotlib.pyplot as plt
import os

r = 100000
c = 220E-06
Vin=5.0
f1 = open('ssg1.txt','r')
header1=f1.readline()       # skip the header
data1 = np.genfromtxt(f1,delimiter='\t')
#data1 = np.genfromtxt(f1,delimiter=',')
y_vr=[]         # array to store the voltage across the resistor
y_vc_theo=[ ]   # arry to store the calculated voltage across capacitor
temp1 = 0.0     # temporary variable
temp2 = 0.0     # temporary variable
tau = r*c       # tau of the circuit

x_time=data1[:,0]
y_vc=data1[:,1]

for i in range(len(x_time)):
        temp1 = 5.0 - y_vc[i]
        y_vr.append(temp1)
        temp2 = Vin-Vin*(np.exp((-x_time[i])/(r*c)))
        y_vc_theo.append(temp2)
        temp1=0.0
        temp2=0.0

f1.close()
X=[0,tau,10*tau,100*tau]
Y=[0.63*Vin,0.63*Vin,0.63*Vin,0.63*Vin] # for observing tau
plt.plot(X,Y,"r--")
plt.plot(x_time,y_vc,"go",label="Expt")
plt.plot(x_time,y_vc_theo,"b*",label="Theoretical")
plt.legend(loc="best")
plt.xlabel('Time (second)')
plt.ylabel ('Voltage across capacitor')
plt.xlim(0,max(x_time))
plt.grid()
plt.show()
```

1.11.5 Results

From the figure, we observe that the experimental data is in agreement with the theoretical value, and we can also confirm the value of the time constant from this graph.

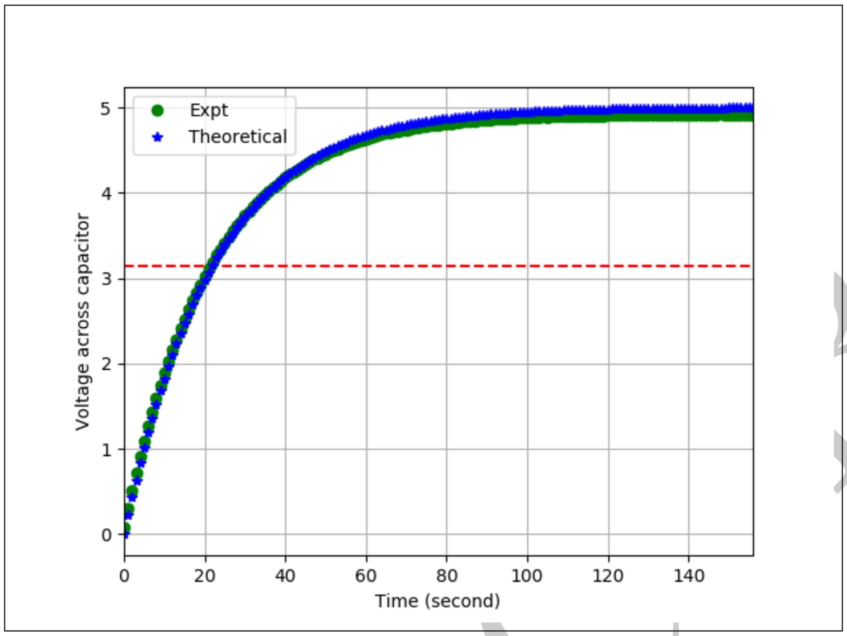


Figure 1.27: Voltage across the capacitor during charging.

1.12 Astable Multivibrator using 555 Timer

1.12.1 Introduction

Astable multivibrator has *no stable state*. It keeps on oscillating between the two states. In a 555 timer, we bring about a transition between the states, using the voltage developed across a capacitor during charging and discharging. Hence, there is a correlation between the output voltage and the voltage across the capacitor, which can be viewed using a two channel oscilloscope.

1.12.2 555 Timer

The linear ICs 555 timer was first introduced in early 1970 is one of the most versatile workhorse in both hobby as well as professional electronics. This IC is a monolithic timing circuit that can produce accurate and highly stable time delays or oscillation. Besides it is easily available and is reliable as well as economical. It has a variety of applications including, the most important being it?s use as a multivibrator, and it?s ability to provide the inputs for a digital *TTL* (**T**ransistor-**T**ransistor **L**ogic), which relies on transistors to achieve switching and maintain logic levels ($0\ V \leq V_{low} \leq 2V$ & $2.7\ V \leq V_{high} \leq 5\ V$)

The IC can operate under a wide range of input voltage $+5\ V \leq V \leq +18\ V$, besides also has a wide operating temperature range from $0^0C\ to\ 70^oC$. Further, it can **source** (ability to deliver) or **sink** (the ability to receive current) about 200 mA of current.

The IC is routinely available as an 8pin mini DIP (dual-in-package) and comprises of 23 transistors, 2 diodes and 16resistors. The pin configuration of the IC is pictorially presented in Fig.1.28.

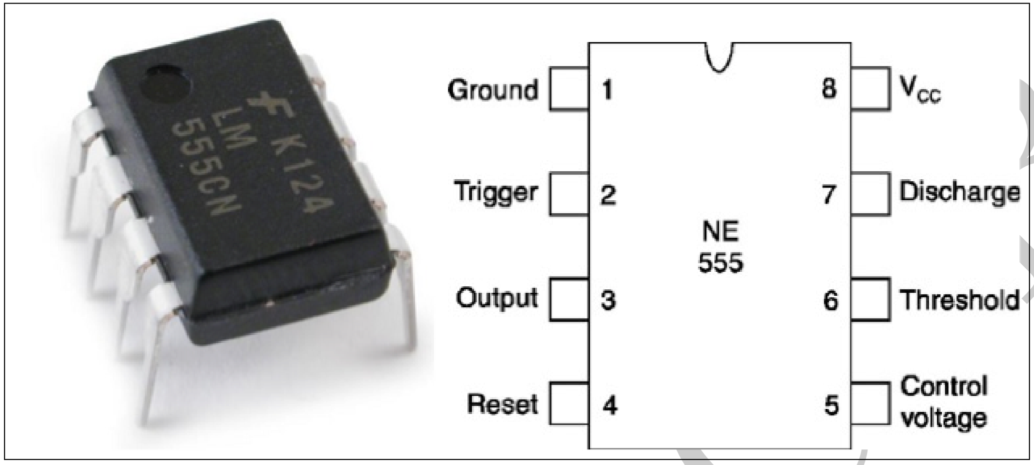


Figure 1.28: The pin configuration of the 555 timer IC.

The primary building blocks (depicted in Fig.1.29) of the IC are

1. Voltage Divider
2. Comparators
3. RS Flip Flop

Comparator as the name suggest is a device that essentially compares the voltages at the input terminals produce an output voltage dependent upon the voltage difference at their inputs. On the other hand a *fip-flop*, has **two** stable states and can be used to store state information, and accordingly produces either a **HIGH** or **LOW** level output at Q based on the states of its inputs. It also has a complementary $\overline{Q}$ output.

The three 5kΩ resistors connected in series internally (this is how the chip got it?s name) which produce a *voltage divider network* (a simple circuit which turns a large voltage into a smaller one) between the supply voltage at **Pin-8** and ground at **Pin11**. The voltage across this series resistive network holds the negative inverting input of *comparator two* at $\frac{2}{3}\ V_{cc}$ and the positive non-inverting input to *comparator one* at $\frac{1}{3}\ V_{cc}$.

The two comparators produce an output voltage dependent upon the voltage difference at their inputs. The outputs from both comparators are connected to the two inputs of the flip-flop. The complimentary output $\overline{Q}$ of the flip-flop is used to control a high current output switching stage to

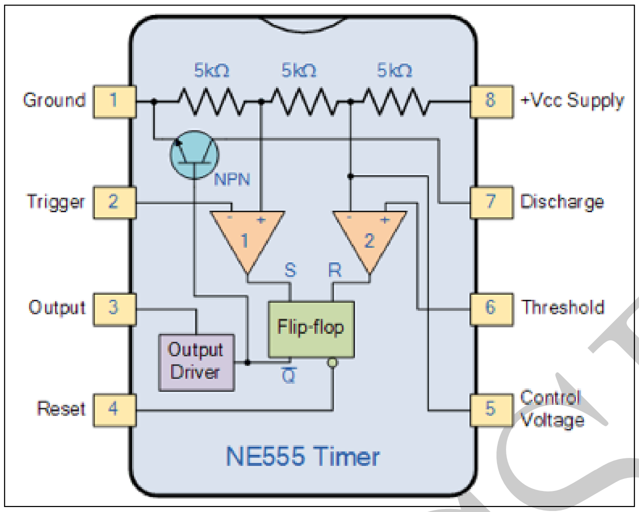


Figure 1.29: A simplified *block diagram* illustrating the internal functional blocks of the 555 timer.

drive the connected load producing either a **HIGH** or **LOW** voltage level at the output pin.

Suppose we use $V_{cc} = 9V$, then a voltage of $6\ V = \frac{2}{3} \times 9$, at the positive input of comparator-II, **Pin-6** would cause it?s output to go HIGH, which in turn being applied to the RESET terminal which causes the FF to get reset (a transition from HIGH to LOW). Similarly when the voltage drops below $3\ V = {}^1\!/_3 \times 9$ at the negative terminal of comparator-I, whose output is connected to the SET input of the FF, causes the FF to make a transition from LOW to HIGH.

Pin-1 serves as the ground pin and all voltages are measured with respect to this pin.

The supply voltage of $+5\ V$ to $+18\ V$ is applied to **Pin-8**, with respect to ground, **Pin-1**.

Output of the timer is available at **Pin-3** and is capable of driving TTL circuits and as mentioned earlier it can source or sink about 100 mA of current. There are two ways in which a load can be connected to the output terminal. One way is to connect it between output **Pin-3** and ground **Pin-1** referred to as normally off load or between **Pin-3** and supply **Pin-8**, termed as normally on load.

Whenever the timer IC is to be reset or disabled, a negative pulse is applied to **Pin-4**, and hence this pin is termed as RESET terminal. The output is reset irrespective of the input condition. When this pin is not to be used for reset purpose, it should be connected to $+V_{cc}$ to avoid any possibility of false triggering.

We refer to **Pin-7** as Discharge Terminal & it is connected internally to the collector of transistor and *routinely a capacitor is connected between this terminal and ground.* It is called discharge terminal because when transistor saturates, capacitor discharges through the transistor. When the transistor is cut-off, the capacitor charges at a rate determined by the external resistor and capacitor.

Pin-5 is termed as Control Voltage Terminal. The threshold and trigger levels are controlled using this pin. The external voltage applied to this pin can also be used to modulate the output waveform and the width. When this pin is not used, it should be bypassed to ground through a $0.01\mu Farad$ capacitor to avoid any noise problems.

A *multi-vibrator* is a circuit which usually has **two states** and it oscillates or switches under external condition between these states. An Astable multi-vibrator has *no stable states* and keeps switching from one state to another. The Mono-stable multi-vibrator has *one stable state*, and under an external input it switches to the unstable state and then returns to it?s stable state. The Bistable multi-vibrator as it?s name suggests has both the states as stable, and the switching between them is achieved using two terminals SET and RESET.

The Astable multi-vibrator is also referred to as Free Running Multivibrator, as it switches between the two states on it?s own without need for any external circuitry. The circuit diagram for configuring the 555 timer as an astable multi-vibrator is shown in the Fig.

The Trigger **Pin-2** and Threshold **Pin-6** are connected to the capacitor, whose value governs the output of the Timer. The capacitor **C** charges through both R_1 & R_2, while discharges only through R_2. Since the Control Voltage pin is not used, the output of 555 will be HIGH when the capacitor voltage goes below $1/3V_{cc}$, and the output will go LOW when the capacitor voltage goes above $2/3V_{cc}$.

When the circuit is switched **ON**, and as the capacitor C would be initially uncharged voltage will be less than $1/3V_{cc}$. So the output of the lower comparator will be HIGH and that of the upper comparator will be LOW. This would *SET* the output of the Flip-flop, which forces the discharging transistor to be OFF and allows the capacitor C to start charging from V_{cc} through resistor R_1 & R_2. The charging continues till the capacitor voltage is $< 2/3V_{cc}$,

$$T_{high} = 0.693 \times \left[(R_1 + R_2) \times C\right]$$

When the capacitor voltage is $2/3V_{cc}$ the FF is *RESET*, and the transistor is turned ON. This allows the capacitor to start discharging through R_2 and the voltage across it starts decreasing

$$T_{low} = 0.693 \times \left[R_2 \times C\right]$$

This continues till the capacitor voltage is less than $1/3V_{cc}$. This *SET's* the FF and the capacitor starts charging once again. This process continues and a rectangular wave (Fig.1.30) is obtained at the output with

$$\begin{aligned}\text{Frequency} &= \frac{1}{\left[T_{high}\ +\ T_{low}\right]} \\ &= \frac{1.44}{\left[R_1 + 2 \times R_2\right]\ \times\ C}\end{aligned}$$

The *Duty Cycle*, is given by

$$\text{Duty Cycle}\ =\ \left[\frac{T_{high}}{(T_{high}\ +\ T_{low})}\right]$$

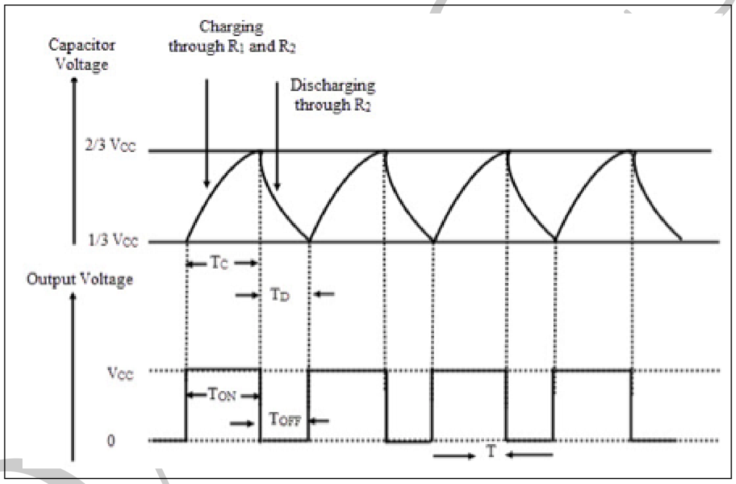


Figure 1.30: Waveforms for 555 timer as Astable Multivibrator.

1.12.3 Circuit

The circuit diagram for an Astable Multivibrator is presented in the Fig.1.31.

If we do not wish to **assemble** the circuit from the components, an open source Arduino compatible NE555 Pulse Frequency Duty Cycle Adjustable Module Square Wave Signal Generator is readily available, and is depicted in Fig. 1.32.

If we were to use the above board, then we can use the 5 Volt and the ground rails from the Arduino board.

The *output* pin from the 555 timer be connected to any one of the analog input terminals of the μ-controller, say A0.

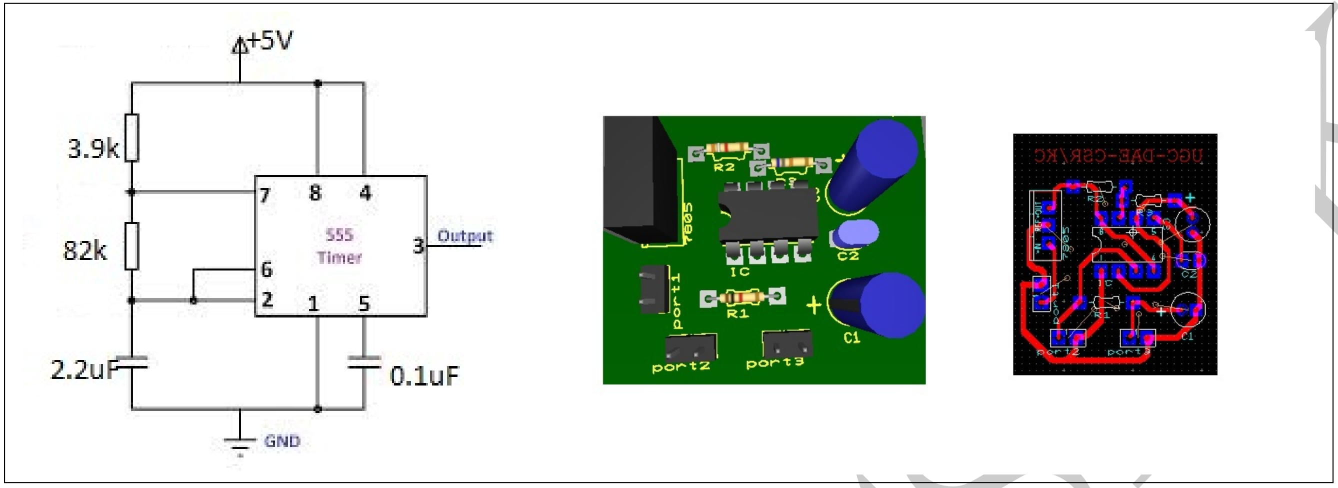


Figure 1.31: Circuit Diagram for 555 timer as Astable Multivibrator.

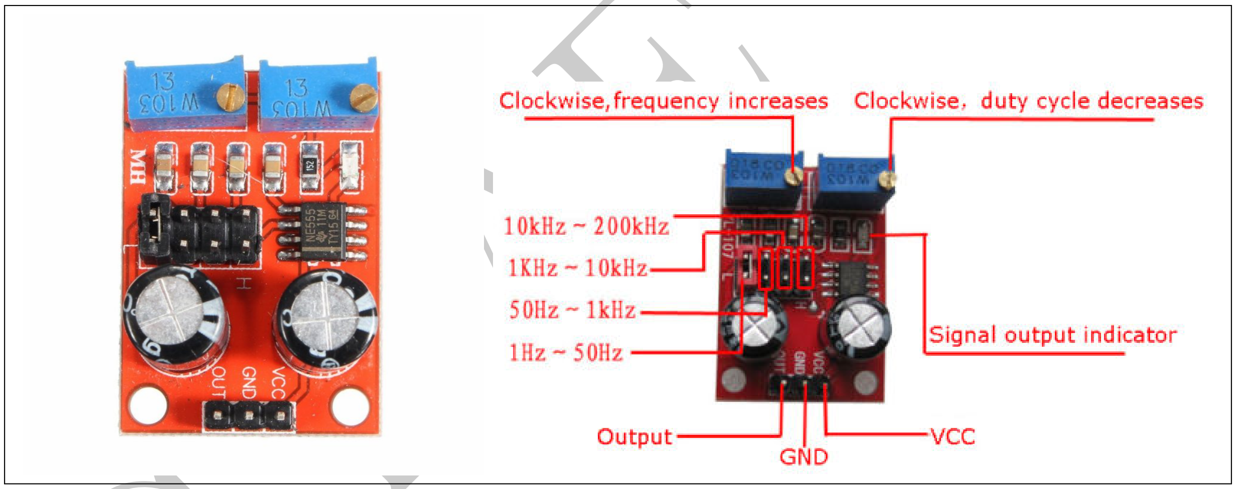


Figure 1.32: Open source 555 timer board.

1.12.4 Sketch for Acquiring Data

```
1  int SensorValue = 0;
2  float Voltage1 = 0;
3  unsigned long startMillis;
4  unsigned long currentMillis;
5  unsigned long now;
6
7  void setup() {
8  // initialize serial communication at 9600 bits per second:
9    Serial.begin(9600);
10 // initialize start time :
11   startMillis = millis();
12 }
13
14 // the loop routine runs over and over again forever:
15 void loop() {
16   currentMillis = millis();
17   now = currentMillis - startMillis;
18   SensorValue = analogRead(A0);
19   Voltage1 = SensorValue * (5.0 / 1023.0);
20   Serial.print(now);
21   Serial.print("\t");
22   Serial.println(Voltage1);
23   delay(10);
24 }
```

Study Of 555 timer

Now, in the above code, if we were to comment out Lines 20-21 and view the results on the **inbuilt serial monitor**, we would observe the Fig. 1.33.

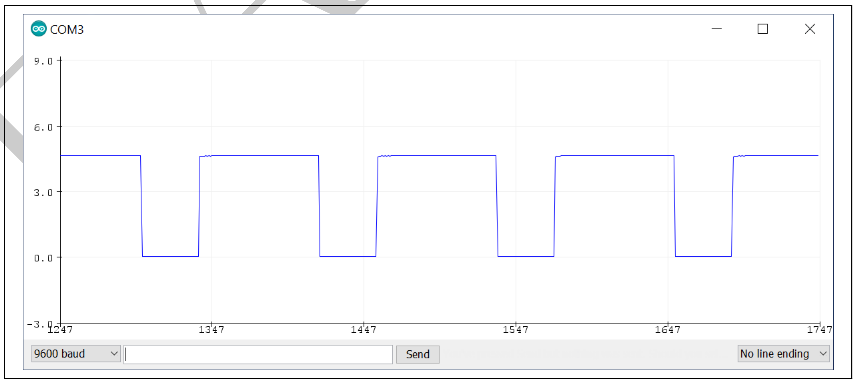


Figure 1.33: Square wave output from 555 timer.

Now, if we were to run the above *sketch* with the two lines uncommented, then on theserial monitor, we would observe a **tab** seperated two column data.

We can save the data CTRL+A ; CTL+C and then paste it in a plain ASCII file, using the Python IDLE.

1.12.5 Analysis of the Data

The following Python code (*astable_mv.py*), allows us to read the data,,and plot it, which is presented in Fig.1.34.

```
import numpy as np
import matplotlib.pyplot as plt
import os

f1 = open('555_timer.txt','r')
header1=f1.readline()        # skip the header
data1 = np.genfromtxt(f1,delimiter='\t')

x_time=data1[:,0]/1000.0
y_vc=data1[:,1]
f1.close()

plt.plot(x_time,y_vc,"g--")
plt.xlabel('Time (second)')
plt.ylabel ('Output Voltage')
plt.title ('555 Timer as Astable Multivibrator')
plt.xlim(0,max(x_time/2.0))
plt.grid()
plt.show()
```

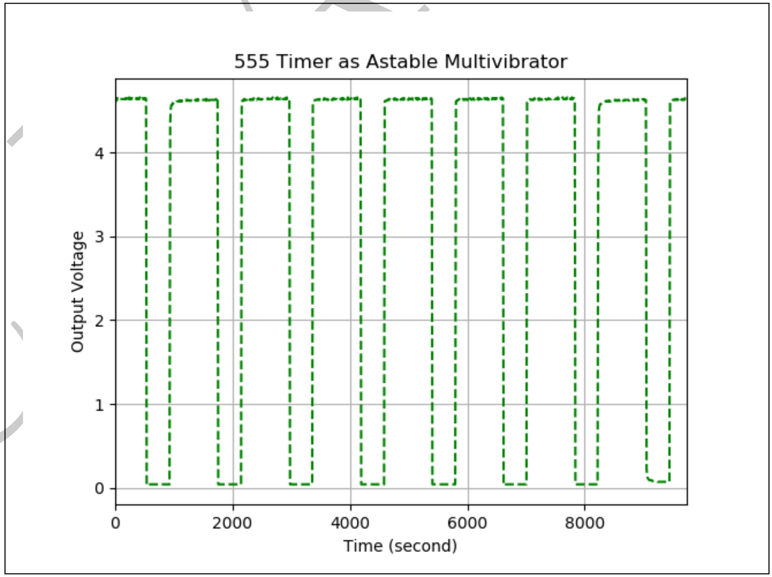


Figure 1.34: Square wave from the 555 timer board.

1.12.6 Results

We can calculate the **Time period** from the x-axis reading, for one complete cycle, from the graph.

When we view the plot, then as we move the cursor over the plot, we are able to observe the x & y co-ordinates. The x would give us the time, we have scaled the **x_data**, from milli-seconds to seconds.

1.13 Transistor as a switch

1.13.1 Introduction

A transistor, is a **three-layered**, **two junction** device, semiconductor device, Fig.1.35.

These devices are also referred to as Bipolar Junction Transisitor (BJT), due two the presence of **two** types of charge carriers *viz.*, holes and electrons.

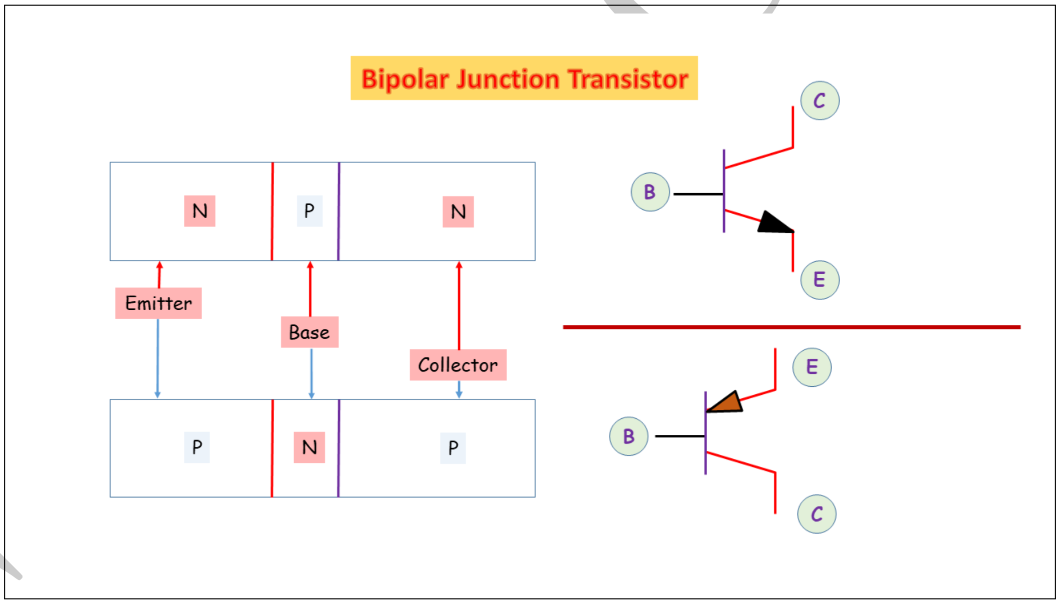


Figure 1.35: Bipolar Junction Transistor.

The Emitter, is heavily doped and is the source of the *majority* charge carriers. The Base, is lightly doped and has a relatively smaller cross sectional area. The Collector has a relatively *large cross sectional area*, so as to efficiently receive the charge carriers which flow into this region.

The **Emitter-Base** junction is forward biased , while the **Collector-Base** junction is reversed biased, under normal operating conditions.

We know that a transistor can be used a switch, to control the flow of current to a particular device.

It is based on the principle of operation of a transistor, that a transistor is operational **only when we have provided it a base drive (current)**, besides other factors, regarding the biasing of the emitter-base and collector-base junctions.

When the voltage at the base is greater than 0.6V, the transistor starts saturating and looks like a short circuit between the collector and the emitter. When the voltage at the base is less than 0.6V the transistor is in cutoff mode. This is pictorially illustrated in Fig.1.36.

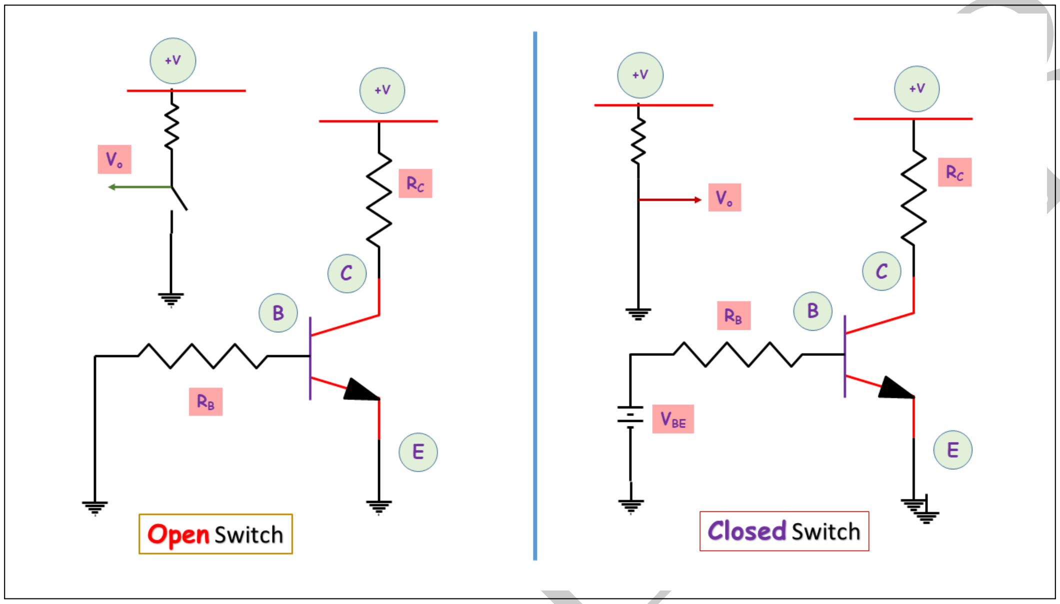


Figure 1.36: BJT as **switch**.

Hence, the *base drive* can be provided from a *micro-controller*, and accordingly the transistor be made to operate analogous to a mechanical switch.

In Fig.1.37, if we were to provide a base drive (voltage) greater than 0.6 V, then the transistor would act as an closed switch and the path for the flow of current is established, so that the LED will **GLOW**.

In absence of any base drive, we have the transistor as an closed switch, which impends the flow of current and accordingly, the LED is **OFF**.

1.13.2 Circuit

Now, we can connect the Digital Pin 13 to the base of the transistor, and on execution of the Blink script, both the *onboard* and *LED* would blink simultaneously.

This is due to the fact that when, we make the *Digital Pin 13* HIGH, we do provide a base drive to the transistor, which drives the transistor, into saturation, and the *base* and *emitter* get short, enabling flow of current, and as a consequence, the LED is ON.

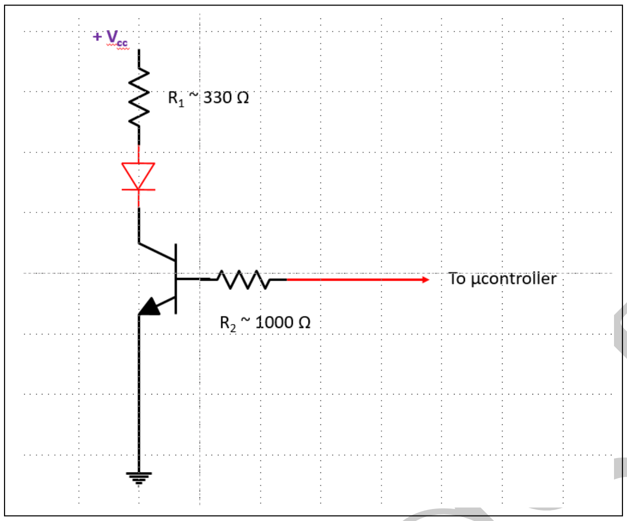


Figure 1.37: Transistor as a switch.

1.13.3 Determination of Planck's Constant from this circuit

We know that when the transistor is operating as a **closed switch**, we have the LED getting forward biased, and allowing the passage of current through it.

Under this condition, if we were to measure the voltage after the LED, when it is ON, we would be able to compute it's $V_{knee} = V_g$, from the quantity $V_{cc} - V_{led}$.

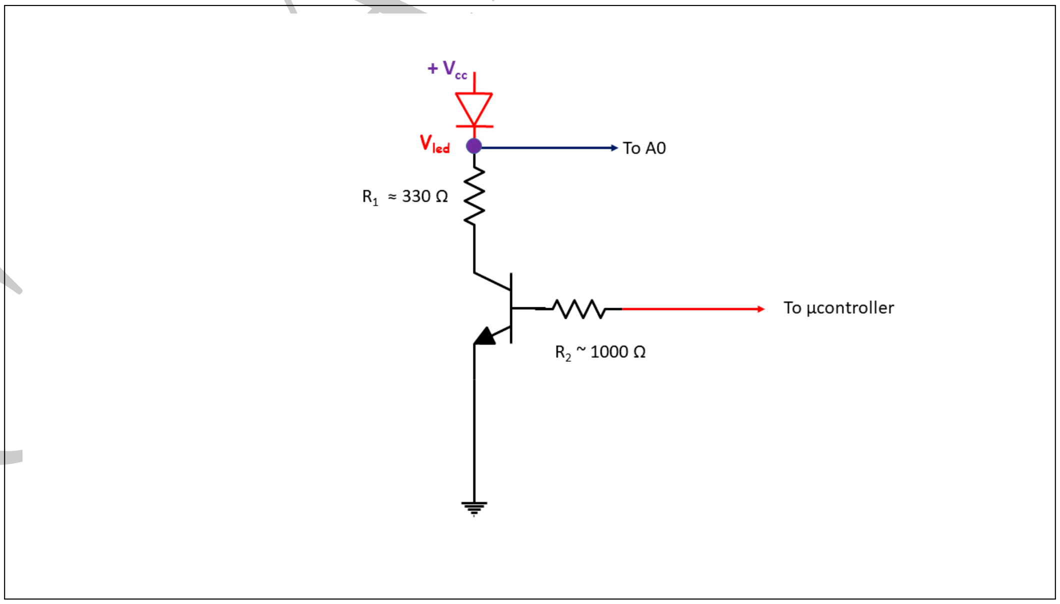


Figure 1.38: Determination of h.

```
int ledPin = 13;    // LED connected to digital pin 9
int SensorValue =0;
float Voltage =0.0;

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  // fade in from min to max in increments of 5 poi
    digitalWrite(ledPin, HIGH);
    SensorValue=analogRead(A0);
    Voltage = SensorValue *(5.0/1023.0);
    Serial.println(Voltage);
   delay(5000);
   digitalWrite(ledPin, LOW);
    SensorValue=analogRead(A0);
    Voltage = SensorValue *(5.0/1023.0);
    Serial.println(Voltage);
   delay(5000);

}
```

Determination of Planck's Constant

Now, we know that the v_g, is related to the Planck's Constant h as

$$h = \frac{e \times V_g \times \lambda_g}{c}$$

We have used coloured LEDs and have extracted the V_g, from the **ON** condition of the LED. The data on the corresponding λ has been obtained from the literature, and remains the main source of uncertainty in the extraction of h. The results are summarized in the table below

Color	$\lambda_g\|_{nm}$	$V_g\|_{volt}$	$h\|_{joule-second}$
Red	665	1.84	6.36×10^{-34}
Yellow	590	1.95	6.52×10^{-34}
Green	560	1.94	6.13×10^{-34}
Orange	635	1.88	5.79×10^{-34}

1.14 Time Period of Pendulum

1.14.1 Introduction

We know that the time period (T) of a simple pendulum is the time taken by the pendulum to perform one complete oscillation.

It is related to the length l of the pendulum and acceleration due to gravity g at that place, such that

$$T = 2\pi\sqrt{\frac{l}{g}}$$

Hence, the problem boils down to performing an accurate time measurement, devoid of human errors such as response time etc.

One possible solution is to use an infrared obstacle sensor, which produces a change in a signal, when an object interrupts or blocks the beam from an IR **receiver** on to the **transmitter**. Commercially such modules are referred to as IR Proximity Sensor.

1.14.2 Proximity Sensor

The IR Proximity sensor is based on the principle of IR reflectance. IR light is constantly emitted by an IR LED, which is received by an IR Receiver LED / Photo-diode, placed in it's direct line of sight. This signal is processed by an Op-Amp and the Op-Amp produces say for example a **HIGH** signal. Thus if the sensor detects an object it will produce a high signal.

The IR transmitter is a light emitting diode which emits Infra-Red radiations whose wavelength is typically around 700 nm – 100 nm. They are very similar to normal LED in appearance.

Infrared receivers detect the radiation from an IR transmitter. IR receivers come in the form of photo-diodes and photo-transistors. Infrared Photo-diodes are different from normal photo diodes as they detect only infrared radiation.

We usually place both the transmitter and receiver in direct line of sight for this experiment, and detect the interruption of the IR beam, due to the passage of the bob of the pendulum.

Hence, when the bob of the pendulum obstructs the transmitter and receiver, we should get a blip in the voltage level (a change in the level, say from $HIGH \longrightarrow LOW$ or vice versa). A representative circuit diagram is presented in the *left panel* of Fig.1.39.

These are available commercially as well, *right panel* of Fig.1.39. In case you use the commercial module, then the IR transmitter and receiver would have to be de-soldered from the pcb, and physically mounted either on a PCB or a breadboard, facing each other.

A simplified version of this presented in Fig.1.40 , could also be used. Care has to be taken to ensure that the bob of the pendulum passes through the line of sight of the transmitter and receiver, without colliding with them (Fig.1.41).

In one oscillation, the bob intercepts the receiver and transmitter three times. Hence, the time difference between three "blips" or "LOW signal" would give us the time period (Fig.1.42).

The code to capture the data is exactly identical to the one used to record the voltage across the capacitor during charging.

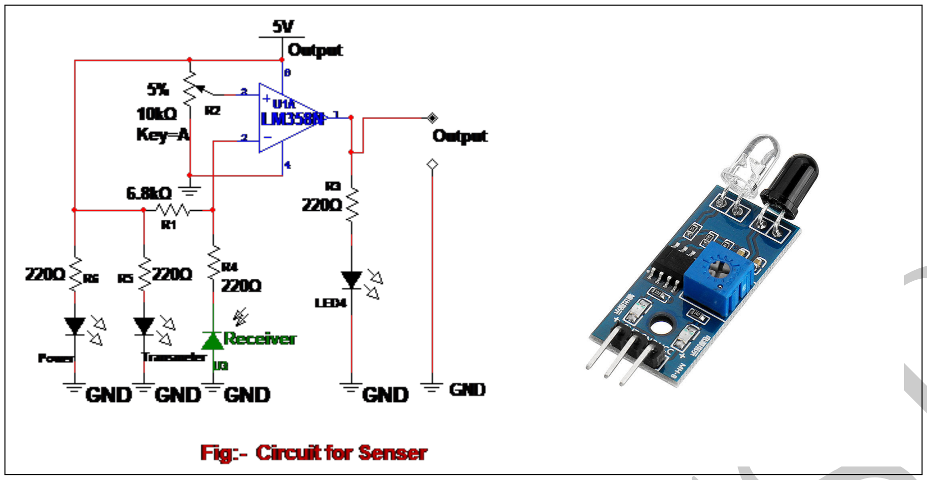


Figure 1.39: Proximity Sensor.

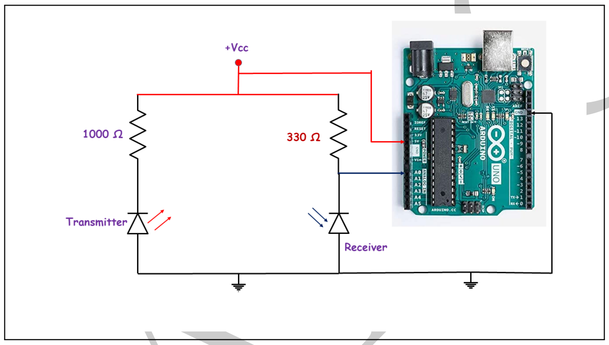


Figure 1.40: **IR Transmitter & Receiver** coupled to Arduino.

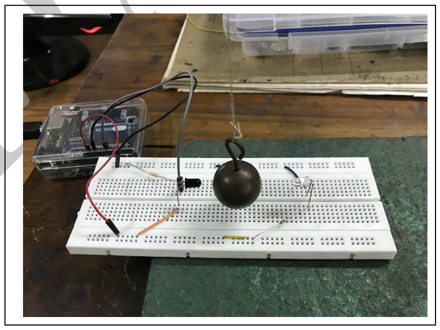


Figure 1.41: Experimental Setup.

1.14.3 Analysis

The stored data is then read back using the code (*pendulum.py*). Since the time is in milli-seconds, we convert it into seconds, before analysis.

```
import numpy as np
import matplotlib.pyplot as plt

data = np.genfromtxt("pendulum_data.txt")

time1 = data[:,0]/1000.0
voltage = data[:,1]

plt.plot(time1,voltage)
plt.xlabel('Time (s)')
plt.ylabel ('Voltage (Arb. unit)')
plt.title ('Measurement of time period of a pendulum')
plt.grid()
plt.show()
```

Measurement of Time Period of a Pendulum

The data is then plotted (Fig.1.42), and since it is a *time stamped data*, we can obtain the time interval between three *blips* or change in level, which would give us the **Time period** for the given pendulum.

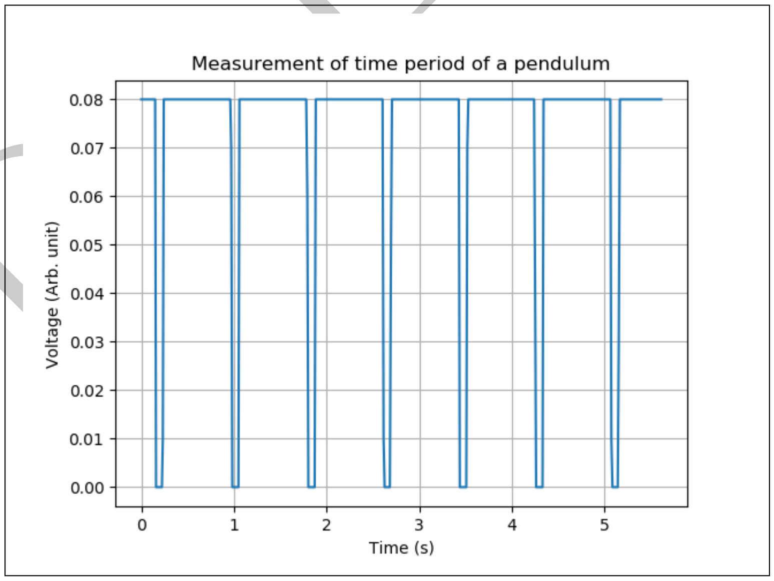


Figure 1.42: Square wave from the 555 timer board.

Now, in the test experiment, we used a pendulum, whose length was $l = 66\ cm$, hence the expected time period is

$$\begin{aligned} T &= \frac{1}{2\pi}\sqrt{\frac{l}{g}} \\ &= \frac{1}{2\pi}\sqrt{\frac{0.66}{9.8}} \\ &\approx 1.6\ s \end{aligned}$$

The time difference between *3 blips* in the recorded signal also yields a value of $T \sim 1.6\ s$.

1.15 Arduino and LDR

1.15.1 Voltage Divider

A voltage divider is a relatively simple circuit which is used to **scale** down a large voltage into a smaller one. This is of particular relevance to Arduino, which accepts, only a maximum of 5 V of *analog* voltage, whereas in the **real world**, we have to deal with much larger voltages at times. The configuration comprises of two series resistors R_1 & R_2 across which we apply the **input** voltage, V_i. The output voltage, V_o is scaled according to the value of the two resistors, such that

$$\begin{aligned} V_o &= V_i \cdot \frac{R_2}{R_1 + R_2} \\ if \quad R_1 = R_2 = R \quad \text{then} & \\ V_o &= V_i \cdot \frac{R}{R+R} \\ &= \frac{V_i}{2} \\ if \quad R_2 >> R_1 \quad \text{then} & \\ V_o &\sim V_i \cdot \frac{R_2}{R_2} \\ &\sim V_i \quad \text{output voltage is very close to the input voltage} \\ if \quad R_2 << R_1 \quad \text{then} & \\ V_o &\sim V_i \cdot \frac{0}{R_1} \\ &\sim 0 \quad \text{most of the input voltage would be across } R_1 \end{aligned}$$

Hence, if R_1 & V_i are known, we could deduce the value of the R_2, which comes in handy, if we were to use a **thermistor**, to measure the temperature, as the resistance of the thermistor is dependent on the temperature. Similarly, a Light Dependent Resistor could be used to qualitatively determine the light in our surroundings.

1.15.2 Light Dependent Resistor

A photo-resistor or **L**ight **D**ependent **R**esistor is a component that is sensitive to light. When light falls upon it then the resistance changes. Values of the resistance of the LDR may change over many orders of magnitude the value of the resistance falling as the level of light increases.

LDRs are made from semiconductor materials to enable them to have their light sensitive properties. However, they are **passive** components, in the sense that they do not comprise of the conventional PN junction.

When, the light intensity, decreases, the resistance of the LDR increases. The LDR being a semiconductor, in absence of light, it has a high resistance, *viz.* absence of sufficient *mobile* or *free* electrons. The vast majority of the electrons are bound / locked into the crystal lattice and are unavailable for a free movement within the crystal.

As light is incident on the semiconductor, the light photons are absorbed by the semiconductor lattice and some of their energy is transferred to the electrons. This gives some of them sufficient energy to break free from the crystal lattice, making them available for conduction (movement within the entire crystal). This results in a lowering of the resistance of the LDR.

This also explains the direct proportionality between the intensity of the light and the lowering of the resistance.

1.15.3 Arduino & LDR

The LDR is used as one of the resistors *viz.* R_2, in a voltage divider circuit, and the **voltage** between the two resistors, is measured using a Arduino as shown in the Fig.1.43.

We expect that in darkness the resistance of the LDR should be quite high, whereas as the intensity of light increases, we expect the resistance of the LDR to reduce substantially.

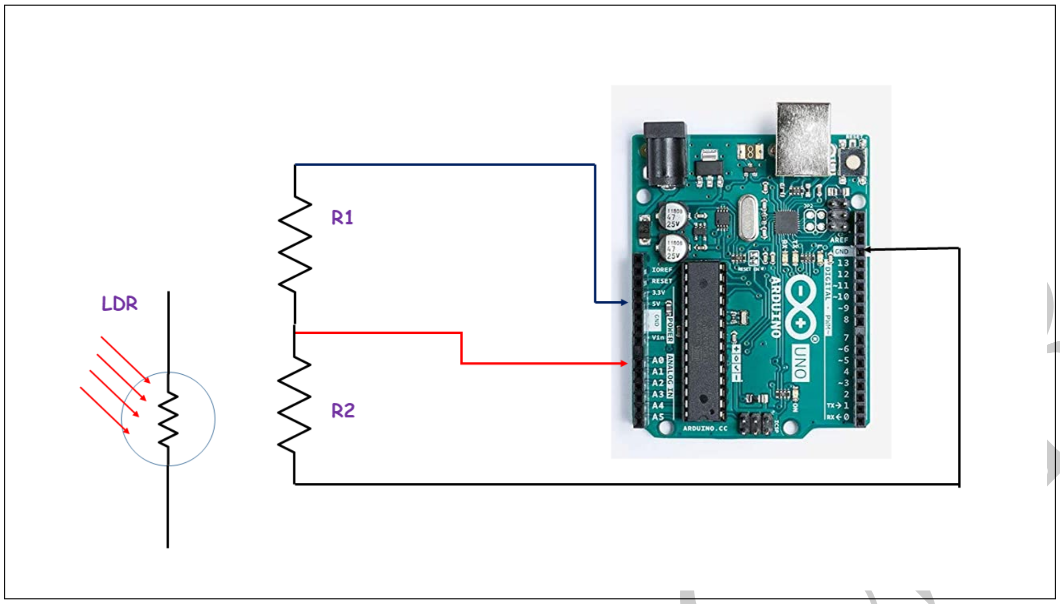


Figure 1.43: LDR connected in a voltage divider (replacing R_2) and coupled to an Arduino.

```
// the setup routine runs once when you press reset:
void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
}

// the loop routine runs over and over again forever:
void loop() {
// read the input on analog pin 0:
int sensorValue = analogRead(A0);
// Convert the analog reading (which goes from 0 - 1023) to a voltage (0 - 5V):
float voltage = sensorValue * (5.0 / 1023.0);
// Compute the ratio of the two voltages
// float vratio = voltage/5.0;
// Compute the value of the resistance of the LDR
float Rx = -1000.0/(vratio-1);
// print out the value you read:
Serial.print(voltage);
Serial.print("\t");
Serial.println(Rx);
delay(5000);
}
```

Study of LDR

In the setup we used, in darkness the value of the resistence of the LDR was $\sim 340\ K\Omega$, and when we used the light from our mobile just above the LDR, the resistance dropped to $\sim\ 1\ K\Omega$

1.16 Precautions

Before commencing any experiment, run the Blink code to ensure the board and the communications with the host computer are well established.

The connectors are a major source of faults. Before commencing any experiment please ensure the continuity of the wires. This can be done either using a multimeter, or connecting the on-board 5V supply to A0 and measuring the voltage. An improper connector would give us wrong voltages, or open connections..