Lab#1 GIT

Basics:

What is version control? Why is it important?

What's the difference between Git and GitHub?

Describe the Git workflow (add, commit, push, pull).

What is a repository in the context of Git?

Commits:

What is a commit in Git?

How is each commit uniquely identified?

Remote Repositories:

What is a remote repository in the context of Git?

What are the default names that Git uses for the repository you cloned from and your local repository?

How do you synchronize changes from a remote repository to your local one, and vice versa?

GitHub Specifics:

What is a pull request?

How do you 'fork' a repository on GitHub, and why might you want to?  
How can you use GitHub to collaborate on open-source projects?

Collaboration and Best Practices:

Why is it important to write clear commit messages?

When collaborating with others, why might it be important to frequently pull the latest changes?

ANSWERS:

BASICS:

1. Version control is a system that tracks and manages changes to files and code, enabling collaboration and keeping a history of revisions. It's important for collaboration, backup, and tracking changes in software development.
2. Git is a distributed version control system used for tracking changes in code, while GitHub is a web-based platform that provides hosting and collaboration features for Git repositories.
3. * + Add: Use git add to stage changes you want to include in the next commit.
     + Commit: After staging, use git commit to save the changes along with a descriptive message.
     + Push: With git push, you upload your committed changes to a remote repository, making them available to others.
     + Pull: git pull retrieves changes from a remote repository and incorporates them into your local branch.
4. A repository in the context of Git is a data structure that stores a collection of files, along with their revision history and metadata. It's where your project's source code and version history are managed, allowing you to track changes, collaborate with others, and maintain a complete history of your project.

COMMITS:

1. A commit in Git is a snapshot of the changes made to a Git repository at a specific point in time. It represents a single coherent set of changes to one or more files in the repository. Commits are accompanied by a commit message that describes the purpose or context of the changes. Commits are fundamental to tracking the history and version control of a Git project.
2. Each commit in Git is uniquely identified by a hexadecimal string called a "commit hash" or "commit SHA-1 hash." This hash is generated based on the content of the commit, including the changes made, the author's information, and the parent commit's hash. It ensures that no two commits in the history of a Git repository have the same identifier, making it possible to track and reference commits accurately.

REMOTE REPOSITORIES:

1. In the context of Git, a remote repository is a copy of a Git repository that is hosted on a different server or location. It serves as a centralized location where multiple developers can collaborate on a project. Remote repositories are used for sharing code, tracking changes made by others, and facilitating team collaboration. Common remote repository hosting services include GitHub, GitLab, and Bitbucket.
2. When you clone a Git repository, the default names used are:
   1. Remote Repository (origin): Git assigns the alias "origin" to the repository you cloned from. You can see this alias when you list remote repositories using the git remote -v command.
   2. Local Repository (the cloned directory): The local directory where you cloned the repository will have the same name as the remote repository unless you specify a different name during the cloning process.
3. To synchronize changes between a remote repository and your local repository in Git:
   1. Fetching Changes from Remote to Local:

* Use git fetch origin (or the name of the remote you want to fetch from) to retrieve changes from the remote repository to your local repository.
* To update your local branches with the changes from the remote, use git merge or git rebase after fetching.
  1. Pushing Changes from Local to Remote:
* After making changes locally, use git push origin <branch> to send your commits to the remote repository.
* Replace <branch> with the name of the branch you want to push.

GITHUB SPECIFICS:

1. A pull request is a feature offered by Git hosting platforms like GitHub, GitLab, and Bitbucket. It allows developers to propose changes to a repository hosted on these platforms.
2. To fork a repository on GitHub:
   1. Go to the GitHub page of the repository you want to fork.
   2. Click the "Fork" button in the top right corner of the repository's page. This will create a copy of the repository under your GitHub account.

You might want to fork a repository on GitHub for several reasons:

* Contribution: If you want to contribute to an open-source project, forking allows you to make changes to your copy without affecting the original project until your changes are ready.
* Experimentation: Forking provides a sandbox environment for experimenting with code. You can try out new features or changes without risk to the original project.
* Backup: It serves as a backup of a repository you're interested in or a starting point for your own projects based on someone else's work.
* Customization: You can fork a repository to customize it for your specific needs while retaining the ability to pull in updates from the original project.
* Collaboration: Forking can be a way to collaborate with others when working on a project together. Each collaborator can fork the repository, make changes, and submit pull requests to merge changes into the main project.
* Ownership: If you want to take over maintenance of a project that you didn't originally create, you can fork it to establish your own version.

3.To collaborate on open-source projects using GitHub, follow these steps:

* Fork the Repository:

Go to the GitHub page of the open-source project you want to contribute to.

Click the "Fork" button in the top right corner. This creates a copy of the repository under your GitHub account.

* Clone Your Fork Locally:

Clone your forked repository to your local machine using git clone.

* Create a Branch:

Create a new branch in your local repository for the specific feature or bug fix you want to work on using git checkout -b branch-name.

* Make Changes:

Make changes to the code in your branch, following the project's contribution guidelines.

* Commit Changes:

Use git commit to commit your changes with clear and descriptive commit messages.

* Push Changes:

Push your branch with changes to your fork on GitHub using git push origin branch-name.

* Create a Pull Request:

On your fork's GitHub page, click the "New Pull Request" button.

Select the original repository and the branch you want to merge your changes into.

Write a descriptive pull request message detailing your changes.

* Discuss and Review:

Collaborators and maintainers of the original project can review your changes, provide feedback, and discuss any necessary modifications.

* Address Feedback:

Make additional commits to address feedback and comments in your pull request.

* Merge Pull Request:

Once your pull request is approved, a maintainer of the original project can merge your changes into the main codebase.

COLLABORATION AND BEST PRACITCES:

1. Writing clear commit messages is important for several reasons:
   1. Communication: Commit messages serve as a form of communication among collaborators. Clear messages convey what changes were made, why they were made, and how they affect the codebase. This aids in understanding the purpose and context of each commit.
   2. Documentation: Commit messages form a historical record of changes to the codebase. They provide a chronological account of how the code has evolved over time, which is invaluable for troubleshooting, auditing, and understanding the project's history.
   3. Collaboration: In collaborative settings, clear commit messages make it easier for team members to review and understand your changes. This speeds up code reviews and reduces the likelihood of misinterpretations or misunderstandings.
   4. Bug Tracking: Well-documented commits can help trace the introduction of bugs or issues. By knowing what changes were made and why, it becomes easier to identify the source of problems and implement fixes.
   5. Code Maintenance: Clear commit messages aid in code maintenance. When revisiting code in the future or when someone else works on the code, a descriptive commit history can provide insights into past decisions and intentions.
   6. Project Onboarding: For new team members or contributors, clear commit messages are crucial for quickly understanding the codebase and the rationale behind previous changes. This streamlines the onboarding process.
   7. Open Source and Community Projects: In open-source projects, where contributors may not have direct access to each other, clear commit messages are essential for understanding and merging contributions from various sources.
2. Frequently pulling the latest changes when collaborating with others is important for several reasons:
   1. Synchronization: Pulling updates from the remote repository ensures that your local copy of the project is synchronized with the latest changes made by other collaborators. This prevents your local copy from becoming outdated.
   2. Avoiding Conflicts: Regularly pulling changes reduces the likelihood of encountering merge conflicts. Merge conflicts occur when two or more people make conflicting changes to the same part of a file. By pulling and merging frequently, you can address conflicts sooner and minimize their complexity.
   3. Timely Bug Fixes and Improvements: Frequent pulls allow you to access bug fixes, feature enhancements, and improvements made by others in real-time. This keeps your development environment up-to-date and allows you to benefit from the work of your collaborators.
   4. Collaboration Efficiency: Pulling the latest changes ensures that you and your collaborators are working on the same codebase. It facilitates smoother collaboration, as everyone is working with the most current version of the project.
   5. Continuous Integration: In projects that use continuous integration (CI), pulling the latest changes triggers automated tests and builds. This helps detect and fix issues early in the development process, leading to higher code quality.
   6. Preventing Divergence: Without frequent pulls, your local code may diverge significantly from the remote repository over time. This can make it challenging to merge your changes back into the project, leading to more complex and time-consuming integration efforts.
   7. Learning and Communication: Pulling updates allows you to see what changes others are making, helping you learn from their coding style, decisions, and contributions. It also facilitates communication and awareness of ongoing work within the team.

LAB#2 Arduino code to blink light and print status simultaneously.

Theory:

• The online simulator wokwi/tinkercad was used to perform the experiment.

• Arduino UNO board was used.

• A led was connected with the GND pin and the pin number 2 of the board.

• Code was written to generate a blinking effect by turning HIGH and LOW with a delay of 1000ms and print the status using Serial.print() method.

Code:

// Declare a variable for the LED pin

int ledPin = 13;

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

void setup() {

// Initialize the digital pin as an output for the LED.

pinMode(ledPin, OUTPUT);

// Initialize serial communication with a baud rate of 9600.

Serial.begin(9600);

}

// The loop function runs over and over again forever

void loop() {

// Turn the LED on

digitalWrite(ledPin, HIGH);

// Wait for one second

delay(1000);

// Turn the LED off

digitalWrite(ledPin, LOW);

// Wait for one second

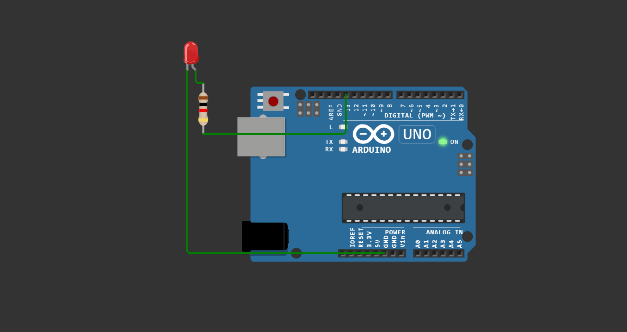
delay(1000);

// Send a message to the serial monitor on your computer

Serial.println("LED is blinking");

}

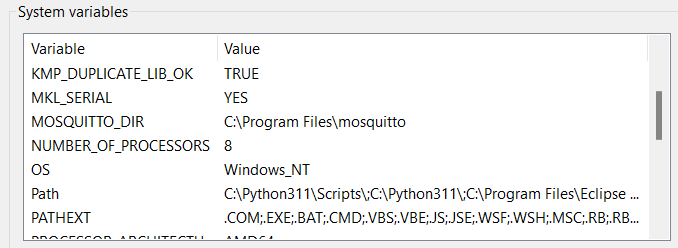
Screenshot:

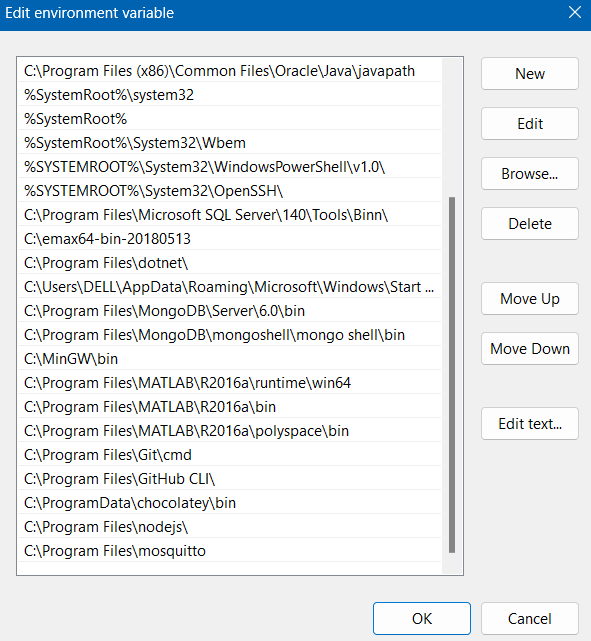


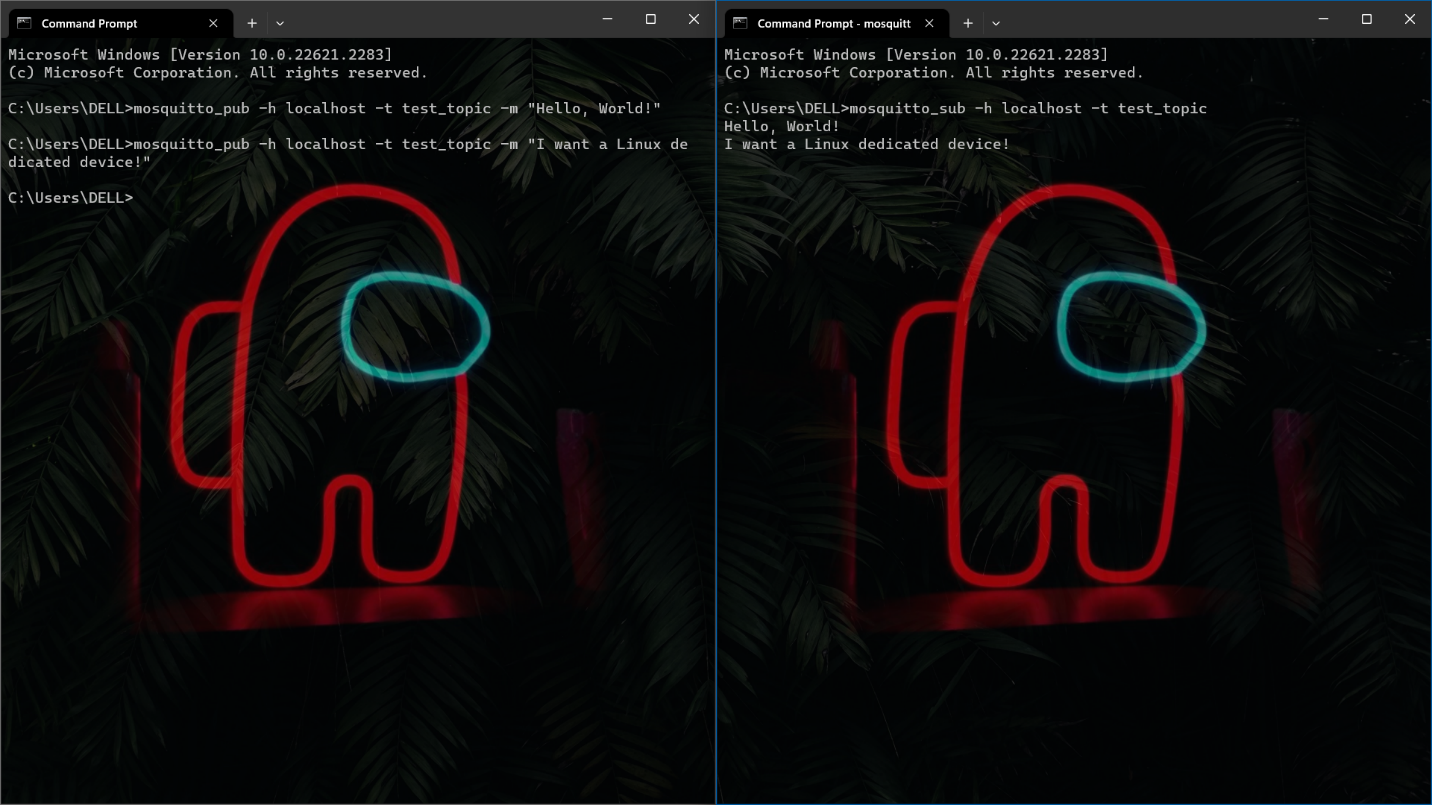
LAB#3 Mosquitto installation and testing

Procedure:

1. Go to <https://mosquitto.org/download/> and download file according to the operating system.
2. Follow the procedure provided here, <https://cedalo.com/blog/how-to-install-mosquitto-mqtt-broker-on-windows/> .
3. Now to access mosquitto without having to visit program files again and again.
   * We must add the mosquitto folder path to the environment variables in System Variables under Path.







Lab #4 Red MQTT Installation and testing

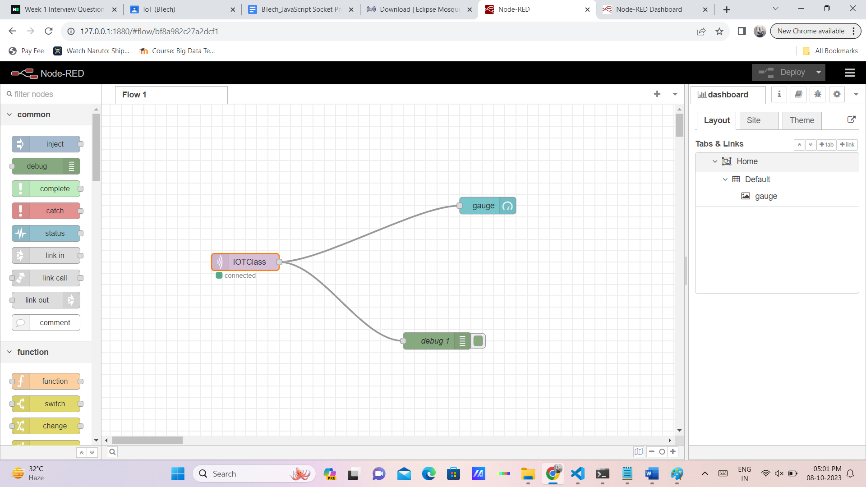
Theory:

RedMQTT is an application designed to read mobile sensor data and forward it to a local or cloud-based MQTT server.

Procedure:

1. Install RedMQTT on a smartphone.
2. Open node red in pc by typing ‘node-red’ in cmd. Also open hivemq broker.
3. Get the broker name, address and port from hivemq.
4. Open node red tab and add mqtt-in, gauge, and debug to the platform. Connect them as shown in the screenshot.
5. Double click on the mqtt-in. Add broker name, address and port. Then, add a topic name and save it.
6. Go to the RedMQTT app and go to settings. Add the same brover information.
7. Then, go to publish on the app. Select the broker name, enter the same topic name and select the sensor.
8. Now publish the node-red flow and the gauge will show the sensor data.

Screenshots:



#LAB 5

Aim: Interfacing DHT sensor with Arduino

Theory:

* DHT11 sensor measures and provides humidity and temperature values serially over a single wire.
* It can measure relative humidity in percentage (20 to 90% RH) and temperature in degree Celsius in the range of 0 to 50°C.
* It has 4 pins; one of which is used for data communication in serial form.

Code:

#include "DHT.h"

DHT dht;

void setup()

{

Serial.begin(9600);

Serial.println();

Serial.println("Status\tHumidity (%)\tTemperature (C)\t(F)");

dht.setup(2); /\* set pin for data communication \*/

}

void loop()

{

delay(dht.getMinimumSamplingPeriod()); /\* Delay of amount equal to sampling period \*/

float humidity = dht.getHumidity(); /\* Get humidity value \*/

float temperature = dht.getTemperature(); /\* Get temperature value \*/

Serial.print(dht.getStatusString()); /\* Print status of communication \*/

Serial.print("\t");

Serial.print(humidity, 1);

Serial.print("\t\t");

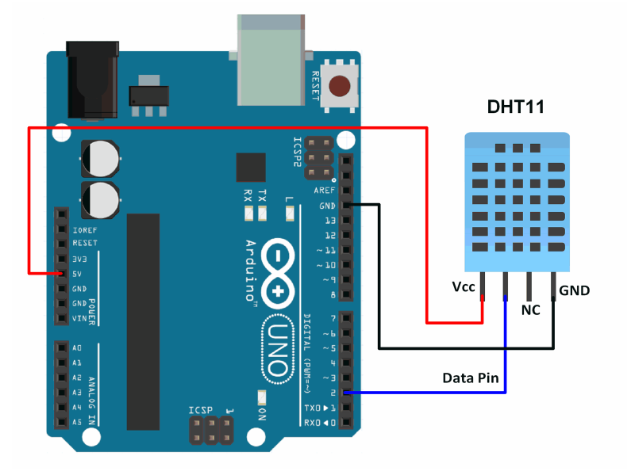
Serial.print(temperature, 1);

Serial.print("\t\t");

Serial.println(dht.toFahrenheit(temperature), 1); /\* Convert temperature to Fahrenheit units \*/

}

Output:



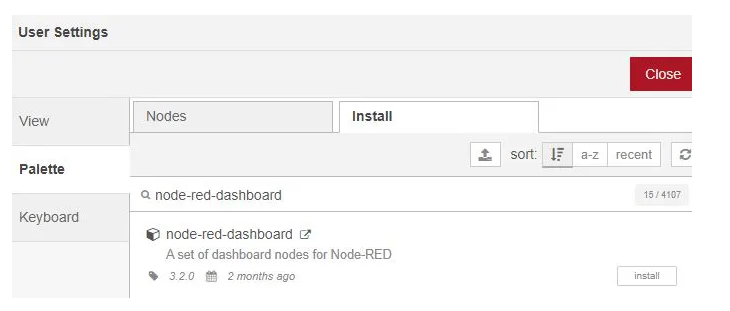
#LAB 6

Aim: Gathering DHT sensor data and visualizing it using an app built by Node-RED.

Theory: nil

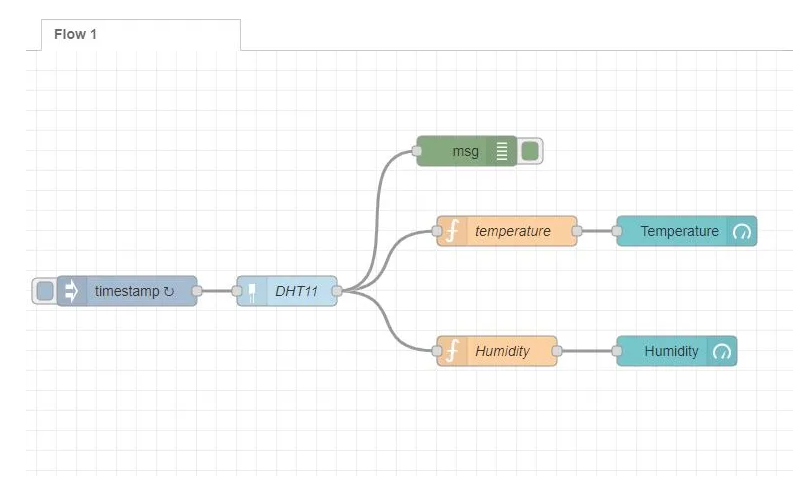
Procedure:

* Open palette and install node red dashboard along with node-red-contrib-dht-sensor

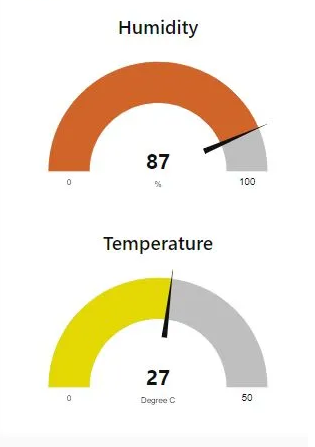


* Create the flow like in the image using the following tags:
  + Inject
  + Function
  + Debug
  + rpi-DHT22
  + Gauge

Set the properties of above tags as said in the following link: https://iotstarters.com/building-node-red-dashboard-with-dht11-sensor/



Output:



#LAB 7

Aim: making a led blind only with a switch

Theory: nil

Code:

const int switchPin = 2; // Pin connected to the external switch

const int ledPin = 13;   // Pin connected to the LED

int switchState = HIGH;   // Variable to store the current state of the switch

int lastSwitchState = HIGH; // Variable to store the previous state of the switch

void setup() {

  pinMode(switchPin, INPUT); // Set the switch pin as INPUT

  pinMode(ledPin, OUTPUT);   // Set the LED pin as OUTPUT

}

void loop() {

  // Read the state of the switch

  switchState = digitalRead(switchPin);

  // If the switch state has changed (button released), start blinking the LED

  if (switchState != lastSwitchState && switchState == HIGH) {

    while (switchState == HIGH) {

      digitalWrite(ledPin, HIGH); // Turn the LED on

      delay(500);                  // Wait for 500 milliseconds (0.5 seconds)

      digitalWrite(ledPin, LOW);  // Turn the LED off

      delay(500);                  // Wait for another 500 milliseconds

      switchState = digitalRead(switchPin); // Read the state of the switch again

    }

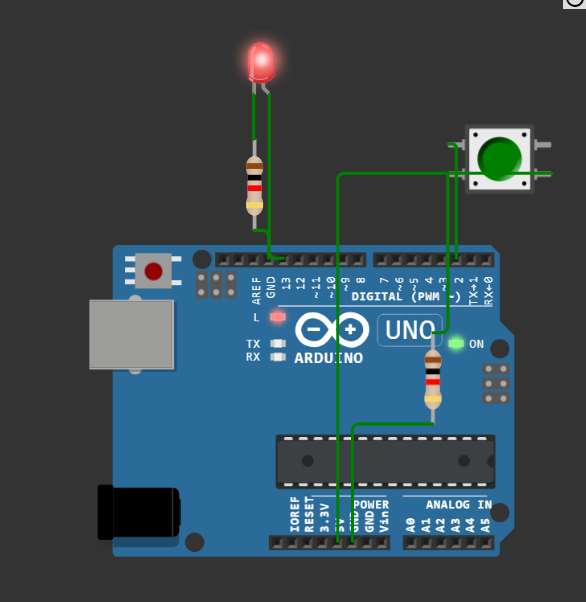
  }

  // Save the current switch state for the next iteration

  lastSwitchState = switchState;

}

Output:



#Lab 8

Aim: led and dht work alternatively using push button

Theory: nil

Code:

#include <DHT.h>

const int buttonPin = 2;  // Pin connected to the push button

const int ledPin = 13;    // Pin connected to the LED

const int dhtPin = 3;     // Pin connected to the DHT sensor

DHT dht(dhtPin, DHT22); // Create a DHT object

int buttonState = HIGH;  // Variable to store the current state of the button

int lastButtonState = HIGH; // Variable to store the previous state of the button

int currentState = 0; // Variable to keep track of the current state

void setup() {

  pinMode(buttonPin, INPUT);   // Set the button pin as INPUT

  pinMode(ledPin, OUTPUT);     // Set the LED pin as OUTPUT

  pinMode(dhtPin, INPUT);      // Set the DHT sensor pin as INPUT

**Serial**.begin(9600);          // Initialize serial communication

  dht.begin();                 // Initialize the DHT sensor

}

void loop() {

  // Read the state of the button

  buttonState = digitalRead(buttonPin);

  // If the button state has changed (button pressed), toggle the state

  if (buttonState != lastButtonState && buttonState == LOW) {

    currentState = 1 - currentState; // Toggle between 0 and 1

    // Perform actions based on the current state

    if (currentState == 0) {

      // State 0: Blink the LED

      blinkEveryTwoSeconds();

    } else {

      // State 1: Print DHT sensor information

      printDHTSensorInfo();

    }

  }

  // Save the current button state for the next iteration

  lastButtonState = buttonState;

}

void blinkEveryTwoSeconds() {

  digitalWrite(ledPin, HIGH); // Turn the LED on

  delay(1000); // Wait for 1 second

  digitalWrite(ledPin, LOW); // Turn the LED off

  delay(1000); // Wait for 1 second

}

void printDHTSensorInfo() {

  // Read temperature and humidity from the DHT sensor

  float temperature = dht.readTemperature();

  float humidity = dht.readHumidity();

  // Print the sensor information to the Serial Monitor

**Serial**.print("Temperature: ");

**Serial**.print(temperature);

**Serial**.print(" °C, Humidity: ");

**Serial**.print(humidity);

**Serial**.println(" %");

  // Add any additional actions based on the sensor data, e.g., controlling an LED

  if (temperature > 25.0) {

    digitalWrite(ledPin, HIGH); // Turn on the LED if the temperature is above 25.0 °C

  } else {

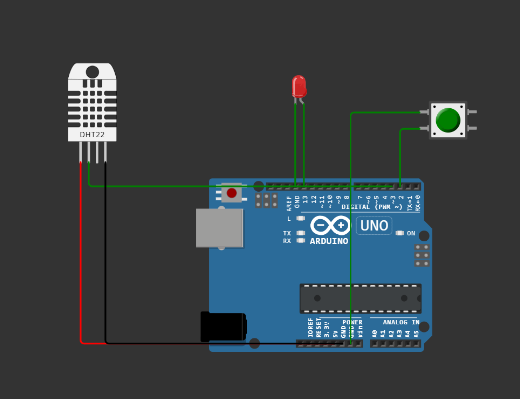
    digitalWrite(ledPin, LOW);  // Turn off the LED otherwise

  }

  delay(2000); // Wait for 2 seconds before printing the next sensor information

}

Output:



#Lab 9

Aim: using ultrasonic sensor and getting a reading out of it

Theory: nil

Code:

#define triggerpin 3

#define echopin 2

int dist;

long dur;

void setup() {

pinMode(triggerpin, OUTPUT);

pinMode(echopin, INPUT);

**Serial**.begin(9600);

**Serial**.println("Measurement begins");

delay(100);

}

void loop() {

  digitalWrite(triggerpin,LOW);

  delayMicroseconds(2);

  digitalWrite(triggerpin, HIGH);

  delayMicroseconds(5);

  digitalWrite(triggerpin,LOW);

  delayMicroseconds(2);

  dur=pulseIn(echopin,HIGH);

  dist=dur\*0.344/2;

**Serial**.print("Distance is: ");

**Serial**.print(dist);

**Serial**.print("cm");

}

Output:

3cm

1432cm

1555cm

.

.

.

.

.

