Document: Comprehensive Summary of Hardware of the Project Development

By:- Aayush Jain

Introduction:-

This document provides a detailed summary of the project development process for setting up sensors on a Raspberry Pi, transmitting the data via MQTT and later on to AWS IoT, and displaying the data on a web interface. It covers the various discussions, questions I had in my mind, approaches taken, changes made, errors faced, and the final outcomes.

This Document Is only thing you will need to refer for all the details regarding the Hardware side of the project

Hardware Used Initially For First Prototype :-

- Raspberry Pi 4b 4GB Ram
- HTU21D Temperature Sensor
- DHT11 Sensor
- Jumper Wires
- Breadboard

Below is The Description Of The Sensors Used

HTU21D

The HTU21D is a digital humidity and temperature sensor manufactured by TE Connectivity. It offers high precision and reliability in measuring humidity and temperature, making it suitable for a wide range of applications.

Features

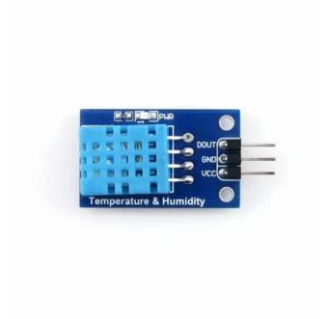
1. **High Accuracy**: Provides accurate readings of humidity (±2% RH) and temperature (±0.3°C).
2. **Digital Output**: Communicates via I2C, providing easy integration with microcontrollers and other digital systems.
3. **Low Power Consumption**: Suitable for battery-powered devices.
4. **Compact Size**: Small form factor, ideal for space-constrained applications.
5. **Fast Response Time**: Quick measurement of humidity and temperature changes.

6. **Calibration**: Factory-calibrated, ensuring accuracy out-of-the-box.
7. **Wide Operating Range**: Operates over a wide range of temperatures and humidity levels.

Specifications

- **Supply Voltage**: 1.5V to 3.6V
- **Humidity Range**: 0% to 100% RH
- **Temperature Range**: -40°C to 125°C
- **Humidity Accuracy**: ±2% RH
- **Temperature Accuracy**: ±0.3°C
- **Communication Interface**: I2C
- **Response Time**: 5 seconds for humidity, 2 seconds for temperature
- **Power Consumption**: 2.7 µW (average)

DHT11 Sensor(Please Don’t use directly in Applications where you need high accuracy)



The DHT11 is a basic, low-cost digital temperature and humidity sensor. It provides reliable data and is widely used in hobbyist and educational projects.

Features

1. **Digital Output:** Simple communication via a single-wire digital interface.
2. **Affordable:** Low cost makes it accessible for a wide range of applications.
3. **Easy to Use:** Requires minimal additional components and coding effort.
4. **Compact Size:** Small and lightweight, easy to integrate into various projects.
5. **Built-in Pull-up Resistor:** Simplifies the circuit design.

Specifications

- **Supply Voltage:** 3.3V to 5V
- **Humidity Range:** 20% to 90% RH
- **Temperature Range:** 0°C to 50°C
- **Humidity Accuracy:** ±5% RH
- **Temperature Accuracy:** ±2°C
- **Communication Interface:** Single-wire digital
- **Response Time:** 6-10 seconds for humidity, 4-8 seconds for temperature
- **Power Consumption:** 0.3 mW (average)

Note :- **RH** in case of Humidity refers to **Relative Humidity**. It is a measure of the amount of moisture (water vapor) present in the air relative to the maximum amount of moisture that the air can hold at a given temperature. It is expressed as a percentage (%).

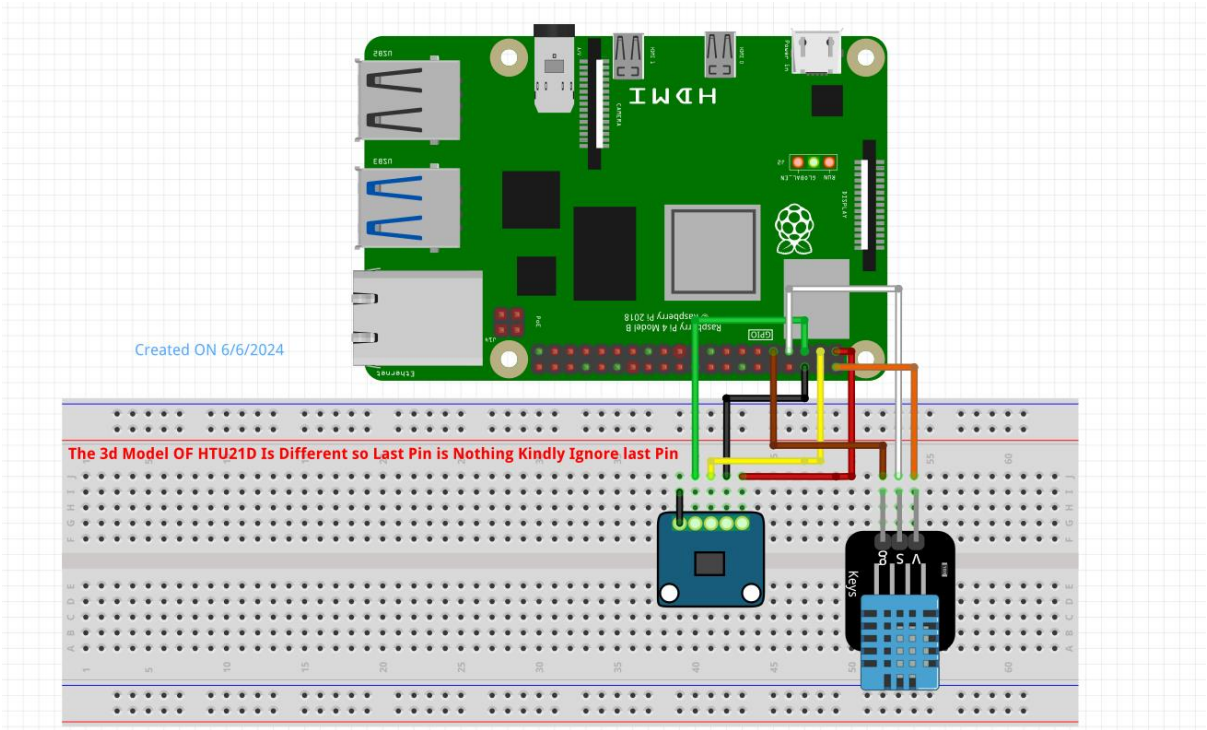
Important :- Before taking readings from DHT11 Let it run for a minute or two so it can heat up a bit this will ensure you get proper readings from the Module

Below I have created a Comparison Chart for Both The Sensors

Here is the table content:

Aspect	HTU21D	DHT11
Measurement Range (Temperature)	-40 to +125°C	0 to +50°C
Measurement Range (Humidity)	0% to 100% RH	20% to 90% RH
Accuracy (Temperature)	±0.3°C	±2°C
Accuracy (Humidity)	±2% RH	±5% RH
Response Time	5 seconds for humidity, 2 seconds for temperature	6-10 seconds for humidity, 4-8 seconds for temperature
Operating Voltage	1.5V to 3.6V	3V to 5.5V
Power Consumption	Very low (typically 2.7µW at 1Hz measurement)	Low (2.5mW max)
Communication Protocol	I2C	Single-wire
Cost	Higher	Lower

Here is The Circuit Diagram I Created for the Current Setup



Stage 1 :-

This Included Just taking the raw data from the sensors and displaying them on the terminal of the pi4 to just check the values of the sensors and know how they perform. The Connections were made according to above schematic

Make sure to Install the required for the below code

Prerequisite :- Make sure I2C AND GPIO are enabled from your raspberry pi configuration

Step 1 : - Install the library for htu21d and dht11 using following command

pip install adafruit-circuitpython-htu21d adafruit-circuitpython-dht

Step 2 :- Create a file name example sensors.py to do that use following command

nano sensors.py

Step 3 :- Copy The Following Code In The Editor :-

Drive Link Is Attached :- https://drive.google.com/file/d/1QWegO1a9NGaK9kgX9-QE63driP8pCqac/view?usp=sharing

Step 4 :- run it using command :- python sensors.py

Step 5 :- Voila The sensors value will start displaying in The Terminal

This was the very first thing that was done on getting the sensors later on in further stages more changes are made

Stage 2 :-

This stage of the project involved to filter out the values of the sensors so we can gett accurate readings as well as implementing AI model to correct the values of the Less accurate DHT11 Sensor on the basis of HTU21D Values

Following Additional Features Were Implemented ➖

1. **Moving Average Filter** :- As part of my project involving the HTU21D and DHT11 sensors for temperature and humidity readings, I've implemented a moving average filter to enhance the accuracy and reliability of the sensor data. This filter is crucial in reducing noise and smoothing out short-term fluctuations, which can otherwise lead to misleading interpretations and suboptimal performance of machine learning models.

Why Implement a Moving Average Filter?

The primary reason for using a moving average filter in this project is to ensure that the data fed into the machine learning models and other downstream processes is as accurate and clean as possible. Sensor readings, especially from relatively inexpensive sensors like the DHT11, can be noisy due to various factors such as electrical interference, environmental conditions, and inherent sensor limitations.

By applying a moving average filter, I can achieve the following:

1. **Noise Reduction**: Minimise random fluctuations in the sensor data, making it easier to detect true changes in the environment.
2. **Smoothing Data**: Provide a more stable and reliable set of readings for further analysis and model training.
3. **Improving Model Accuracy**: Feed cleaner data into the machine learning models, which helps improve their accuracy and generalisation.

How the Moving Average Filter Works in My Code

Here's a detailed explanation of how the moving average filter is integrated into my Python script for the Raspberry Pi:

1. **Initialization**:
    - I start by defining a window size for the moving average filter. This window size determines the number of recent data points to consider for each average calculation. In my script, I've set this window size to 5.
    - Then use deque from the collections module to maintain the recent readings for both temperature and humidity from the HTU21D and DHT11 sensors.
2. **Function Definition**:

- I define a function moving_average(data_array) that takes an array of data points as input and returns the average. If the array is empty, it returns 0. This ensures that the function can handle edge cases gracefully.

The above mentioned is the part from the code that handles data

3. **Applying the Filter**:

- After collecting new sensor readings, I append these readings to the corresponding deques.
- I then call the moving_average function on these deques to calculate the smoothed values.

Smoothing Process:

- The moving_average function calculates the mean of the data points in the deque. By averaging these points, the function reduces the impact of any single noisy reading, resulting in a smoother signal

Benefits in The Project

- **Improved Data Quality**: The moving average filter helps in providing more consistent and reliable data from the sensors, which is essential for accurate analysis and model training.
- **Enhanced Model Performance**: By feeding smoother data into the machine learning models, the models can learn better and make more accurate predictions. This is particularly important for correcting DHT11 readings using HTU21D data.
- **Robustness to Noise**: The filter makes the system more robust to random noise and transient spikes, ensuring that the sensor readings reflect the true environmental conditions more closely.

2. Training Machine Learning Models

As part of our project involving the HTU21D and DHT11 sensors for temperature and humidity readings, We have integrated machine learning models to improve the accuracy of the DHT11 sensor readings. By training these models, we aim to correct the DHT11 sensor data using the more accurate HTU21D sensor data. Here's a detailed explanation of the training process from my perspective.

Why Train Machine Learning Models?

The DHT11 sensor, while cost-effective, often provides less accurate readings compared to the HTU21D sensor. To enhance the reliability of the DHT11 data, We decided to use

machine learning models to correct its readings based on the more accurate data from the HTU21D sensor. The trained models can predict the corrected temperature and humidity values from the raw DHT11 sensor data, thus improving overall data quality.

Steps in the Training Process

1. **Data Collection**:
    - We first collect data from both the HTU21D and DHT11 sensors. This data includes temperature and humidity readings.
    - The data is continuously logged and stored in a CSV file named sensor_readings.csv.
2. **Preprocessing**:
    - The collected data is preprocessed to remove any outliers and to smooth the readings using a moving average filter. This ensures that the data used for training is clean and representative of the actual environmental conditions.
    - Missing values are handled, and any rows with incomplete data are dropped.
3. **Feature Extraction**:
    - For temperature correction, the DHT11 temperature readings are used as the feature, and the HTU21D temperature readings are used as the target.
    - Similarly, for humidity correction, the DHT11 humidity readings are used as the feature, and the HTU21D humidity readings are used as the target.
4. **Splitting the Data**:
    - The data is split into training and testing sets. This is done to ensure that the models are trained on a portion of the data and validated on another portion to assess their performance.
    - We use an 80-20 split, where 80% of the data is used for training and 20% for testing.
5. **Training the Models**:
    - We use the RandomForestRegressor from the sklearn library to train the models. This algorithm is chosen for its robustness and ability to handle non-linear relationships.
    - The training involves fitting the model on the training data to learn the mapping from the DHT11 readings to the HTU21D readings.

Model Validation:

- The trained models are validated using the testing set to assess their accuracy. The performance is measured using the R-squared score, which indicates how well the models can predict the corrected values.
- The results are logged to provide insights into the model's performance.

Saving the Models:

- Once trained, the models are saved to disk using joblib. This allows the models to be loaded and used for real-time correction of DHT11 readings without retraining.

Benefits of the Machine Learning Approach

- **Enhanced Accuracy**: The corrected DHT11 readings closely match the more accurate HTU21D readings, providing more reliable data.
- **Robustness**: The use of RandomForestRegressor helps in handling outliers and non-linear relationships, making the models robust to variations in the data.
- **Real-Time Correction**: The trained models can be used in real-time to correct the DHT11 readings as new data is collected, ensuring continuous accuracy.
- **Scalability**: The models can be retrained periodically as more data is collected, allowing them to adapt to any changes in sensor behaviour or environmental conditions.

3. MQTT Setup

As part of the project involving the HTU21D and DHT11 sensors, MQTT (Message Queuing Telemetry Transport) plays a crucial role in facilitating the communication between my Raspberry Pi, where the sensors are connected, and AWS IoT, which I use for data storage, processing, and analysis. Here's a detailed explanation of how MQTT is integrated into my project and why it is essential.

Why MQTT?

MQTT is a lightweight messaging protocol designed for low-bandwidth, high-latency, or unreliable networks. It is ideal for IoT applications due to its efficiency and ease of use. The key reasons for choosing MQTT in my project are:

- **Efficiency**: MQTT minimises the amount of data transmitted over the network, making it suitable for scenarios where bandwidth is limited.
- **Simplicity**: The protocol is simple to implement, allowing for quick and straightforward integration with IoT devices and cloud services.
- **Reliability**: MQTT supports Quality of Service (QoS) levels, ensuring that messages are delivered reliably even in case of network disruptions.
- **Scalability**: MQTT is designed to handle numerous devices and messages, making it scalable for larger IoT deployments.

How MQTT is Integrated into My Project

1. **Setting Up MQTT on Raspberry Pi**:
    - On my Raspberry Pi, where the HTU21D and DHT11 sensors are connected, I use the AWS IoT Python SDK to establish an MQTT connection with AWS IoT Core. This connection allows my Pi to publish sensor data to AWS IoT and subscribe to relevant topics for receiving messages.

Below mentioned is the code for the same

1. Initial Setup and Challenges

Objective: To set up sensors (HTU21D and DHT11) on a Raspberry Pi to collect temperature and humidity data.

Initial Challenges:

- **Issues with Installing Sensor Libraries**: One of the first challenges I faced was installing the necessary sensor libraries. The adafruit_htu21d and adafruit_dht libraries were not initially installed, leading to import errors.
- **Permission Errors**: I encountered permission errors while accessing the hardware components of the Raspberry Pi, which required running the script with elevated privileges using sudo. So make sure while running commands to run with sudo and add your user to sudoers group

2. Sensor Data Collection

Approach:

- I interfaced the sensors using the I2C and GPIO pins of the Raspberry Pi. I used the adafruit_htu21d and adafruit_dht libraries to read data from the HTU21D and DHT11 sensors, respectively.
- To ensure reliable data collection, especially for the DHT11 sensor which is known for intermittent read failures, I implemented error handling and retries.

Changes Made:

- **Runtime Error Handling**: I adjusted the code to handle runtime errors that commonly occur with the DHT11 sensor.
- **Optimization**: I optimised the data reading intervals to reduce CPU usage and ensure efficient data collection.

3. Data Processing and Correction

Objective: To correct the DHT11 sensor readings using HTU21D data.

Approach:

- **Moving Averages and Outlier Detection**: I implemented moving averages and outlier detection to preprocess the sensor data. This step helped in smoothing out the noise and removing any anomalous readings.
- **Random Forest Model**: I trained a Random Forest model to correct the DHT11 sensor readings based on the HTU21D data. The model was trained using a supervised learning approach, where HTU21D readings were used as the ground truth.

Changes Made:

- **Data Preprocessing**: I added code for data preprocessing using moving averages and outlier detection.
- **Model Training and Retraining**: I implemented logic for training the Random Forest model and retraining it periodically after collecting a specified number of new data points.

4. Data Transmission to AWS IoT

Objective: To transmit the collected and corrected data to AWS IoT.

Approach:

- **MQTT Communication**: I set up MQTT communication with AWS IoT to transmit the sensor data. I used AWS credentials and certificates for secure communication.
- **AWS IoT Setup**: I used AWS IoT Core for device management and data transmission.

Challenges:

- **Certificate Paths**: I faced issues with specifying the correct paths for the AWS IoT certificates and keys.
- **MQTT Connection Errors**: I encountered connection errors while setting up the MQTT client.

Changes Made:

- **Corrected Certificate Paths**: I ensured that the certificate file paths were correctly specified in the code.
- **Error Logging and Handling**: I added comprehensive error logging and handling for connection retries and other potential issues.

5. Web Interface for Data Visualization

Objective: To display the sensor data on a web interface.

Initial Approach: I planned to use Flask and Socket.IO to serve a web page and update data in real-time.

Challenges:

- **Live Data Updates**: Initially, I faced issues with updating the data on the web page in real-time.
- **Graph Data Retention**: The graphs were not retaining historical data, making it difficult to visualize trends over time.

Final Approach:

- **AWS DynamoDB**: I used AWS DynamoDB to store historical sensor data.
- **Chart.js for Visualization**: I updated the web interface to fetch data from AWS and display it using Chart.js for interactive and real-time data visualization.

Changes Made:

- **Flask Endpoints**: I modified Flask endpoints to fetch data from AWS.
- **HTML and JavaScript Updates**: I updated the HTML and JavaScript code to ensure real-time data visualization and historical data retention.

6. Deployment

Objective: To deploy the web interface using Vercel.

Challenges:

- **Vercel Configuration**: Configuring Vercel for deploying the Flask application was a challenge.
- **Environment Variables**: I had to handle environment variables and AWS credentials securely during deployment.

Changes Made:

- **Vercel Configuration File**: I created a `vercel.json` configuration file for the deployment setup.
- **Environment Variables**: I set up environment variables on Vercel to manage AWS credentials securely.

But there was an issue in the approach using vercel. Why ? . The reason being that vercel can be used only for stratic webpages and we can't run our own server of flask in order to receive the values from the server . So i had to try a different approach . i used a website called pythonanywhere.com for the same it allowed me to host a flask server so we could fetch data from aws.

http://krrishverma.pythonanywhere.com/

This is the current dashboard for the sensor data reading

Stage 3:-

Now that we had achieved the basic functioning of our idea demonstartion with help of sensors now we wanted to measure the current and voltage characteristics of the mosfets now.

Now for the Mosfets we can collect and map the following data on the graphs

1. Transfer Characteristics (Id vs. Vgs)
2. Output Characteristics (Id vs. Vds)
3. Transconductance (gm vs. Vgs)
4. On-Resistance (Rds(on) vs. Vgs)

The Above mentioned parameters are what we need to plot for the mosfet to check for different outputs based on Different values of Vgs and Vds

First Approach We Took ➖

Hardware Used:

1. ACS712 Current Sensor Module
2. Load Resistor
3. Pi4
4. MCP3008 ADC(Analog to Digital Converter)
5. Jumper Wires
6. A P or N Channel MOSFET

Challenges Faced :-

1. **PI 4 does not have inbuilt ADC so it cannot accept analog inputs directly on tis pin that is why we are using an external ADC here.**
2. **Resolution Of ADC :- Currently MCP3008 has only 10 Bit Resolution however to measure changes properly with high accuracy we**

Introduction

This section provides a comprehensive overview of the discussions and findings related to the process of characterizing MOSFET parameters using lab bench power supplies, a Raspberry Pi, an MCP3008 ADC, and a current sensor. The goal is to measure and plot key MOSFET parameters such as transfer characteristics (Id vs. Vgs) and output characteristics (Id vs. Vds) accurately. This document outlines the motivation, approaches taken, challenges faced, and solutions implemented throughout the process.

Motivation

The need to fabricate custom sensors and measure specific parameters of MOSFETs necessitated a comprehensive approach to collect and analyze data. Given the unavailability of DACs, we explored using lab bench power supplies to manually set Vgs and Vds, with an automated data acquisition system for live monitoring and recording.

Equipment and Components

- **Lab Bench Power Supplies**: For providing variable Vgs and Vds.
- **Raspberry Pi**: For data acquisition and control.
- **MCP3008 ADC**: For reading analog values (Id).
- **Current Sensor (ACS712)**: For measuring the drain current (Id).
- **MOSFET**: The device under test.
- **Connecting Wires and Breadboard**: For circuit connections.
- **Multimeter**: Optional, for verification.
- **Voltage Sensor:** We were using resistors to make one of our own

Experimental Setup

Circuit Connections

1. **Vgs Supply**:
    - Positive terminal connected to the gate of the MOSFET through a 10kΩ resistor.
    - Negative terminal connected to ground.
    - Vgs measurement point connected to ADC channel (CH1).

2. **Vds Supply**:
    - Positive terminal connected to the drain of the MOSFET.
    - Negative terminal connected to ground.
3. **Current Sensor (ACS712)**:
    - VCC connected to 5V.
    - GND connected to ground.
    - OUT connected to ADC channel (CH0).
4. **MCP3008 ADC to Raspberry Pi**:
    - VDD connected to 3.3V or 5V (depending on the reference voltage).
    - GND connected to ground.
    - SCLK connected to SCLK (GPIO 11, Pin 23).
    - MISO connected to MISO (GPIO 9, Pin 21).
    - MOSI connected to MOSI (GPIO 10, Pin 19).
    - CS connected to CE0 (GPIO 8, Pin 24).

Data Collection

- The Raspberry Pi reads live Vgs and Id values via the MCP3008 ADC.
- The bench power supplies manually set the Vgs and Vds values.
- A Python script handles data acquisition, real-time monitoring, and plotting.

Key Factors to Consider

1. **Current Range**: The expected range of current you need to measure.
2. **Resolution and Precision**: The level of accuracy and detail required in your measurements.
3. **Voltage Range**: The voltage levels in your application, particularly the drain-to-source voltage of the MOSFET.
4. **Design Complexity**: The ease of integration into your existing circuit.

INA228 vs. INA260 for Measuring MOSFET Drain Current

INA228

- **Resolution**: 20-bit ADC provides very high resolution, making it suitable for applications requiring precise current measurements.
- **Shunt Resistor**: Requires an external shunt resistor, offering flexibility to choose a value that matches your current measurement needs. For instance, a 15 mΩ shunt resistor gives extremely fine resolution but limits the maximum measurable current.
- **Voltage Range**: Can measure bus voltages up to 85V, which is useful for high-voltage applications.
- **Offset Voltage**: Lower typical input offset voltage (±2.5 µV) leads to lower offset current, enhancing precision in low-current measurements.
- **Flexibility**: Highly adaptable to various current ranges due to the external shunt resistor.

INA260

- **Resolution**: 16-bit ADC, which is generally sufficient for most applications but offers less resolution compared to the INA228.
- **Shunt Resistor**: Integrated 2 mΩ shunt resistor simplifies the design but limits flexibility. This setup is straightforward but might not be as precise for low-current measurements.

- **Voltage Range**: Measures bus voltages up to 36V, suitable for many standard applications but limited compared to the INA228.
- **Offset Voltage**: Higher typical input offset voltage (±10 µV) results in a higher offset current, which can be significant in very low-current measurements.
- **Design Simplicity**: Easier to integrate due to the built-in shunt resistor, making it suitable for applications where design simplicity and robustness are critical.

Practical Examples

High-Precision, Low-Current Measurement (INA228)

If you need to measure very low currents with high precision, such as in a scientific instrument or precision industrial application, the INA228 is a better choice.

- **Example Setup**: Use a 15 mΩ shunt resistor for very fine resolution.
  - **Resolution**: Approximately 0.0208 µA.
  - **Maximum Current**: Around 10.9227 A.
  - **Applications**: Precision monitoring of MOSFET drain current in sensitive circuits.

High-Current, Standard Precision Measurement (INA260)

If you need to measure higher currents with adequate precision and simplicity, such as in power electronics or battery management systems, the INA260 is more suitable.

- **Example Setup**: Integrated 2 mΩ shunt resistor.
  - **Resolution**: 1.25 µA.
  - **Maximum Current**: 41 A.
  - **Applications**: General-purpose monitoring of MOSFET drain current in power circuits, automotive applications, and embedded systems.

Conclusion

- **INA228**: Best for high-precision, low-current measurements where flexibility and accuracy are critical. Suitable for applications where precise control and monitoring of MOSFET drain current are essential.
- **INA260**: Best for high-current measurements with sufficient precision and simpler design requirements. Ideal for applications where ease of integration and robust high-current measurements are needed.

  Second Approach We took-

  Hardware Used-

  Raspberry Pi

  Breakout board

  INA260

  Connecting Wires

  Breadboard