



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

ARDUINO-BASED CUSTOMIZABLE HANDMADE MULTIMETER

ANALOG CIRCUITS
SUBJECT CODE: BECE206L
Slot: D1 + TD1

LOKESH R (23BEC1009)

GAUTHAM R (23BEC1069)

LALITH KISHORE (23BEC1088)

A V SIVA VARSHAN (23BEC1262)

Submitted to

Prof Chandramauleshwar Roy

CONTENTS

PART OF THE REPORT	PAGE NO
Introduction	3
Motivation and purpose	3
Aim	4
Components Required	4
Theory	5
Hardware Connections	10
Arduino Code	11
Gains from the Project	32
Conclusion	34
Future Scope	34
References	34

ARDUINO-BASED CUSTOMIZABLE HANDMADE MULTIMETER PROJECT

INTRODUCTION

In the field of electronics and electrical engineering, **multimeters** are one of the most essential tools for measuring a wide range of electrical parameters. These devices play a crucial role in **troubleshooting circuits**, **verifying component functionality**, and **performing routine measurements** during the design and development of electronic systems. Commercially available multimeters are widely used across industries and educational institutions due to their convenience, compactness, and ease of use. However, there are certain limitations to these pre-built instruments, particularly when it comes to understanding the **underlying working principles** and the **flexibility** of their design.

This project aims to address these challenges by developing a **handmade multimeter** that offers the same fundamental measurement capabilities as commercial multimeters, but with added benefits that enhance **learning**, **customization**, and **educational value**. The goal of this multimeter is not only to create a functional tool for electrical measurements but also to serve as an educational project that allows users to gain deeper insights into **electronics**, **sensors**, and **signal processing**.

MOTIVATION AND PURPOSE OF THE PROJECT

While commercial multimeters are widely available, they are often **black boxes** for users, where the inner workings remain hidden. Users typically rely on the multimeter's display without understanding the principles behind the readings. This is particularly true for students, hobbyists, and beginners who want to learn the **fundamentals** of electronics and measurement techniques but may be limited by the **complexity** or **cost** of commercial tools. In contrast, a handmade multimeter serves as an **educational tool** that allows users to gain hands-on experience in both the **hardware design** and **software programming** needed to make accurate electrical measurements.

By building a multimeter from the ground up, this project offers a **deeper understanding** of how measurements are made. It enables users to experiment with key components such as **sensors**, **analog-to-digital conversion**, and **signal conditioning**, which are often abstracted away in commercial products. This approach helps develop a stronger foundation for those interested in pursuing more advanced work in fields such as **embedded systems**, **analog electronics**, and **instrumentation design**.

Another important motivation for this project is the **cost factor**. Commercial multimeters can be expensive, particularly those with advanced features, making them inaccessible for many students, hobbyists, or DIY enthusiasts. By building a multimeter from readily available and low-cost components like **Arduino** boards, **resistors**, **sensors**, and **displays**, this project provides a more affordable alternative for individuals or institutions that may not have access to high-end commercial tools. It serves as a **cost-effective solution** for educational and practical use, without sacrificing the capability to perform important electrical measurements.

Additionally, commercial multimeters, while highly efficient, often lack flexibility in terms of **customization**. Each user may have different needs, whether it's a specific measurement range,

particular sensor integrations, or additional functionalities (e.g., the ability to generate an AC signal or measure frequency). This handmade multimeter can be **tailored to specific needs**, making it an ideal choice for applications where **personalization** or the ability to modify the device for specialized tasks is essential. It can also serve as a **platform for learning and experimentation**—users can modify the design, add new features, or change its measurement capabilities as their knowledge and requirements evolve.

AIM

This project aims to design and implement a cost-effective, multifunctional Arduino-based multimeter tailored for both educational and practical use. This versatile device will measure **essential electrical parameters**—voltage, current, resistance, temperature, Frequency and continuity—offering a broad range of applications in electronics labs, workshops, and educational settings.

Additionally, for generating **AC inputs** for testing the multimeter a DAC is constructed with **R-2R ladder**. The ability to measure the frequency of this AC signal is an essential feature of the multimeter, allowing it to be more versatile in testing electronic components that work with alternating currents. This component also provides valuable hands-on experience with **AC waveform generation**, making the tool an excellent resource for studying analog concepts.

In sum, this project presents an accessible, low-cost multimeter that consolidates multiple measurement functions and signal generation capabilities into a single Arduino-based unit. Its design emphasizes affordability, functionality, and educational value, making it a valuable addition for students, hobbyists, and professionals who require a portable and comprehensive multimeter solution.

COMPONENTS REQUIRED

- Arduino Uno
- LCD Display (16x2), attached with I2C module
- Resistors (for voltage divider and current sensing)
- Thermocouple (for temperature measurement)
- Hall effect sensor (for current measurement)
- Capacitors (for filtering purposes)
- Probes for continuity testing
- Op-Amp (in DAC)
- Battery (9V)
- Breadboard and jumper wires
- 6-bit R-2R ladder for DAC

THEORY

Voltage Measurement

In this project, voltage measurement is achieved using a **voltage divider circuit**, a fundamental approach for reducing input voltage to a level safe for the Arduino's analog-to-digital converter (ADC). The voltage divider circuit consists of two resistors in series. The input voltage to be measured is applied across these resistors, and the voltage that falls across one of the resistors (connected to the Arduino's ADC pin) represents a fraction of the input voltage.

Here's how it works: when a high input voltage (higher than the Arduino's ADC maximum of 5V) needs to be measured, the voltage divider scales it down to a range that the ADC can safely process. By selecting appropriate resistor values, the output voltage from the divider is a known fraction of the input, falling within the ADC's range of 0-5V.

Once the Arduino reads this scaled voltage via the ADC, it applies a mathematical conversion to calculate the actual input voltage. If V_{in} is the input voltage, R_1 and R_2 are the resistances in the voltage divider, and V_{out} is the scaled voltage reaching the ADC, then:

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

To find V_{in} , the Arduino uses the rearranged formula:

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

After measuring V_{out} with the ADC, the Arduino multiplies it by the appropriate scaling factor to yield V_{in} , the original voltage. This approach enables the Arduino-based multimeter to measure voltages significantly higher than its nominal ADC limit, while ensuring accurate and safe voltage readings across various applications. To stabilise the values, EMA (Exponential Moving Average) is used.

Current Measurement

In this project, a Hall effect sensor is employed to **measure current non-invasively**, allowing current detection without needing to physically interrupt the circuit. This method is especially useful in applications where circuit continuity must be maintained or where traditional current measurement methods are impractical.

The principle behind the **Hall effect sensor** is that when electric current flows through a conductor, it creates a magnetic field surrounding the conductor. The strength of this magnetic field is directly proportional to the current. The sensor is designed to detect this magnetic field by producing a voltage that varies in response to the magnetic field's strength.

The sensor typically consists of a thin strip of semiconductor material, and when it is placed near the current-carrying conductor, the magnetic field exerts a force on the charge carriers within the sensor. This force causes a displacement of charges to one side of the semiconductor, resulting in a measurable potential difference, or "Hall voltage." The Hall

voltage is proportional to the magnetic field strength, which, in turn, is proportional to the current flowing through the conductor.

This analog voltage output from the Hall sensor is then fed into one of the Arduino's ADC (Analog-to-Digital Converter) pins. The Arduino reads this voltage and, using calibration data, translates it into a corresponding current value. Calibration involves mapping the Hall effect sensor's output to known current values, allowing the Arduino to accurately interpret any measured voltage as a specific current level.

The advantages of using a Hall effect sensor for current measurement include:

- **Electrical Isolation:** There is no direct electrical connection between the sensor and the conductor, allowing for safer current measurements, especially with high currents.
- **Bidirectional Current Sensing:** Many Hall effect sensors can detect both the direction and magnitude of the current, providing information about current flow direction.
- **Low Power Consumption:** Hall effect sensors generally consume little power and can be used in continuous monitoring applications.

By leveraging the Hall effect sensor, this Arduino-based multimeter can perform reliable and precise current measurements while maintaining the integrity and safety of the circuit. This setup expands the functionality and application range of the device, making it suitable for both educational experiments and practical use cases where traditional ammeters may not be feasible.

Temperature Measurement

In this project, temperature measurement is accomplished using a **thermocouple**, which is a highly effective and widely used temperature sensor, particularly suited for extreme temperature ranges. A thermocouple operates on the **Seebeck effect**: when two different metals are joined at one end (the sensing junction) and kept at a different temperature from the other ends (reference junction), a small voltage is generated at the open ends, proportional to the temperature difference between the two junctions.

Working behind temperature measurements:

1. **Thermocouple Composition:** A thermocouple consists of two wires made from different metals, like nickel-chromium (Type K thermocouple). These two wires are joined at one end, forming the "hot" or "sensing" junction, which is placed at the location where the temperature is to be measured. The other ends of the wires (the "cold" or "reference" junction) remain at a known reference temperature.
2. **Voltage Generation:** The temperature difference between the sensing and reference junctions generates a small voltage, known as the thermoelectric voltage. This voltage is minute, typically in the microvolt or millivolt range, and varies according to the thermocouple type and the temperature gradient. For example, a Type K thermocouple produces approximately 41 microvolts per degree Celsius difference.
3. **Amplifying the Signal:** Because the thermocouple voltage is very small, an amplifier (often an instrumentation amplifier) is required to boost the signal to a level readable by the Arduino's ADC. Specialized thermocouple amplifier ICs, such as the MAX6675 or MAX31855, are commonly used with Arduino. These ICs also provide cold-junction

compensation, which automatically corrects the readings for temperature variations at the reference junction, ensuring accurate measurements.

4. **Reading the Temperature:** The amplified voltage output from the thermocouple is then read by the Arduino's ADC. The Arduino uses a calibration curve specific to the thermocouple type to convert the voltage into an accurate temperature reading. Thermocouple types (e.g., K, J, T) each have their own standardized voltage-temperature relationships (known as calibration curves), which are available in lookup tables or equations.
5. **Calibration and Temperature Conversion:** The Arduino applies the thermocouple's calibration equation to convert the amplified voltage reading into the temperature at the sensing junction. For instance, in a Type K thermocouple, the equation or table correlates voltage to temperature based on the thermoelectric characteristics of nickel-chromium and nickel-aluminium.

Advantages and Practical Application:

- **Wide Measurement Range:** Thermocouples can measure a vast temperature range, from cryogenic levels to over 1000°C, making them highly versatile.
- **Fast Response Time:** Due to their small size, thermocouples respond quickly to temperature changes, making them ideal for applications requiring rapid temperature readings.
- **Durability and Versatility:** Thermocouples are robust and can be used in a variety of environments, including industrial, scientific, and educational applications, where durability and high-temperature range are essential.

This approach allows the Arduino-based multimeter to measure temperature accurately and efficiently, further enhancing its utility as a comprehensive, multifunctional tool for both educational and practical applications.

Frequency measurement

Frequency is defined as the number of cycles a periodic signal completes in one second. The formula for frequency f is given by:

$$f = \frac{1}{T}$$

where:

- f = Frequency (in Hz)
- T = Period of the signal (in seconds)

To measure the frequency of an AC signal, we need to:

1. Measure the period T , which is the time taken for one complete cycle of the waveform.
2. Calculate the frequency using the formula above.

In digital systems, we typically count the number of rising edges (or falling edges) of the waveform over a given period. This count is then used to calculate the frequency.

Continuity testing

In this project, continuity testing is an essential function, allowing the Arduino-based multimeter to detect if a circuit is closed or broken. Continuity tests are typically used to verify connections in circuits, cables, or other electrical components, helping to quickly identify faults, breaks, or disconnected elements.

How Continuity Testing Works:

1. **Small Current Injection:** When the continuity test is initiated, the Arduino sends a small current through the circuit or component under test. This current is minimal and safe, ensuring that it won't damage any sensitive components.
2. **Resistance Threshold:** The Arduino measures the resistance between the two test points. If the resistance is below a predefined threshold (typically around 50 ohms or less), the circuit is considered continuous, meaning that there is an uninterrupted electrical path between the test points. If the resistance is above this threshold, it indicates an open circuit or a break.
3. **Practical Thresholds and Adjustments:** The continuity threshold can be adjusted in the Arduino's code to account for specific application needs. For instance, a stricter threshold may be used for testing high-precision circuits, while a more relaxed threshold could be suitable for general-purpose applications.

Benefits and Practical Application:

- **Fast Fault Detection:** Continuity testing is a quick way to identify breaks, bad connections, or defective components in circuits.
- **Convenience and Versatility:** The buzzer or LED indicator offers an immediate, easily interpretable result, making it simple for users to test multiple points in a circuit quickly.
- **Educational and Practical Value:** This function introduces users to concepts of circuit continuity and resistance, making it particularly useful in educational settings.

The continuity test is a straightforward but highly valuable addition to this Arduino-based multimeter, enhancing its usefulness for troubleshooting and quality assurance in both academic and practical contexts. By incorporating audible and visual indicators, the device provides immediate feedback, ensuring efficient and user-friendly operation.

6-bit DAC Using R-2R Ladder

This project features a 6-bit Digital-to-Analog Converter (DAC) based on an R-2R ladder network, a key addition that enables the Arduino-based multimeter to generate smooth analog signals from digital inputs. This capability enhances the device's functionality by allowing it to produce AC signals, enabling AC voltage measurement and testing.

How the 6-bit R-2R Ladder DAC Works:

1. **R-2R Ladder Design:** An R-2R ladder DAC consists of a specific resistor configuration, where resistors are arranged in a repeated sequence of values, either RRR or 2R2R2R. This setup creates a binary-weighted network that efficiently converts a digital binary input into a corresponding analog voltage output. The DAC's 6-bit design

means it can represent 64 discrete levels of voltage (from 0 to 63 in binary) between the reference voltage and ground.

2. **Digital Input to Analog Output:** The Arduino generates a 6-bit binary value, which is fed into the R-2R ladder. Each bit in this binary value represents a different voltage weight. The DAC combines the weighted contributions of each bit to produce a corresponding analog voltage. For instance, the highest bit contributes half the reference voltage, while the lowest bit contributes a fraction of this value, allowing for precise voltage steps in the analog output.
3. **Smooth, Scalable Analog Output:** The R-2R ladder DAC enables the Arduino to produce smooth, graduated voltage levels. As the binary input value changes, the output voltage adjusts smoothly across the range. This enables the multimeter to handle a variety of AC signals for measurement and testing. For example, by updating the binary input in a wave pattern (like a sine or triangle wave), the Arduino can generate AC waveforms, providing a source for testing the multimeter's AC measurement accuracy.
4. **Optional Amplification:** For applications requiring higher output voltages or greater signal strength, the analog output from the DAC can be further amplified using an operational amplifier (op-amp). The op-amp boosts the analog signal, allowing the multimeter to produce stronger, more stable AC signals when required. This amplification option adds flexibility, enabling the device to meet various testing needs.

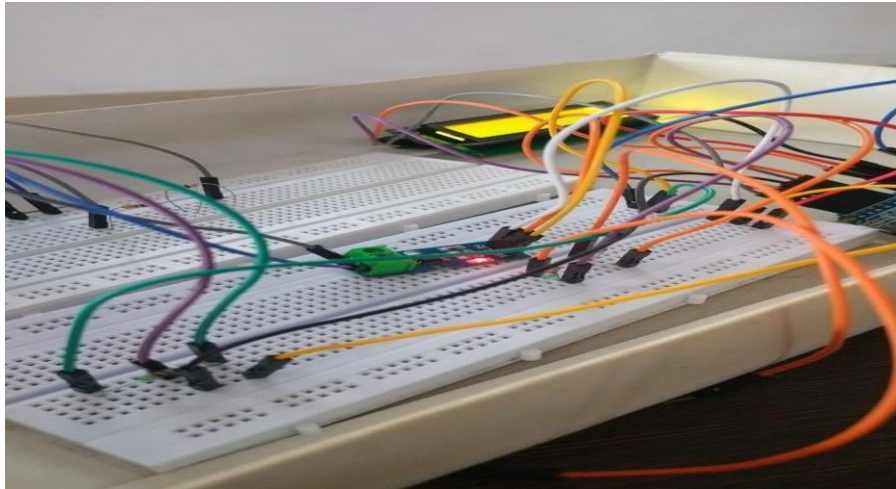
Advantages and Practical Applications:

- **Testing AC Measurement Capabilities:** The DAC can produce test signals that allow the multimeter to measure and validate AC voltage readings, making it a useful tool for calibration and troubleshooting.
- **Signal Generation for Educational Use:** The R-2R ladder DAC introduces users to digital-to-analog conversion principles, bridging digital systems and analog signal generation. This capability is valuable in educational environments where students can experiment with and visualize waveform generation.
- **Enhanced Flexibility and Accuracy:** The smooth, stable output provided by the R-2R ladder network enables precise control over analog signals, enhancing the versatility of the Arduino-based multimeter as a comprehensive testing instrument.

Incorporating a 6-bit R-2R DAC transforms this multimeter from a simple measurement tool into a multifunctional device capable of generating and analysing AC signals.

PROCEDURE

Hardware Connections are as per the code.





Analog Concepts Involved

This project makes use of several key analog concepts:

- **Voltage Divider:** Used to scale down voltages for safe measurement by the Arduino's ADC.
- **Hall Effect:** For non-invasive current measurement based on the magnetic field generated by current flow.
- **Seebeck Effect:** The basis for temperature measurement using a thermocouple.
- **Digital-to-Analog Conversion (DAC):** The 6-bit DAC converts digital signals from the Arduino into analog voltages for AC signal generation.

ARDUINO CODE

The Arduino code is responsible for reading analog values from the sensors, processing these values, and displaying the results on the LCD. For current and temperature, specific formulas are applied to convert sensor outputs into meaningful measurements. The code also handles user input through buttons for switching between measurement modes.

The codes attached:

- 1) Multimeter main code (behind the working)
- 2) Frequency Measurement Code
- 3) Additional: Triangle Wave generator
- 4) Additional: Square Wave generator
- 5) Additional: Sine Wave generator

Code for Multimeter (Voltage, Continuity, Resistance, Current, Temperature)

```
#include <Wire.h>

#include <LiquidCrystal_PCF8574.h>

#include "max6675.h" // Include MAX6675 library

LiquidCrystal_PCF8574 lcd(0x27); // Adjust your I2C address if necessary


// Resistance Measurement Variables

const int refResistor = 3300; // Reference resistor value (in ohms)

const int analogPinRes = A2; // Pin for resistance measurement

const float Vcc = 5.0; // Adjust this if your Arduino's 5V pin is not exactly 5V


// Current Measurement Variables

const int sensorPin = A4; // Analog pin for ACS712

const float sensitivity = 0.185; // Sensitivity for ACS712-5A

float VREF = 5;

const int numReadings = 7;


// Voltage Measurement Variables

const int voltagePin = A0; // Pin for voltage measurement

const int continuityPin = A2; // Pin for continuity check

const int continuityThreshold = 750;

const int voltageReadingsCount = 10;

float continuityEMA = 0.0;

float voltageEMA = 0.0;

const float continuityAlpha = 0.1;

const float voltageAlpha = 0.1;


// Temperature Measurement Variables

int soPin = 12; // SO=Serial Out

int csPin = 10; // CS = Chip Select

int sckPin = 13; // SCK = Serial Clock

MAX6675 temperatureSensor(sckPin, csPin, soPin); // Create instance object for MAX6675


// User Input Variable

int mode = 0; // 1 for Resistance, 2 for Current, 3 for Voltage, 4 for Temperature
```

```

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

    lcd.begin(16, 2);

    lcd.setBacklight(1);

    lcd.clear();

    lcd.print("Select Mode:");

    lcd.setCursor(0, 1);

    lcd.print("1:R 2:I 3:V 4:T");
}

void loop() {
    if (Serial.available()) {
        char input = Serial.read();

        if (input == '1') mode = 1; // Resistance Mode
        if (input == '2') mode = 2; // Current Mode
        if (input == '3') mode = 3; // Voltage Mode
        if (input == '4') mode = 4; // Temperature Mode

        lcd.clear();
    }

    switch (mode) {
        case 1:
            measureResistance();

            break;

        case 2:
            measureCurrent();

            break;

        case 3:
            measureVoltage();

            break;

        case 4:
            measureTemperature();

            break;

        default:

```

```

        lcd.setCursor(0, 0);

        lcd.print("Select Mode:");

        lcd.setCursor(0, 1);

        lcd.print("1:R 2:I 3:V 4:T");

        break;
    }

    delay(1000); // Refresh rate
}

// Function to measure resistance
void measureResistance() {
    int sensorValue = analogRead(analogPinRes);

    float voltage = sensorValue * (Vcc / 1023.0);

    float unknownResistance = 0;

    lcd.setCursor(0, 0);

    lcd.print("Resistance:  ");

    if (voltage >= Vcc - 0.1) {

        lcd.setCursor(0, 1);

        lcd.print("Out of Range  ");

    } else if (voltage <= 0.1) {

        lcd.setCursor(0, 1);

        lcd.print("Short Circuit  ");

    } else {

        unknownResistance = (voltage * refResistor) / (Vcc - voltage);

        lcd.setCursor(0, 1);

        lcd.print(unknownResistance);

        Serial.println(unknownResistance);

        lcd.print(" ohms    ");

    }
}

// Function to measure current
float getMedianReading() {

```

```

int readings[numReadings];

for (int i = 0; i < numReadings; i++) {
    readings[i] = analogRead(sensorPin);

    delay(10);
}

for (int i = 0; i < numReadings - 1; i++) {
    for (int j = i + 1; j < numReadings; j++) {
        if (readings[i] > readings[j]) {
            int temp = readings[i];
            readings[i] = readings[j];
            readings[j] = temp;
        }
    }
}

return readings[numReadings / 2];
}

```

```

void measureCurrent() {
    float sensorValue = getMedianReading();
    float voltage = sensorValue * (5.0 / 1023.0);
    float current = (VREF - voltage);
}

```

```

lcd.setCursor(0, 0);
lcd.print("Current:   ");
lcd.setCursor(0, 1);
lcd.print(current);
Serial.println(current);
lcd.print(" A      ");
}

```

// Function to measure voltage

```

void measureVoltage() {
    int continuityValue = analogRead(continuityPin);
    continuityEMA = continuityAlpha * continuityValue + (1.0 - continuityAlpha) * continuityEMA;
    bool isContinuous = (continuityEMA < continuityThreshold);
}

```

```

    lcd.setCursor(0, 0);

    if (isContinuous) {

        lcd.print("Continuity: YES");

        float voltageSum = 0.0;

        for (int i = 0; i < voltageReadingsCount; i++) {

            int sensorValue = analogRead(voltagePin);

            float currentVout = sensorValue * (5.0 / 1023.0) * (3333.0 / 33.0);

            voltageEMA = voltageAlpha * currentVout + (1.0 - voltageAlpha) * voltageEMA;

            voltageSum += currentVout;

            delay(10);

        }

        float avgVoltage = voltageSum / 10.0;

        lcd.setCursor(0, 1);

        lcd.print("Voltage: ");

        lcd.print(avgVoltage, 2);

        lcd.print(" V ");

    } else {

        lcd.setCursor(0, 1);

        lcd.print("Voltage: NA  ");

    }

}

// Function to measure temperature

void measureTemperature() {

    float celsius = temperatureSensor.readCelsius();

    float fahrenheit = temperatureSensor.readFahrenheit();

    lcd.setCursor(0, 0);

    lcd.print("Temp: ");

    lcd.print(celsius);

    lcd.print(" C ");

    lcd.setCursor(0, 1);

    lcd.print(fahrenheit);

```



```

    lcd.print(" F ");

    Serial.print("Temperature: ");

    Serial.print(celsius);

    Serial.print(" °C, ");

    Serial.print(fahrenheit);

    Serial.println(" °F");
}

```

Code for Frequency Measurement

```

#define DAC_PIN_0 2

#define DAC_PIN_1 3

#define DAC_PIN_2 4

#define DAC_PIN_3 5

#define DAC_PIN_4 6

#define DAC_PIN_5 7


#define ANALOG_FEEDBACK_PIN A0

#define NUM_SAMPLES 64


volatile int index = 0;

bool in_error_state = false;

unsigned long error_start_time = 0;


float alpha = 0.1; // Smoothing factor for exponential smoothing
float smoothed_feedback = 0;


// Debounce settings

int error_debounce_counter = 0;

const int debounce_threshold = 5;


// Error thresholds

const int error_threshold_low = 50;

const int error_threshold_high = 950;

```

```

// Triangular wave lookup table
const uint8_t triangular_wave[NUM_SAMPLES] = {
    0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60,
    63, 59, 55, 51, 47, 43, 39, 35, 31, 27, 23, 19, 15, 11, 7, 3,
    0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60,
    63, 59, 55, 51, 47, 43, 39, 35, 31, 27, 23, 19, 15, 11, 7, 3
};

volatile unsigned long last_time = 0;
volatile unsigned long current_time = 0;
volatile float frequency = 0.0;

float amplitude = 0.0;
int max_value = 0;
int min_value = 1023;

void setup() {
    DDRD |= 0b11111100; // Set DAC pins as outputs
    Serial.begin(9600);

    // Faster analogRead setup for improved performance
    ADCSRA = (ADCSRA & 0xF8) | 0x04; // Set ADC prescaler to 16

    // Timer1 setup for waveform generation
    noInterrupts();
    TCCR1A = 0;
    TCCR1B = 0;
    TCNT1 = 0;
    OCR1A = 249; // Generates ~1kHz triangular wave
    TCCR1B |= (1 << WGM12); // CTC mode
    TCCR1B |= (1 << CS11) | (1 << CS10); // Prescaler 64
    TIMSK1 |= (1 << OCIE1A);
    interrupts();

```

```

// Attach interrupt to detect rising edge for frequency calculation
attachInterrupt(digitalPinToInterrupt(DAC_PIN_0), calculateFrequency, RISING);
}

ISR(TIMER1_COMPA_vect) {
    generateWaveform();
}

void generateWaveform() {
    uint8_t value = triangular_wave[index++];
    PORTD = (PORTD & 0x03) | (value << 2);
    if (index >= NUM_SAMPLES) index = 0;
}

void calculateFrequency() {
    current_time = micros(); // Get current time in microseconds
    if (last_time > 0) {
        unsigned long time_diff = current_time - last_time; // Time between rising edges
        frequency = 1000000.0 / time_diff; // Frequency in Hz (1 million microseconds = 1 second)
    }
    last_time = current_time; // Update last time for next interrupt
}

void processFeedback() {
    int currentReading = analogRead(ANALOG_FEEDBACK_PIN);

    // Exponential smoothing for noise reduction
    smoothed_feedback = alpha * currentReading + (1 - alpha) * smoothed_feedback;

    // Track min and max values for amplitude calculation
    if (currentReading > max_value) max_value = currentReading;
    if (currentReading < min_value) min_value = currentReading;

    // Calculate peak-to-peak amplitude
    amplitude = max_value - min_value;
}

```

```

// Reset max and min values periodically for next calculation
if (index == 0) {
    max_value = 0;
    min_value = 1023;
}

// Debugging output for monitoring feedback
Serial.print("Smoothed Feedback: ");
Serial.println(smoothed_feedback);

// Error checking with debounce logic
if (smoothed_feedback < error_threshold_low || smoothed_feedback > error_threshold_high) {
    error_debounce_counter++;
    Serial.print("Debounce Counter: ");
    Serial.println(error_debounce_counter);

    if (error_debounce_counter >= debounce_threshold) {
        handleError();
    }
} else {
    error_debounce_counter = 0; // Reset debounce counter if feedback is within range
}
}

void handleError() {
    if (!in_error_state) {
        Serial.println("Error detected! System in error state, attempting recovery...");
        in_error_state = true;
        error_start_time = millis();
    }
}

void attemptRecovery() {
    if (in_error_state && (millis() - error_start_time > 2000)) { // 2-second recovery window

```

```

    Serial.println("Attempting to recover...");

    error_debounce_counter = 0;

    in_error_state = false;

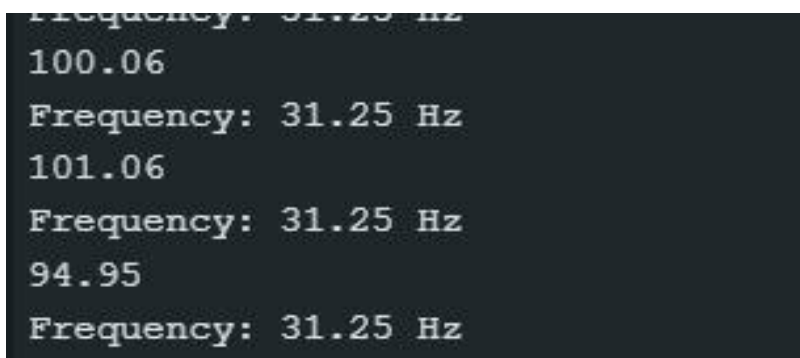
    Serial.println("Recovery successful, system back to normal.");
}

}

void loop() {
    if (!in_error_state) {
        processFeedback();
    } else {
        attemptRecovery();
    }

    // Output the frequency and amplitude periodically
    Serial.print("Frequency: ");
    Serial.print(frequency);
    Serial.print(" Hz, Amplitude: ");
    Serial.print(amplitude);
    Serial.println(" units");
    delay(1000); // Update every second
}

```



The screenshot shows a serial monitor with a dark background and light green text. It displays a repeating pattern of frequency and amplitude values. The frequency is consistently 31.25 Hz, and the amplitude values are 100.06, 101.06, and 94.95, appearing in a staggered fashion.

```

Frequency: 31.25 Hz
100.06
Frequency: 31.25 Hz
101.06
Frequency: 31.25 Hz
94.95
Frequency: 31.25 Hz

```

Code for Triangle Wave Generation

```
#define DAC_PIN_0 2
#define DAC_PIN_1 3
#define DAC_PIN_2 4
#define DAC_PIN_3 5
#define DAC_PIN_4 6
#define DAC_PIN_5 7
#define ANALOG_FEEDBACK_PIN A0
#define NUM_SAMPLES 64
#define BUFFER_SIZE 10

volatile int index = 0;

bool error_detected = false;
bool in_error_state = false;
unsigned long error_start_time = 0;

float alpha = 0.1; // Smoothing factor for exponential smoothing
float smoothed_feedback = 0;

// Debounce settings
int error_debounce_counter = 0;
const int debounce_threshold = 5; // Number of consecutive errors needed to trigger error state

// Error thresholds
const int error_threshold_low = 50;
const int error_threshold_high = 950;

// Triangular wave lookup table
const uint8_t triangular_wave[NUM_SAMPLES] = {
    0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60,
    63, 59, 55, 51, 47, 43, 39, 35, 31, 27, 23, 19, 15, 11, 7, 3,
    0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60,
    63, 59, 55, 51, 47, 43, 39, 35, 31, 27, 23, 19, 15, 11, 7, 3
};
```

```

void setup() {

  DDRD |= 0b11111100; // Set DAC pins as outputs

  Serial.begin(9600);

  // Faster analogRead setup

  ADCSRA = (ADCSRA & 0xF8) | 0x04; // Set ADC prescaler to 16

  // Timer setup for waveform generation

  noInterrupts();

  TCCR1A = 0;

  TCCR1B = 0;

  TCNT1 = 0;

  OCR1A = 249;

  TCCR1B |= (1 << WGM12); // CTC mode

  TCCR1B |= (1 << CS11) | (1 << CS10); // Prescaler 64

  TIMSK1 |= (1 << OCIE1A);

  interrupts();

}

ISR(TIMER1_COMPA_vect) {

  generateWaveform();

}

void generateWaveform() {

  uint8_t value = triangular_wave[index++];

  PORTD = (PORTD & 0x03) | (value << 2);

  if (index >= NUM_SAMPLES) index = 0;

}

void processFeedback() {

  int currentReading = analogRead(ANALOG_FEEDBACK_PIN);

  // Exponential smoothing for noise reduction

  smoothed_feedback = alpha * currentReading + (1 - alpha) * smoothed_feedback;

  Serial.println(smoothed_feedback);

```

```

// Error checking with debounce logic

if (smoothed_feedback < error_threshold_low || smoothed_feedback > error_threshold_high) {
    error_debounce_counter++;

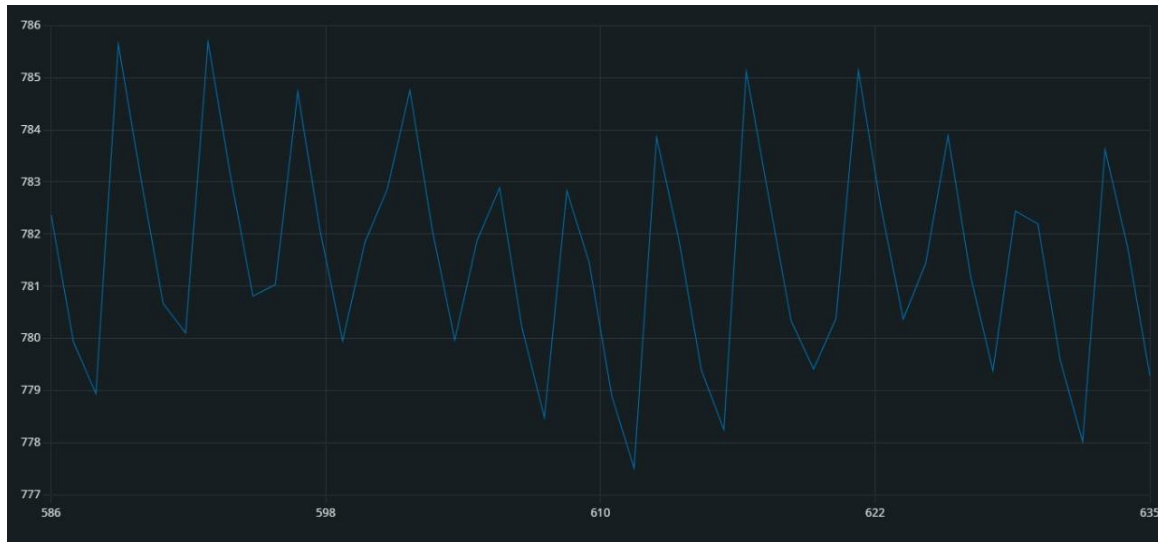
    if (error_debounce_counter > debounce_threshold) {
        handleError();
    }
} else {
    error_debounce_counter = 0; // Reset debounce counter if within range
}
}

void handleError() {
    if (!in_error_state) {
        Serial.println("System in error state, attempting recovery...");
        in_error_state = true;
        error_start_time = millis();
    }
}

void attemptRecovery() {
    if (in_error_state && (millis() - error_start_time > 2000)) { // 2-second recovery window
        Serial.println("Attempting to recover...");
        error_debounce_counter = 0;
        in_error_state = false;
    }
}

void loop() {
    if (!in_error_state) {
        processFeedback();
    } else {
        attemptRecovery();
    }
}

```

Code for Square Wave Generation

```
// DAC Output Pins

#define DAC_PIN_0 2

#define DAC_PIN_1 3

#define DAC_PIN_2 4

#define DAC_PIN_3 5

#define DAC_PIN_4 6

#define DAC_PIN_5 7

// Analog Feedback Pin

#define ANALOG_FEEDBACK_PIN A0


// Waveform Settings

constexpr int SQUARE_WAVE_FREQ = 1000; // Frequency in Hz


// Error Handling Thresholds

constexpr int ERROR_THRESHOLD_LOW = 30; // Increase/decrease as needed

constexpr int ERROR_THRESHOLD_HIGH = 970; // Increase/decrease as needed

constexpr int DEBOUNCE_THRESHOLD = 10; // Increase this value to reduce false errors

constexpr int RECOVERY_TIME_MS = 2000; // 2 seconds recovery window


// Smoothing Factor for Exponential Smoothing

constexpr float SMOOTHING_FACTOR = 0.2;


// Global Variables
```

```

volatile bool square_wave_state = false;

volatile int error_debounce_counter = 0;

volatile bool in_error_state = false;

float smoothed_feedback = 0;

unsigned long error_start_time = 0;


// Function Prototypes

void setupTimer();

void generateSquareWave();

void processFeedback();

void handleError();

void attemptRecovery();


void setup() {

    // Set DAC pins as outputs

    DDRD |= 0b11111100;

    Serial.begin(9600);

    // Set ADC prescaler to 16 for faster analog reads

    ADCSRA = (ADCSRA & 0xF8) | 0x04;


    // Setup Timer for square wave generation

    setupTimer();

}

// Timer1 Setup for Square Wave Generation

void setupTimer() {

    noInterrupts();

    TCCR1A = 0;

    TCCR1B = 0;

    TCNT1 = 0;

    OCR1A = (F_CPU / (2 * SQUARE_WAVE_FREQ * 64)) - 1;

    TCCR1B |= (1 << WGM12);

    TCCR1B |= (1 << CS11) | (1 << CS10);

    TIMSK1 |= (1 << OCIE1A);

    interrupts();

}

```

```

// Interrupt Service Routine for Square Wave Generation
ISR(TIMER1_COMPA_vect) {

    generateSquareWave();

}

// Function to Generate Square Wave
void generateSquareWave() {

    square_wave_state = !square_wave_state;

    if (square_wave_state) {

        PORTD |= 0b11111100;

    } else {

        PORTD &= ~0b11111100;

    }

}

// Function to Read Feedback and Perform Error Detection
void processFeedback() {

    int currentReading = analogRead(ANALOG_FEEDBACK_PIN);

    // Apply Exponential Smoothing
    smoothed_feedback = SMOOTHING_FACTOR * currentReading + (1 - SMOOTHING_FACTOR) * smoothed_feedback;

    // Apply Moving Average Filter (optional)
    static float moving_average = 0;
    moving_average = (moving_average * 0.9) + (smoothed_feedback * 0.1);

    //Serial.print("Smoothed Feedback: ");
    Serial.println(smoothed_feedback);

    // Error Detection with Debounce Logic
    if (moving_average < ERROR_THRESHOLD_LOW || moving_average > ERROR_THRESHOLD_HIGH) {

        error_debounce_counter++;

        if (error_debounce_counter >= DEBOUNCE_THRESHOLD) {

            handleError();

        }

    }
}

```

```

    } else {

        error_debounce_counter = 0;

    }
}

// Function to Handle Error State
void handleError() {

    if (!in_error_state) {

        Serial.println("Error detected! Entering error state...");

        in_error_state = true;

        error_start_time = millis();

    }

}

// Function to Attempt System Recovery
void attemptRecovery() {

    if (millis() - error_start_time > RECOVERY_TIME_MS) {

        Serial.println("Attempting recovery...");

        error_debounce_counter = 0;

        in_error_state = false;

    }

}

void loop() {

    if (!in_error_state) {

        processFeedback();

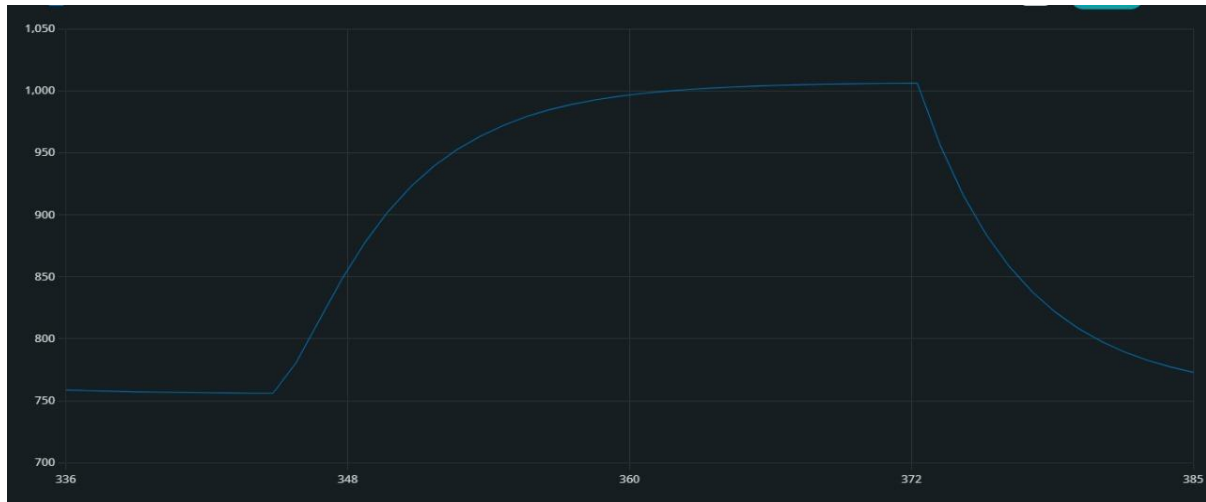
    } else {

        attemptRecovery();

    }

}

```



Code for Sine Wave Generation

```
#define DAC_PIN_0 2

#define DAC_PIN_1 3

#define DAC_PIN_2 4

#define DAC_PIN_3 5

#define DAC_PIN_4 6

#define DAC_PIN_5 7

#define ANALOG_FEEDBACK_PIN A0

#define NUM_SAMPLES 64

#define BUFFER_SIZE 10

volatile int index = 0;

bool error_detected = false;

bool in_error_state = false;

unsigned long error_start_time = 0;

float alpha = 0.1; // Smoothing factor for exponential smoothing

float smoothed_feedback = 0;

// Debounce settings

int error_debounce_counter = 0;

const int debounce_threshold = 5; // Number of consecutive errors needed to trigger error state

// Error thresholds

const int error_threshold_low = 50;

const int error_threshold_high = 950;
```

```

// Triangular wave lookup table

const uint8_t triangular_wave[NUM_SAMPLES] = {

    0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60,

    63, 59, 55, 51, 47, 43, 39, 35, 31, 27, 23, 19, 15, 11, 7, 3,

    0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60,

    63, 59, 55, 51, 47, 43, 39, 35, 31, 27, 23, 19, 15, 11, 7, 3

};\

void setup() {

    DDRD |= 0b11111100; // Set DAC pins as outputs

    Serial.begin(9600);

    // Faster analogRead setup

    ADCSRA = (ADCSRA & 0xF8) | 0x04; // Set ADC prescaler to 16


// Timer setup for waveform generation

    noInterrupts();

    TCCR1A = 0;

    TCCR1B = 0;

    TCNT1 = 0;

    OCR1A = 249;

    TCCR1B |= (1 << WGM12); // CTC mode

    TCCR1B |= (1 << CS11) | (1 << CS10); // Prescaler 64

    TIMSK1 |= (1 << OCIE1A);

    interrupts();


ISR(TIMER1_COMPA_vect) {

    generateWaveform();

}

void generateWaveform() {

    uint8_t value = triangular_wave[index++];

    PORTD = (PORTD & 0x03) | (value << 2);

    if (index >= NUM_SAMPLES) index = 0;

}

```

```

void processFeedback() {
    int currentReading = analogRead(ANALOG_FEEDBACK_PIN);

    // Exponential smoothing for noise reduction
    smoothed_feedback = alpha * currentReading + (1 - alpha) * smoothed_feedback;

    Serial.println(smoothed_feedback);

    // Error checking with debounce logic
    if (smoothed_feedback < error_threshold_low || smoothed_feedback > error_threshold_high) {
        error_debounce_counter++;

        if (error_debounce_counter > debounce_threshold) {
            handleError();
        }
    } else {
        error_debounce_counter = 0; // Reset debounce counter if within range
    }
}

void handleError() {
    if (!in_error_state) {
        Serial.println("System in error state, attempting recovery...");

        in_error_state = true;
        error_start_time = millis();
    }
}

void attemptRecovery() {
    if (in_error_state && (millis() - error_start_time > 2000)) { // 2-second recovery window
        Serial.println("Attempting to recover...");

        error_debounce_counter = 0;
        in_error_state = false;
    }
}

```

```

}

}

void loop() {

  if (!lin_error_state) {

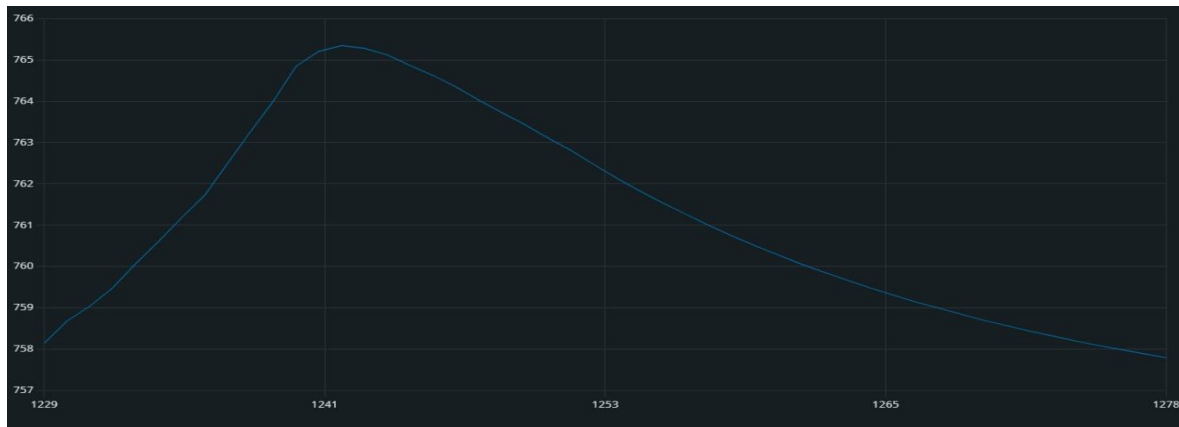
    processFeedback();

  } else {

    attemptRecovery();

  }
}

```



GAINS FROM THE PROJECT

By completing this project, we not only developed a functional and versatile Arduino-based multimeter but also gained a deeper understanding of various key concepts in electronics, both on the hardware and software sides. This hands-on experience was invaluable, as it enabled us to bridge theoretical knowledge with practical application. Given below is a more detailed breakdown of the specific learning outcomes and skills gained through this project.

1. Hands-on Experience with Arduino Programming

- **Programming Skills:** The project required extensive use of Arduino's C/C++-based programming environment to control hardware components like sensors, ADCs, DACs, and output indicators. We gained proficiency in writing code to manage data acquisition, signal processing, and interfacing with external modules.
- **Sensor Integration:** Interfacing sensors such as the Hall effect sensor (for current measurement) and thermocouple (for temperature measurement) provided experience in reading analog signals, converting them to digital format using the Arduino's ADC, and then processing these readings to display meaningful results.
- **Data Handling:** We learned how to handle sensor data, perform necessary conversions (e.g., voltage to temperature), and apply calibration factors to ensure accurate readings. This process helped develop our understanding of data flow and manipulation within embedded systems.

2. Sensor Interfacing and Signal Conditioning

- **Analog-to-Digital Conversion (ADC):** Interfacing with various sensors like the voltage divider and Hall effect sensor involved reading analog signals through the Arduino's ADC. We learned the limitations of the Arduino's ADC resolution and how to work around them using techniques such as signal amplification and calibration.

- **Amplification and Filtering:** To accurately measure small voltages, such as those from the Hall effect sensor or thermocouple, we used operational amplifiers to condition the signals. This experience deepened our understanding of how to design circuits that improve signal quality before they are fed into the Arduino for processing.
- **Precision and Accuracy:** We explored methods to enhance the precision of measurements by calibrating sensors and adjusting the software to account for environmental factors. This taught us how to implement practical solutions to real-world problems that might affect measurement accuracy.

3. Understanding and Implementing Analog Concepts

- **Voltage Divider and RC Circuits:** We used voltage dividers for voltage measurement and the RC time constant method for capacitance measurement. This project reinforced our understanding of key analog concepts like Ohm's Law, voltage division, and the behavior of capacitors in time-dependent circuits.
- **DAC Design:** One of the more complex aspects of the project was designing and implementing a 6-bit DAC using an R-2R ladder network. Through this, we learned how digital signals can be converted to analog voltage levels, allowing the Arduino to output smooth waveforms that simulate AC signals. This required a strong understanding of digital-analog conversion principles and the role of DACs in modern electronics.
- **Practical Applications of Operational Amplifiers:** We applied operational amplifiers in various scenarios—both for signal conditioning (e.g., amplifying the thermocouple signal) and for driving higher currents through the DAC output. This helped us appreciate the role of op-amps in analog circuit design, signal amplification, and noise reduction.

4. Designing and Implementing Practical Circuits for Measurement

- **Circuit Design:** Designing and testing the various circuits for voltage, current, capacitance, temperature, and continuity measurement provided hands-on experience with real-world circuit implementation. We applied concepts such as passive component selection, resistor networks, and feedback in op-amps to create circuits that were both functional and accurate.
- **Component Selection and Troubleshooting:** We learned how to select appropriate components (e.g., resistors, capacitors, sensors) based on circuit requirements, and we gained experience in troubleshooting when circuits didn't work as expected. This required iterative testing, debugging, and making adjustments to both hardware and software.
- **Power Management:** Understanding the power requirements of the system, particularly the sensors and the DAC, helped us design circuits that could work efficiently within the constraints of the Arduino's 5V power supply. We also had to consider current consumption and how to minimize power usage in our multimeter design.

5. Practical Circuit Testing and Calibration

- **Testing Circuit Functionality:** After building and designing the circuits for each measurement function (voltage, current, capacitance, temperature, continuity), we performed real-world tests to validate each circuit's functionality. This included comparing the Arduino's readings with those from a commercial multimeter to assess accuracy.
- **Calibration Procedures:** For example, to ensure the Hall effect sensor provided accurate current readings, we performed a calibration using known reference currents and adjusted the software accordingly. Similarly, we calibrated the thermocouple by comparing its readings against a calibrated thermometer.

6. Understanding Multimeter Design

- **Multimeter Features:** This project gave us the opportunity to design a multimeter from the ground up. We learned how to combine various measurement functions into a single, cohesive tool, ensuring that the Arduino-based multimeter could accurately perform different types of measurements and present the results in a practical and user-friendly way.
- **User Interface and Output:** We worked on the user interface design, such as displaying results on an LCD screen and providing feedback through LEDs or buzzers for functions like continuity. This helped us learn how to make electronic devices more intuitive and accessible to users.

CONCLUSION

- This project gave us a comprehensive understanding of both hardware and software aspects of electronics. We gained practical knowledge of Arduino programming, sensor interfacing, signal processing, and analog circuit design. Additionally, we developed a deeper appreciation for measurement techniques and the importance of accuracy in real-world applications. By applying fundamental analog and digital concepts in the design of a functional multimeter, we were able to integrate theoretical learning with hands-on experience, which will be invaluable in both future academic pursuits and professional work in electronics and embedded systems.

FUTURE SCOPE

- This multimeter can be further improved by adding features such as automatic range selection, wireless communication for data logging, and higher accuracy with more precise sensors. The 6-bit DAC could be upgraded to an 8-bit or higher resolution for more accurate AC signal generation. Moreover, integration with software tools like MATLAB or Python for advanced data processing could provide additional functionality.

REFERENCES

- GitHub link for code, images, presentation slides: <https://github.com/gautham-here/digitalmult>