Contents

1 Sensor Description and Theory of Operation 2

1.1 Rogowski Coil–Based Current Sensor 2

1.2 Operating Principle and Induced EMF 3

1.3 Meaning of Mutual Inductance $\boldsymbol{M}$ 3

1.4 Differentiating Nature of the Rogowski Coil 4

1.5 Integrator Circuit for Current Reconstruction 5

1.6 Signal Conditioning and Amplification 5

1.7 Calibration and Practical Considerations 6

2 System Architecture. 7

2.1 Components: Hardware Blocks 7

3 Bare-Metal Firmware Architecture 9

3.1 Firmware Structure 10

3.2 Peripheral Initialization (Bare-Metal) 10

3.2.1 ADC Initialization (Single-Ended, AVcc Reference) 10

3.2.2 Timer1 Initialization (CTC Mode, 1 kHz Interrupt) 10

3.2.3 I^2C (TWI) Initialization for SSD1306 11

4 Digital Processing and code explanation. 11

4.1.1 ADC sampling 11

4.1.2 FIR filter and Fixed-point arithmetic 12

4.1.3 Frequency Estimation 14

4.1.4 Integration 15

4.1.5 Calculation of Current 15

4.1.6 Display 16

4.1.7 SPI interface 16

4.2 ESP8266 as SPI Slave with Wi-Fi Connectivity and MQTT 17

4.2.1 Json Format for publishing values 18

5 Edge Processing and Visualization 18

6 Project Prototype 19

7 Power uses. 20

8 Challenges 22

Project Repository: https://github.com/pointertoaperson/current_sensor

1 Sensor Description and Theory of Operation

1.1 Rogowski Coil–Based Current Sensor

The current sensing subsystem is based on a *Rogowski coil*, which is an air-cored, non-invasive sensor used for measuring alternating or rapidly changing currents. Unlike conventional current transformers, the Rogowski coil does not use a ferromagnetic core, which eliminates magnetic saturation and allows a wide measurement bandwidth. This makes it suitable for applications in power electronics, industrial monitoring, and frequency measurement.

The Rogowski coil is placed around a current-carrying conductor without requiring electrical contact, providing galvanic isolation between the measured system and the sensing electronics.

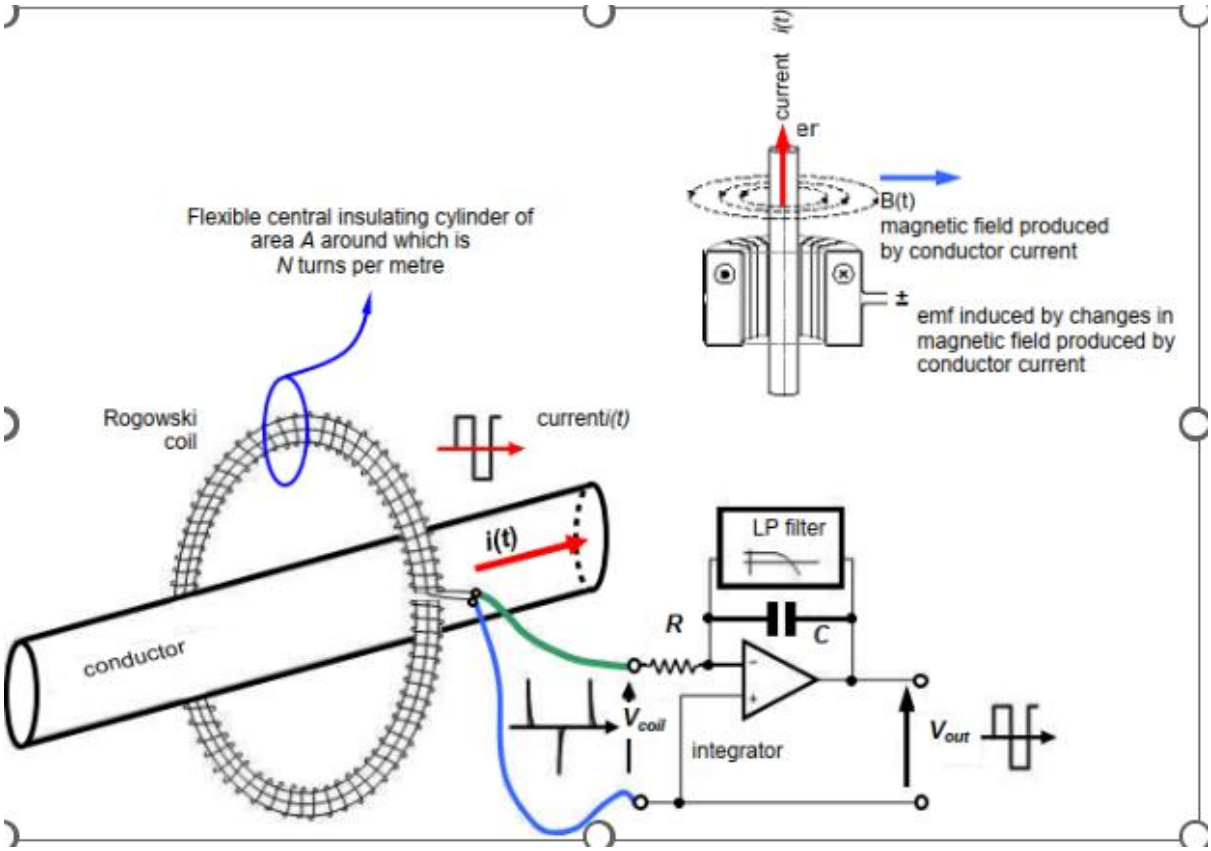




Fig: Hand-wound Rogowoski coil

1.2 Operating Principle and Induced EMF

When an alternating current flows through a conductor enclosed by the Rogowski coil, it generates a time-varying magnetic field. According to *Faraday's law of electromagnetic induction*, this changing magnetic field induces an electromotive force (EMF) in the coil.

The induced voltage at the coil terminals is given by:

$$v_{coil}(t) = -M\frac{di(t)}{dt}$$

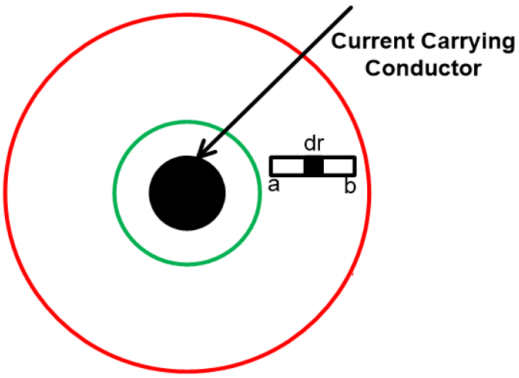
where:

- $v_{coil}(t)$is the induced voltage (EMF) of the Rogowski coil
- $i(t)$is the current flowing through the conductor
- $\frac{di(t)}{dt}$represents the rate of change of the current
- $M$is the *mutual inductance* between the conductor and the coil

The negative sign indicates polarity according to Lenz's law and depends on the orientation of the coil.

1.3 Meaning of Mutual Inductance $\boldsymbol{M}$

$$M = N \times \frac{\mu \times h}{2\pi} \times \ln\frac{a}{b}$$



Rogowski Coil

The mutual inductance Mdescribes how effectively the magnetic field generated by the conductor couples into the Rogowski coil. Its value depends on factors such as:

- Number of turns in the coil
- Coil geometry and enclosed area
- Distance between the conductor and the coil
- Uniformity of the winding

Because the Rogowski coil is air-cored and hand-wound, the value of $M$is relatively small. As a result, the induced voltage is typically low, especially for low-frequency or low-amplitude currents.

1.4 Differentiating Nature of the Rogowski Coil

From the induced EMF equation, it can be observed that the Rogowski coil output is proportional to the time derivative of the current rather than the current itself:

$$v_{coil}(t) \propto \frac{di(t)}{dt}$$

This means:

- Higher-frequency currents produce larger coil voltages
- Pure DC current produces no output
- The coil inherently behaves as a current differentiator

To obtain a signal proportional to the actual current waveform, additional signal processing is required.

1.5 Integrator Circuit for Current Reconstruction

To recover a voltage proportional to the current, the Rogowski coil output is fed into an active integrator circuit implemented using an operational amplifier.

The integrator output voltage is given by:

$$v_{out}(t) = \frac{1}{R_{in}C_f}\int v_{coil}(t)\,dt$$

Substituting the coil equation yields:

$$v_{out}(t) = \frac{M}{R_{in}C_f}\cdot i(t)$$

Thus, after integration, the output voltage becomes directly proportional to the measured current.

Here:

- R_{in} is the integrator input resistor
- C_f is the feedback capacitor
- $R_{in}C_f$ defines the integrator time constant and frequency response

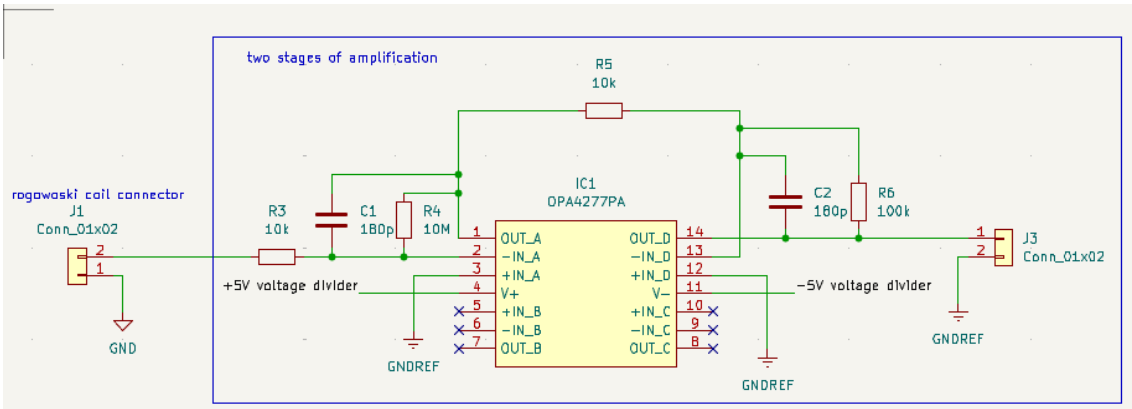
A practical integrator configuration is used to avoid DC saturation and offset drift.

1.6 Signal Conditioning and Amplification

Due to the small mutual inductance of the Rogowski coil, the integrated signal amplitude is relatively low. Therefore, the sensor front-end includes:

- An additional amplification stage to increase signal magnitude
- Voltage level shifting to center the AC waveform around mid-supply (2.5 V)
- Filtering and decoupling components to reduce noise

This ensures that the output signal fits within the input range of the microcontroller's ADC (0–5 V) and can be sampled accurately.



The integrator amplifier is constructed using OPA4277PA opamp with the time constant of $RC = 10K \times 180pF = 1.8\mu S$ with bleed resistor of 10Megaohm for higher bandwidth. A 100 times gain stage was cascaded after this.

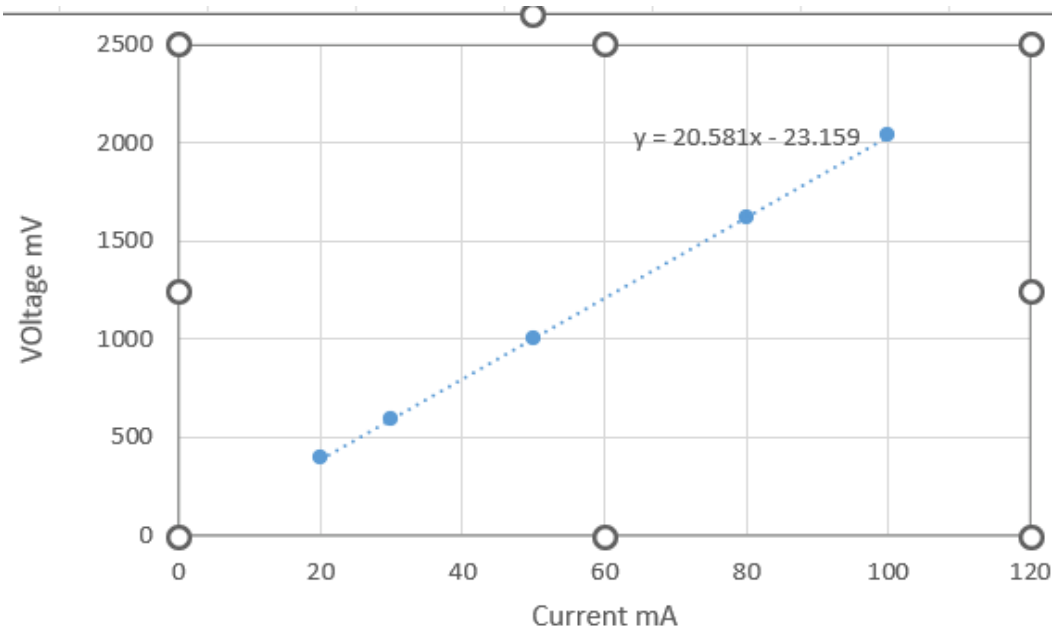
*(*Note this approach had a major pitfall and later opted for digital method of integrating which will be discussed in later relevant section.)*

1.7 Calibration and Practical Considerations

Because the exact mutual inductance $M$of a hand-wound Rogowski coil is difficult to calculate analytically, the entire sensor assembly (coil + integrator*(*software)* + amplifier) is treated as a black box during calibration.

Known currents are applied using a function generator and reference resistor, and the corresponding output voltages are measured. A regression model is then used to establish the relationship between measured voltage and actual current.

This empirical calibration approach compensates for coil non-idealities and component tolerances. The figure shows the measurement with applied current (mA) *VS* the output of the system(mV) and linear fitting of the samples in *Excel*.



2 System Architecture.

2.1 Components: Hardware Blocks

- **ATmega328P (16 MHz)**
  – Performs ADC sampling, filtering, frequency estimation, and display control.
- **Rogowski Coil + Integrator/Signal Conditioner**
  – Produces a small AC voltage proportional to di/dt, conditioned to ADC range.
- **ADC (10-bit, 0–1023)**
  – **Sa**mples the waveform at 1 kHz.
- **SSD1306 OLED (128×32, I^2C)**
  – Displays Ipeak, Irms, and frequency.
- **ESP8266**
  – Connects to WiFi and publishes Current and frequency via MQTT protocol.
- **Edge device(computer)**
  - A computer simulating as an edge device listening to the MQTT topic published by ESP8266.

Initially the system implementation was based on purely analog integrator which posed challenge for proper integration *(if even integration at all)*. The system behaved more as a high gain amplifier.

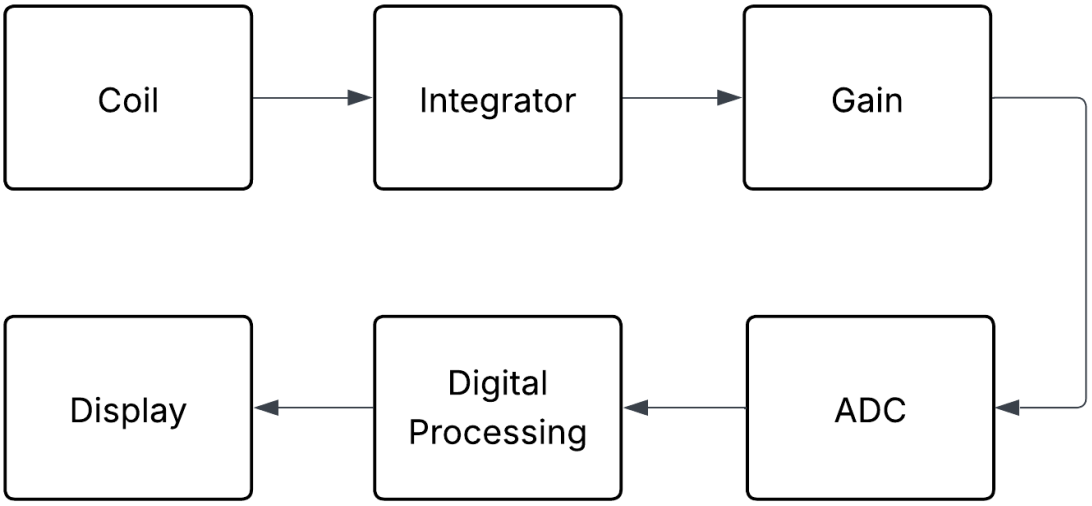


Fig: Initial system architecture

Later the *integration* was done in software after the adc sampling and filtration stage. The ESP8266 was added for its WiFi capability so as to send the measurement via MQTT to an edge device *(* here a computer listening to the MQTT topic using python script)*. The ATmega328P communicates ESP8266 via SPI interface.

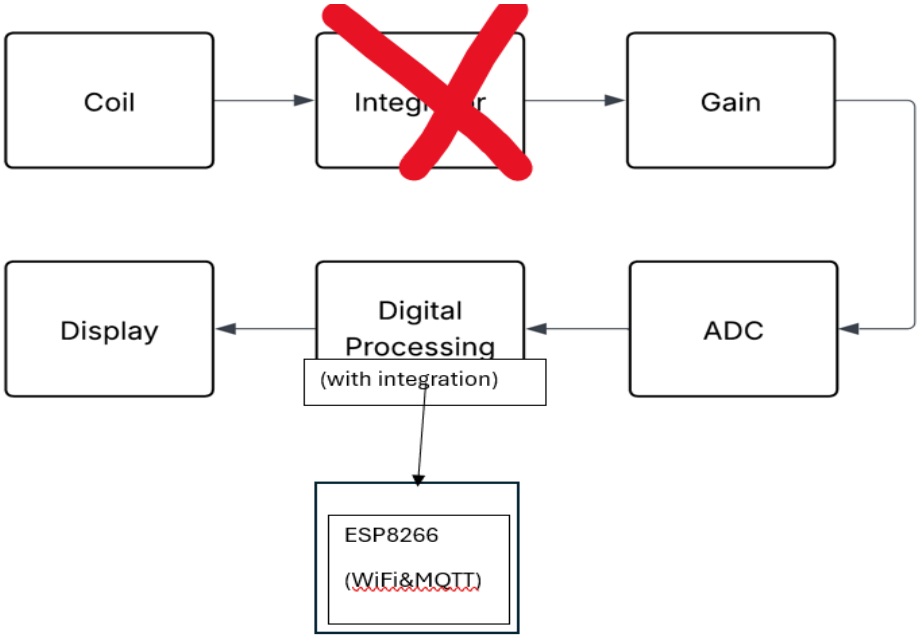


Fig: Revised System architecture

3 Bare-Metal Firmware Architecture

This section explains the role and operation of each firmware subsystem implemented on the sensor node. The firmware represents the cyber layer of the CPS and is responsible for deterministic data acquisition, real-time signal processing, and communication with higher layers. Each subsection below describes *what the block does*, *why it is required*, and *how it contributes to CPS functionality*.

The bare-metal firmware implementation code is maintained in the public GitHub repository:

https://github.com/pointertoaperson/current_sensor

This firmware runs directly on an ATmega328P microcontroller (16 MHz) without any operating system, RTOS, or Arduino framework. The ESP8266 code is based on ESP-IDF framework.

All peripherals in ATmega are configured using direct register-level programming in C, ensuring deterministic behaviour and full control over hardware resources.

3.1 Firmware Structure

The firmware is organized into modular source files:

- main.c – system initialization and main execution loop
- fir.c/fir.h – fir filtering function source and header file
- adc.c / adc.h – ADC configuration and sampling routines
- timer.c / timer.h – hardware timer configuration
- i2c.c / i2c.h – custom I^2C (TWI) driver
- spi.c/spi.h – custom SPI driver for Sending data to ESP8266
- ssd1306.c / ssd1306.h – OLED display driver
- fonts.h – bitmap font definitions for visualization

The main loop remains intentionally minimal, with most system functionality handled through interrupt service routines (ISRs).

3.2 Peripheral Initialization (Bare-Metal)

3.2.1 ADC Initialization (Single-Ended, AVcc Reference)

Configures the ADC to sample the analog signal from the Rogowski coil conditioning circuit using AVcc (5 V) as reference.

```
void adc_init(void) {
    ADMUX = (1 << REFS0);                // AVcc as reference voltage
    ADCSRA = (1 << ADEN)  |              // Enable ADC
             (1 << ADIE)  |              // Enable ADC interrupt
             (1 << ADPS2) | (1 << ADPS1); // Prescaler = 64
}
```

3.2.2 Timer1 Initialization (CTC Mode, 1 kHz Interrupt)

Provides a precise hardware time base for periodic sampling without software delays.

```
void timer1_init(void) {
    TCCR1B |= (1 << WGM12);      // CTC mode
    OCR1A = F_CPU / 1000;        // 1 kHz interrupt rate
    TIMSK1 |= (1 << OCIE1A);     // Enable Timer1 compare interrupt
    sei();                       // Enable global interrupts
```

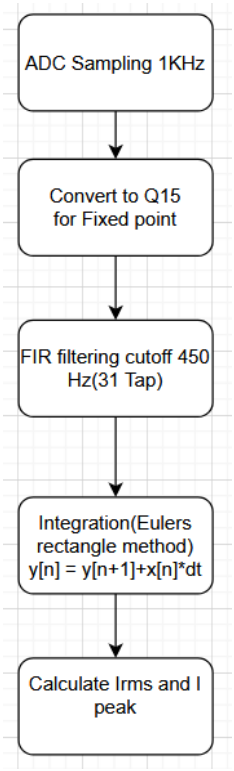
```
}
```

3.2.3 I²C (TWI) Initialization for SSD1306

initializes the I²C peripheral for communication with the OLED display.

```
void i2c_init(void) {
    TWSR = 0x00;                                // Prescaler = 1
    TWBR = ((F_CPU / 100000UL) - 16) / 2;       // 100 kHz I2C clock
    TWCR = (1 << TWEN);                         // Enable TWI
}
```

4 Digital Processing and code explanation.



4.1.1 ADC sampling

To achieve 1 kHz ADC sampling on the ATmega328p:

1. Timer1 is configured in CTC mode to generate an interrupt every 1 ms (1 kHz).
2. In the Timer1 ISR, we start an ADC conversion by setting the ADSC bit.
3. When the ADC finishes, the ADC ISR (ADC_vect) reads the value and stores it in a buffer.
4. This process repeats until the buffer is full.

- Using Timer1 ensures precise, evenly spaced sampling.

- The ADC ISR makes sure each sample is captured immediately after conversion.
- According to Nyquist, a 1 kHz sampling rate can capture signals up to 500 Hz *(in theory)* without aliasing.

```
// ADC Initialization

void adc_init(void) {
    ADMUX  = (1 << REFS0); // Use AVcc as reference, ADC0 channel
    ADCSRA = (1 << ADEN) |  // Enable ADC
             (1 << ADIE) |  // Enable ADC interrupt
             (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0); // Prescaler 128
}

// Timer1 Initialization (1 kHz)

void timer1_init_1khz(void) {
    TCCR1A = 0;
    TCCR1B = (1 << WGM12) | (1 << CS11) | (1 << CS10); // CTC mode, prescaler=64
    OCR1A  = (F_CPU / (64UL * 1000)) - 1;             // Compare match for 1 kHz
    TIMSK1 |= (1 << OCIE1A);                          // Enable Timer1 Compare A interrupt
}

// Timer1 ISR: Start ADC

ISR(TIMER1_COMPA_vect) {
    ADCSRA |= (1 << ADSC); // Start ADC conversion
}

// ADC ISR: Read sample

ISR(ADC_vect) {
    samples[sample_idx++] = ADC; // Store ADC value
    if (sample_idx >= SAMPLE_COUNT) {
        samples_ready = 1;       // Flag buffer full
        TIMSK1 &= ~(1 << OCIE1A); // Stop Timer1
        ADCSRA &= ~(1 << ADEN);   // Disable ADC
    }
}
```

4.1.2 FIR filter and Fixed-point arithmetic

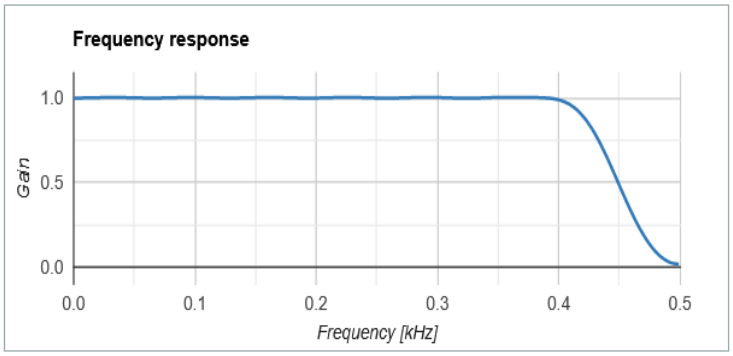
Since the sampled signal has high frequency noises, we use Finite Impulse Response (FIR) filter to remove those. The filter was designed using an online tool (https://fiiir.com/) for *cutoff 450Hz Transition bandwidth 100Hz* and *sampling frequency of 1KHz*. This is a 31-tap filter.

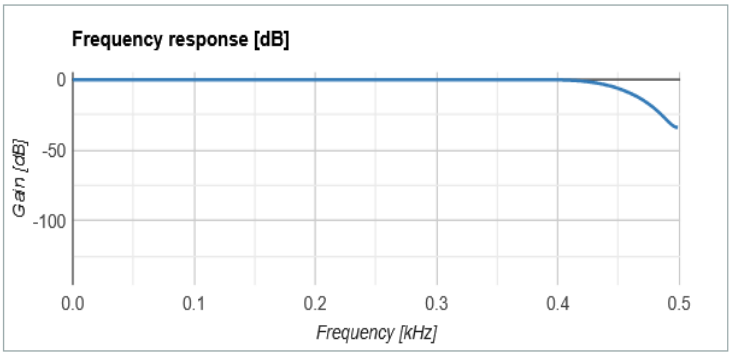
-0.001700447766765450	0.021670197363365697	0.097534726242944722	0.004355381155512681
0.001950458528754498	-0.034474512789894253	-0.089976467760056286	0.000000000000000011
-0.002376422241525983	0.049100429265352338	0.078427104501585118	-0.002079774815281043
0.002621374076042475	-0.064269416213281236	-0.064269416213281264	0.002621374076042476
-0.002079774815281043	0.078427104501585104	0.049100429265352359	-0.002376422241525982
0.000000000000000011	-0.089976467760056286	-0.034474512789894260	0.001950458528754498
0.004355381155512680	0.097534726242944722	0.021670197363365715	-0.001700447766765450
-0.011523516174424046	0.901481773255341401	-0.011523516174424051	

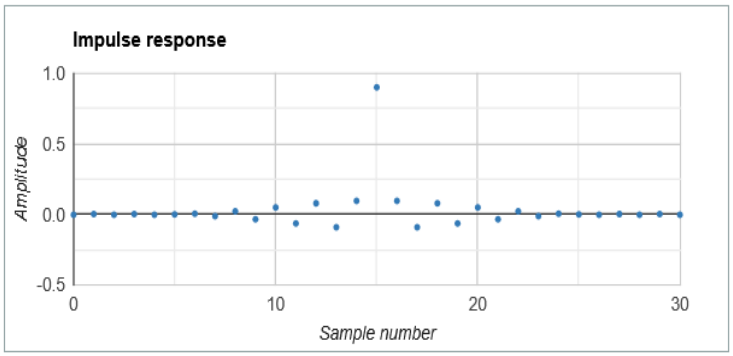
Filter equation is given by:

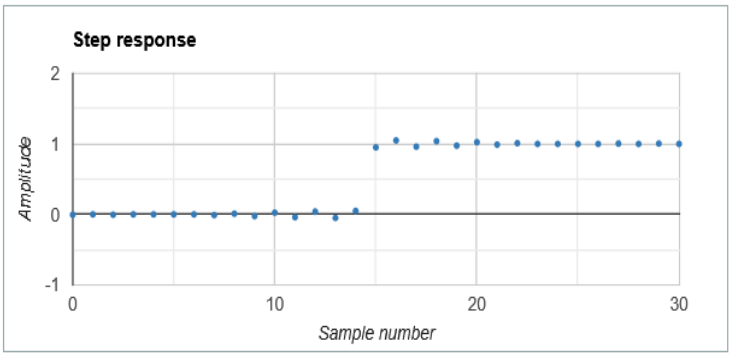
$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k]$$; where N = 31

Filter Characteristics









Since the coefficients are very small float values. This filter will be computationally and memory wise very expensive operation. Thus, fixed point arithmetic, precisely Q15, is used for the filtering and following integration chain. And is converted as follows:

$$Q15_{integer} = floatvalue \times 2^{15} \;\; with\; range[-1,1)$$

```
#include "fir.h"

// Circular buffer to store last FIR_TAPS input samples
int16_t fir_buffer[FIR_TAPS] = {0};

// Q15 scaled symmetric coefficients (32 coefficients)
// Symmetry allows reducing multiplications by half
const int16_t fir_coeff[FIR_TAPS] = {
  -56, 64, -78, 86, -68, 0, 143, -377, 710, -1128,
  1605, -2105, 2573, -2952, 29530, 29530, -2952, 2573,
  -2105, 1605, -1128, 710, -377, 143, 0, -68, 86, -78,
  64, -56, 0, 0
};

// FIR filter function: input is Q15 sample, output is Q15 filtered sample
fir_process(int16_t input)
{
    uint8_t fir_index = 0;           // Index for circular buffer
    fir_buffer[fir_index] = input;   // Store new input in buffer
    int32_t acc = 0;                 // 32-bit accumulator to prevent overflow

    // Loop over half of the taps because coefficients are symmetric
    for (uint8_t i = 0; i < FIR_TAPS / 2; i++)
```

```
    {
         // Calculate indices for mirrored input samples
        uint8_t a_idx = (fir_index + i) % FIR_TAPS;
        uint8_t b_idx = (fir_index + FIR_TAPS - 1 - i) % FIR_TAPS;

        // Multiply coefficient by sum of mirrored samples and accumulate
        // int32_t used because Q15 * Q15 = Q30
        acc += (int32_t)fir_coeff[i] * (fir_buffer[a_idx] + fir_buffer[b_idx]);
    }

    // Move circular buffer index forward
    fir_index = (fir_index + 1) % FIR_TAPS;

    // Convert from Q30 back to Q15 by shifting 15 bits
    acc = acc >> 15;

    // Saturate to 16-bit signed integer range
    if (acc > 32767) acc = 32767;
    if (acc < -32768) acc = -32768;

    return (int16_t)acc; // Return filtered Q15 output
}
```

4.1.3 Frequency Estimation

The function estimates the frequency of a discrete-time signal by counting zero-crossings. A zero-crossing occurs when the signal changes sign (from positive to negative or negative to positive).

For a periodic signal like a sine wave:

- Two zero-crossings correspond to one complete cycle.
- By counting the number of zero-crossings in a known sample window, the function can estimate the signal frequency.

```
// Estimate frequency from filtered or integrated samples
float estimate_frequency(int16_t *data, uint8_t count)
{
    // Determine the sign of the first sample: 1 if positive, 0 if negative
    uint8_t last = (data[0] >= 0);

    // Counter for zero-crossings
    uint16_t crossings = 0;

    // Loop through all samples to detect zero-crossings
    for (uint8_t i = 1; i < count; i++)
    {
        // Determine the sign of the current sample
        uint8_t curr = (data[i] >= 0);

        // If the sign has changed compared to the previous sample,
        // a zero-crossing occurred
        if (curr != last)
            crossings++;

        // Update last sample sign for next iteration
        last = curr;
    }

    // Each cycle has two zero-crossings, so divide by 2
    // Multiply by SAMPLE_RATE_HZ / count to scale cycles in the window to Hz
    float freq = ((float)crossings / 2.0f) * ((float)SAMPLE_RATE_HZ / (float)count);

    return freq; // Return estimated frequency in Hz
}
```

4.1.4 Integration

Forward euler method is used for the integration and is given by:

$$y[n]=y[n-1]+x[n]\cdot dt$$

```
   //  Fixed-point integration scaling
    const int32_t dt_scaled = 327; // integration increment scaling
    const uint8_t dt_shift = 10;   // right shift to fit int16_t safely

// Apply FIR filter and fixed-point integration
        int32_t y_acc = 0;   // 32-bit accumulator

        for (uint8_t i = 0; i < SAMPLE_COUNT; i++)
        {
            int16_t centered = samples[i] - 512;        // center ADC to 0
            int16_t filtered = fir_process(centered);   // Q15 filtered sample

            // Fixed-point integration: y[n] = y[n-1] + x[n]*dt_scaled
            y_acc += (int32_t)filtered * dt_scaled;

            // Scale down to fit int16_t safely
            integrated[i] = (int16_t)(y_acc >> dt_shift);

            samples[i] = filtered; // store filtered sample back if needed
        }
```

4.1.5 Calculation of Current

The peak and RMS current are computed from the integrated signal array to quantify both the instantaneous and effective current values. For each sample $y[n]$in the integrated array, the absolute value is taken to ensure positivity, and the peak current is determined as the maximum absolute value:

$$I_{\text{peak}} = \max\left(|\, y[n]\,|\right), n = 0,1,\dots,N-1$$

Simultaneously, the RMS current is computed as the square root of the mean of the squared absolute values of all samples:

$$I_{\text{RMS}} = \sqrt{\frac{1}{N}\sum_{n=0}^{N-1} |\, y[n]\,|^2}$$

Finally, both values are scaled from fixed-point representation to physical units (volts or milliamps) using a precomputed scaling factor. This

approach allows efficient computation on a microcontroller while providing accurate peak and effective current measurements, capturing the overall energy content of the signal after FIR filtering and integration.

```
//  Compute peak and RMS from integrated array
int16_t max_integrated = 0;       // Variable to store maximum (peak) value
int32_t sum_sq_integrated = 0;    // Accumulator for sum of squares (for RMS)

// Loop through all samples in the integrated array
for (uint8_t i = 0; i < SAMPLE_COUNT; i++)
{
    int16_t val = integrated[i]; // Get current sample

    if (val < 0) val = -val;     // Take absolute value for peak measurement

    if (val > max_integrated)    // Update peak if current value is larger
        max_integrated = val;

    sum sq integrated += (int32 t)val * val; // Accumulate square of the sample for RMS
}

// Scale from fixed-point to physical units (e.g., volts or mA)
float integrated_peak = max_integrated * scale;                // Peak current
float integrated_rms  = sqrt((float)(sum_sq_integrated / SAMPLE_COUNT)) * scale; // RMS
current
```

4.1.6 Display

The computed values are displayed in SSD1306 oled display in human readable form. The display is connected via I2C and the driver and fonts are written all in baremetal. The code explanation is not done in this report.



4.1.7 SPI interface

SPI interface is used to transfer the values from ATmega328p to ESP8266.

The values are sent in following format

[Command(0x02)] [Sub(0x02)] [Peak(2-byte)] [RMS(2-byte)] [RMS(2-byte)] [Freq(2-byte)]

ESP8266 requires 0x02 command byte to acknowledge the start of transmission.

```
#include "spi.h"

// Initialize SPI as master
 spi_init(void) {
    // Set MOSI, SCK, SS as output
    DDRB |= (1 << MOSI_PIN) | (1 << SCK_PIN) | (1 << SS_PIN);
    // Enable SPI, Master, set clock rate fck/16
    SPCR = (1 << SPE) | (1 << MSTR) | (1 << SPR0);
    CS_HIGH(); // Set SS high
}

// Transmit a single byte
 spi_send_byte(uint8_t data) {
    SPDR = data;               // Load data into SPI data register
    while (!(SPSR & (1 << SPIF))); // Wait until transmission complete
}

// Transmit a 16-bit value (high byte first)
 spi_send_uint16(uint16_t data) {
    spi_send_byte((data >> 8) & 0xFF); // MSB
    spi_send_byte(data & 0xFF);        // LSB
}

void spi_send_current(uint16_t peak, uint16_t rms, uint16_t freq) {
    CS_LOW();
     delay us(5);          // Small delay before starting

    // Header / command
    spi_send_byte(0x02);   // Command
    spi_send_byte(0x00);   // Sub-command

    // Data
    spi_send_uint16(peak);
    spi_send_uint16(rms);
    spi_send_uint16(freq);



    CS_HIGH();
    _delay_us(5);          // Small delay after finishing
}
```

The values are multiplied by 100 before transmitting so that these can be safle converted to uint16_t and not much precision is lost.

```
spi_send_current((uint16_t)(integrated_peak * 100.0f) ,(uint16_t)(integrated_rms  * 100.0f),
(uint16_t)(freq * 10.0f));
```

4.2 ESP8266 as SPI Slave with Wi-Fi Connectivity and MQTT

CommunicationThe ESP8266 is configured as an SPI slave to receive data from the ATmega microcontroller. It then connects to a Wi-Fi network and publishes the received data using the MQTT protocol. The ESP8266 code is developed using the ESP_RTOS_SDK, a framework similar to esp-idf, which is provided by the vendor and available at

https://github.com/espressif/ESP8266_RTOS_SDK.

The code to it is in the project GitHub.

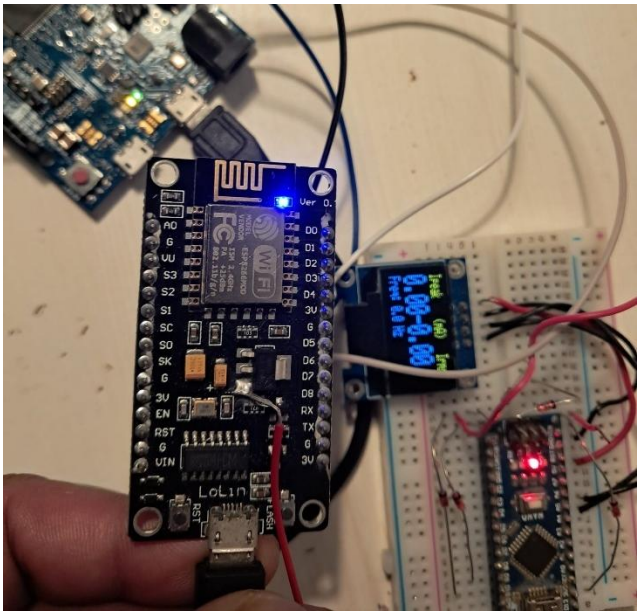


Fig: ESP8266

4.2.1 Json Format for publishing values

The sensor values from the ATmega are formatted as a JSON string on the ESP8266 before being transmitted. Each value Ipeak, Irms, and Freq is converted to floating-point format and included in the JSON structure, for example: { "Ipeak": 2.34, "Irms": 1.56, "Freq": 50.12 }. This JSON string is then published to the MQTT broker under the topic /esp8266/sensor using the ESP8266 MQTT client. Any device or application subscribed to this topic can receive the data in real-time and process it for visualization or further analysis.

```
void mqtt_publish_values(uint16_t Ipeak, uint16_t Irms, uint16_t Freq)
{
    if (!mqtt_connected || !client) {
        ESP_LOGW(TAG, "MQTT not connected, skipping publish");
        return;
    }

    char payload[128];
    snprintf(payload, sizeof(payload),
             "{ \"Ipeak\": %f, \"Irms\": %f, \"Freq\": %f }",
             Ipeak/100.0f, Irms/100.0f, Freq/100.0f);

    esp_mqtt_client_publish(client, "/esp8266/sensor", payload, 0, 1, 0);
    ESP_LOGI(TAG, "Published JSON: %s", payload);
}
```

5 Edge Processing and Visualization

A Python script was developed using the Paho-MQTT library to subscribe to

the topic “esp8266/sensor” on the public “brokerbroker.hivemq.com”. The received sensor data is visualized in real-time using matplotlib.animation, displaying an animated graph with a 1-second rolling window for dynamic monitoring.

The script is named edge_listener.py and can be found in root of the repository.

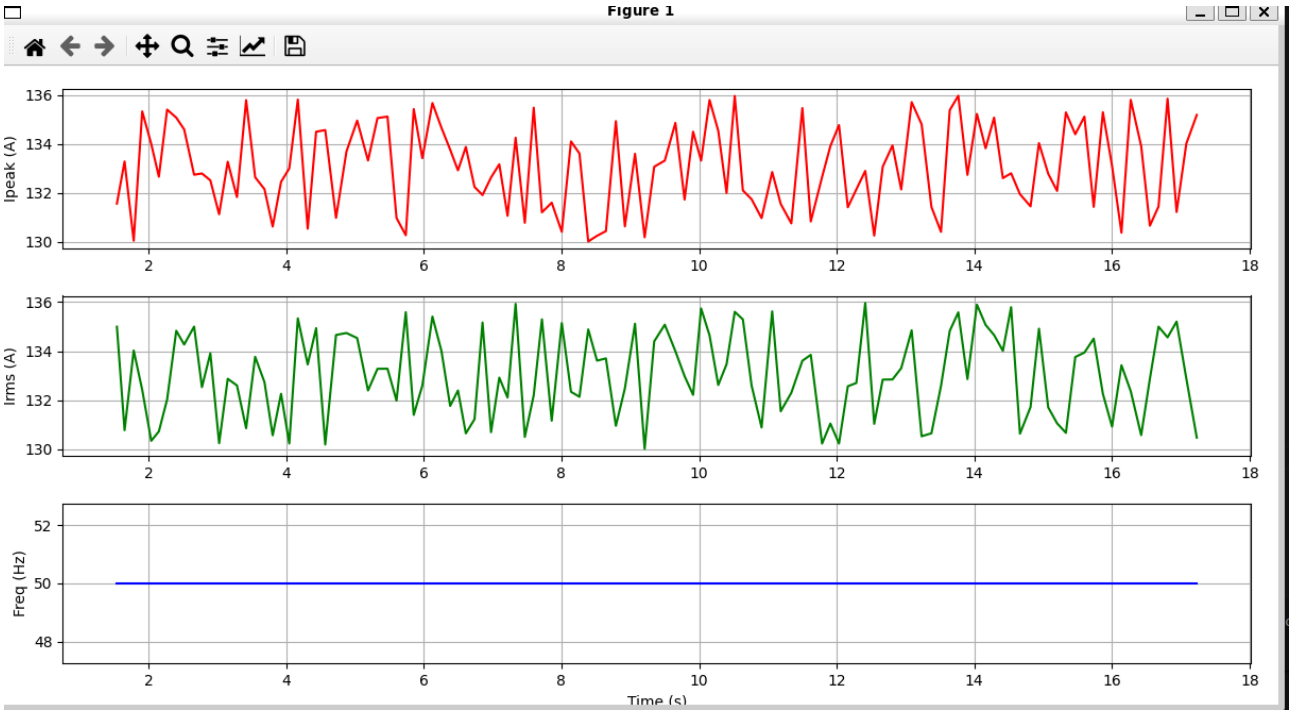


Fig. Visualization by edge computing node

6 Project Prototype

A bread board prototype was built for the project. And measurements were taken and compared with the multimeter measurement. The reading had very high fluctuation and although it is not rigorously examined, it is believed to have about 70% of the accuracy after the second architecture implementation.

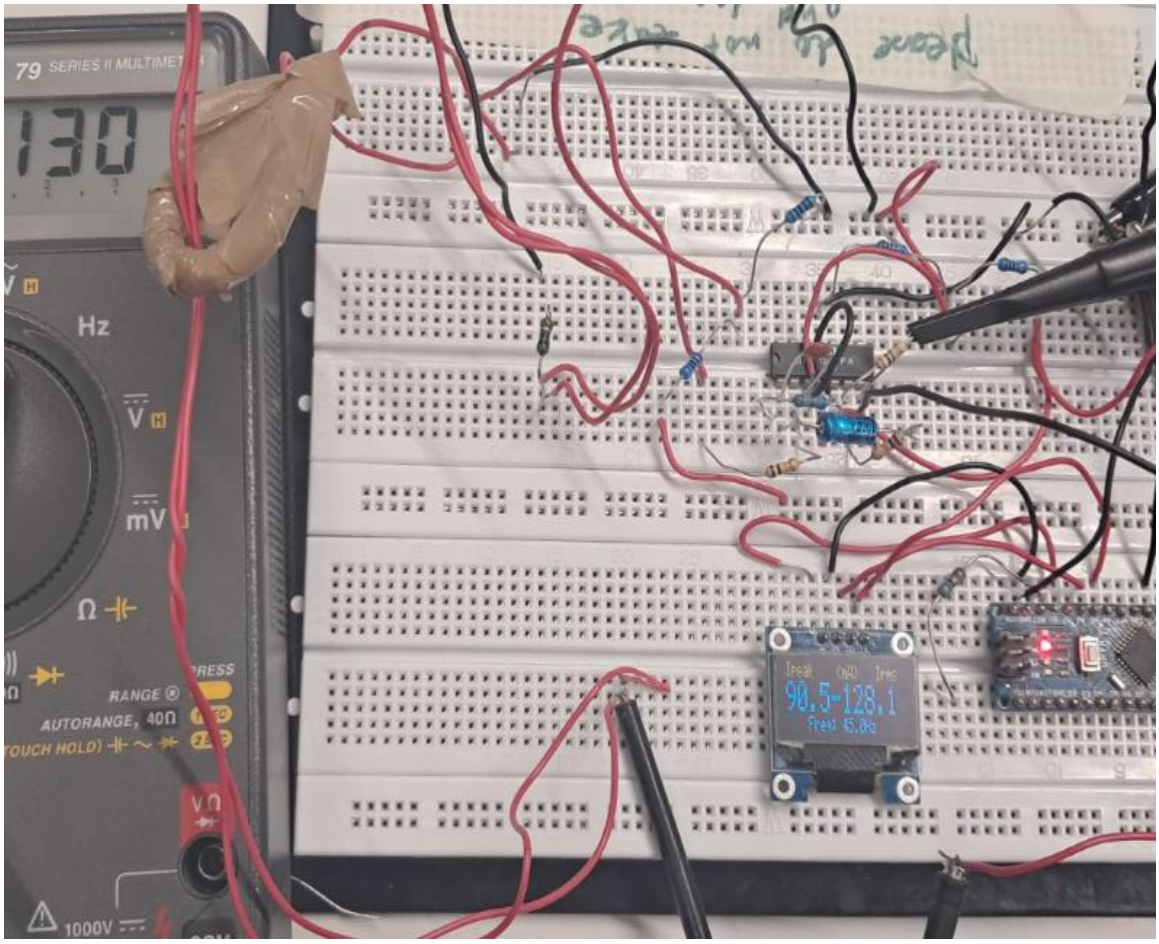


Fig: Breadboard prototype with measurement

7 Power uses.

The image below shows the current consumption of the ATmega328p for every sampling cycle. It can me noted that for every conversion by ADC there is increase in the current consumption.





Fig: voltage across 10ohm shunt resistor

The current draw for each sampling time :

75.3mV/10R = 7.53mA

The power consumption for each sampling time :

7.53mA * 5V = 37.65mWatt.

8 Challenges

- Coil Fabrication:
  Building a Rogowski coil for low-frequency and low-current applications by hand is challenging due to the precise winding required for consistent performance.

- Signal Integration:
  Designing and implementing an integrator circuit with sufficient sensitivity for accurate measurement is difficult, especially when dealing with very small induced voltages.

- Noise and Interference:
  The measurements are highly susceptible to electrical noise, which can distort the signal and make calibration and data accuracy difficult, particularly at low currents and low frequencies.

- Signal Processing:
  Properly integrating the coil output without introducing offset or drift requires careful circuit design and filtering, as improper integration can result in distorted or unreliable waveforms.