



SAPIENZA
UNIVERSITÀ DI ROMA

Current Monitoring System with Arduino and Hall Effect Sensor

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Corso di Laurea in Ingegneria Informatica e Automatica

Candidate

Cristian Di Iorio

ID number 1983177

Thesis Advisor

Prof. Giorgio Grisetti

Academic Year 2023/2024

Current Monitoring System with Arduino and Hall Effect Sensor

Bachelor's thesis. Sapienza – University of Rome

© 2024 Cristian Di Iorio. All rights reserved

This thesis has been typeset by L^AT_EX and the Sapthesis class.

Author's email: diiorio.1983177@studenti.uniroma1.it, cristiandiiorio12@gmail.com

Contents

1	Introduction	2
2	Related work	3
2.1	AVR/Arduino	3
2.1.1	Interrupts	3
2.1.2	Timers	4
2.1.3	ISR	4
2.1.4	Analog to Digital converter	4
2.1.5	Makefile	5
2.1.6	Pseudoterminal Interfaces	6
2.2	Electrical loads	7
2.3	Current sensor	8
2.3.1	Hall Effect sensors	8
2.4	Operational Amplifier	9
2.4.1	Clipping and saturation	10
2.5	Data Interpretation	12
2.5.1	Nyquist-Shannon Sampling Theorem	12
2.5.2	Root Mean Square	13
2.5.3	Least Squares Regression	14
2.6	Docker	14
2.7	Instruments	15
3	My Contribution	16
3.1	Hardware	16
3.1.1	System wiring with 5 Volt power supply	16
3.1.2	System wiring with 9 Volt power supply	17
3.1.3	Arduino ATMega2560	17
3.1.4	Current Sensor Board	17
3.1.5	Current sensor	18
3.1.6	Operational Amplifier	18
3.1.7	Potentiometer	19
3.2	Software	19
3.2.1	Meter - Arduino side	19
3.2.2	Communication between meter and receiver	19
3.2.3	Receiver - Linux side	20
3.2.4	Makefile	21
3.3	Docker	21

4	Use cases	24
4.1	Boot up	24
4.2	Meter	24
4.2.1	Setup	24
4.2.2	Sensor data interpretation	26
4.2.3	Least Squares Regression	27
4.3	Receiver	28
4.3.1	Setup	28
4.3.2	Online mode	29
4.3.3	Query and Clearing mode	31
4.4	Docker and Website	32
5	Conclusions	33
	Bibliography	34
	Acknowledgements	36

Alla mia famiglia e ai miei amici

Chapter 1

Introduction

In this project we want to build a current monitoring system. Monitoring electrical current is crucial in a wide array of applications, from industrial machinery to household electronics. Precise and real-time current measurement is essential for ensuring safety, optimizing energy consumption and protecting equipment from overloads or faults. However, traditional current monitoring systems can be expensive and complex.

This project aims to showcase a low-cost, easy-to-build current monitoring system using an Arduino microcontroller, a Hall effect current sensor and a Linux computer. The Hall effect sensor, known for its non-intrusive and accurate current measurement, offers an efficient alternative to more cumbersome and expensive methods. By integrating it with an Arduino, we want to provide a solution that can be easily adapted for various projects. The Linux computer ensures that the user has an interface to program the Arduino, debug it and view the data that is collected.

Throughout this thesis, our goal is to create a simple and reliable current monitoring system that anyone can implement. We aim to explain each step in detail, including both the *how* and the *why* behind our actions. Additionally, we try to demonstrate how fundamental concepts from physics, electronics, and signal theory influence the design and components of our system.

Chapter 2

Related work

This chapter provides an in-depth review of the scientific literature and key concepts that form the foundation of this thesis.

2.1 AVR/Arduino

Arduino is an open-source electronics platform known for its versatility in creating interactive projects. While it is commonly associated with the Arduino IDE (Integrated Development Environment), the hardware can also be programmed using AVR, a low-level programming language that provides greater control over the microcontroller's functions.

The ATMega 2560 microcontroller is capable of interfacing with various sensors and peripherals, such as current sensors, for real-time monitoring and control. Programming the ATMega 2560 with AVR allows for direct access to the microcontroller's registers and memory, enabling precise and efficient operation, which is particularly beneficial for applications that require fine-tuned performance and a deeper understanding of hardware-level programming. In this project we made ample use of low-level features like interrupts and timers.

2.1.1 Interrupts

An interrupt is a signal sent to the microcontroller that temporarily halts the execution of the main program. The benefits of using interrupts are efficiency and responsiveness. They allow the microcontroller to perform other tasks while waiting for an event rather than checking for it continuously. This also helps lower power consumption as the microcontroller can be put in a low-power state between events. The use of interrupts also helps with reducing the response time to events.

When an interrupt occurs, the microcontroller pauses its task, saves its state and jumps to a special function known as an Interrupt Service Routine (ISR). The ISR is a short and fast bit of code that is made to address the specific event that triggered the interrupt. According to the documentation of AVR-GCC [1], the vector table is preconfigured to link to interrupt routines with specific, predefined names. By using the correct name for the routine, it will automatically be invoked when the corresponding interrupt occurs.

There are different types of interrupts like external, timer and usart. In this project we used timer and usart interrupts to manage the execution flow of the code. USART (Universal Synchronous and Asynchronous Receiver-Transmitter) interrupts are triggered by events like receiving or transmitting data. Timer interrupts depend on the timer system of the Arduino.

2.1.2 Timers

The Mega2560 has 5 timers [2], which can be used for different purposes, like recording the exact time an external event happens, timing specific intervals, creating waveform signals and initiating interrupts at defined time intervals.

A timer has a counter (TCNT), that is incremented at each clock cycle (16MHz) or a fraction of it that is a power of 2. Each timer has 3 output compare registers (OCR) and when the counter value matches the OCR value an event is generated. An event can result in an interrupt or toggling a pin. The actions taken after an event occurs are controlled through a set of special registers (TCCR).

2.1.3 ISR

An Interrupt Service Routine (ISR) is a special function that gets executed when a specific interrupt event occurs. The microcontroller uses an interrupt vector table to map specific interrupt sources to their corresponding ISRs. Each interrupt has a unique vector that points to the address of the ISR in memory. When an interrupt occurs, the microcontroller automatically consults this vector table to find and execute the correct ISR. It is crucial to ensure that each ISR is defined with the correct interrupt vector.

When an interrupt event occurs, the microcontroller automatically saves certain registers (such as the program counter and status register) to the stack. This allows the ISR to execute without corrupting the state of the main program. After the ISR finishes execution, these registers are restored and the program resumes from the exact point where it was stopped. This mechanism allows the microcontroller to respond quickly to external or internal events without needing to constantly check for them. In the AVR-Arduino environment, ISRs are defined using a specific syntax:

```
ISR(TIMER1_COMPA_vect) {  
    //code to handle the interrupt  
}
```

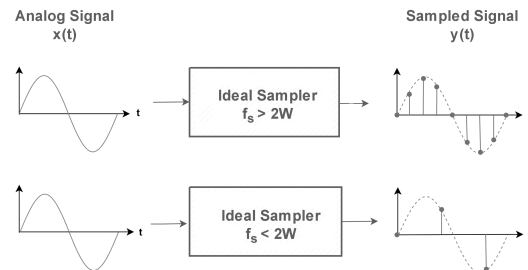
The argument of the ISR is the interrupt vector corresponding to the specific event that triggers the ISR. Since an ISR halts the main program execution, the code inside it should be kept as short and efficient as possible to ensure the system remains responsive. Functions that cause delays or blocking I/O operations should be avoided within an ISR, as they can lead to unresponsiveness or missed interrupts. Instead, the ISR should set flags or store data to be processed in the main program loop, allowing the main code to handle any time-consuming tasks.

2.1.4 Analog to Digital converter

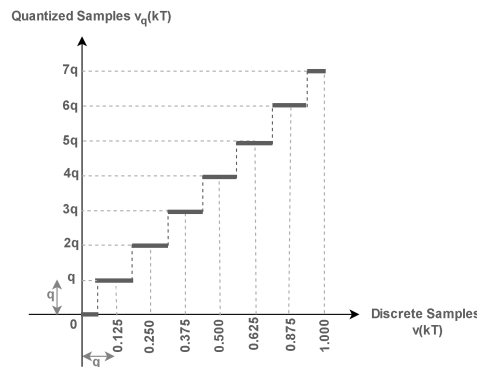
An analog to digital converter (ADC) is an electronic device that converts continuous analog signals into a discrete digital representation. It is a necessary step when processing analog signals in digital circuits since those work using only two discrete states, High (1) and Low (0). This means that since analog signals have an

infinite number of different values they are not compatible with digital circuits. The conversion from analog to digital consists of three steps:

1. Sampling, when the analog signal is sampled at regular intervals according to the sampling rate. The higher the sampling rate, the more accurate the digital signal will be [3].



2. Quantization, when the ADC replaces each sampled value with the closest approximated value, which is chosen from a finite set of discrete values. The most common quantization levels are 8-bit, 16-bit and 24-bit. Using more levels helps in reducing quantization errors.



3. Encoding, when the quantized value is converted into a binary number. The number of bits used in this step depends on the quantization level used by the ADC.

This kind of converter can be used to make digital circuits interact with the real world. For example they allow digital computers to measure sound waves, light or voltages.

2.1.5 Makefile

A Makefile is a special file used by the make utility to automate the process of building projects. It is useful in large projects because it specifies how to compile and link a program. It defines a set of rules and dependencies that the make utility uses to determine the correct sequence of operations. These rules typically involve compiling source code files into object files and linking those into an executable.

According to the official documentation [4], each rule consists of:

- Target: the name of the file that is going to be created.
- Prerequisites: files that are used as input to create the target, like source code files.
- Recipe: commands to be executed when building the target from the prerequisites.

Here is a sample Makefile:

```
# Define the compiler to use
CC = gcc

# Define flags for the compiler
CFLAGS = -Wall -g

# Define the target executable
TARGET = myprogram

# List of source files
SRCS = main.c misc.c

# List of object files (replace .c with .o)
OBJS = $(SRCS:.c=.o)

# Default rule to build the target
all: $(TARGET)

# Rule to build the target executable
$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

# Rule to compile .c files into .o files
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

# Rule to clean up compiled files
clean:
    rm -f $(OBJS) $(TARGET)

# Phony targets (not actual files)
.PHONY: all clean
```

We used Makefile because it provides automation and portability to the project, as shown in [3.2.4](#).

2.1.6 Pseudoterminal Interfaces

In our project we will be using device files, which are created by the kernel to provide an easy-to-user interface to make serial communication easier.

According to the man page *pty(7)* [5] a pseudoterminal is a pair of virtual character devices that provide bidirectional communication between the two ends of the channel, called the *master* and *slave*. The slave side of a pseudoterminal functions just like a traditional terminal interface. A process that requires a terminal connection can open the slave side of a pseudoterminal and be controlled by a program connected to the master side. Any input sent to the master side is received

by the process on the slave side as if it were typed directly on a terminal.

For instance, sending an interrupt character (like *CTRL-C*) to the master device will trigger an interrupt signal (SIGINT) for the foreground process group connected to the slave. Similarly, any data written to the slave side can be read by the program connected to the master side. The data flow between the master and slave sides occurs asynchronously, similar to how data flows with a physical terminal. There are two pseudoterminal APIs: BSD-style and UNIX 98-style. In this project we will be using the former.

2.2 Electrical loads

Electrical loads can be categorized based on how they use electrical energy and how they interact with the power supply. These are the main types:

- **Resistive** loads consume electrical energy mainly in the form of heat. The relationship between voltage and current is described by Ohm's law $V = Ri$, so it's linear. This means that the current waveform follows the voltage waveform, resulting in a sinusoidal current waveform if the supply is using alternating current. Some examples are incandescent light bulbs and electric heaters.
- **Inductive** loads involve magnetic fields, which create a phase shift between voltage and current. So the current lags behind the voltage. The current waveform is sinusoidal, however there might be some phase lag. Examples of devices of this kind are electric motors, transformers and fans.
- **Capacitive** loads store energy in an electric field. They cause the current to lead the voltage, a behaviour opposite to inductive loads. The current waveform is sinusoidal and it leads the voltage waveform. This type of loads are not commonly used in standalone devices however they are often found in circuits for power factor correction.
- **Non-linear** loads draw current in a non-sinusoidal manner, causing the current waveform to be highly distorted, often with sharp pulses and irregular shapes. Some examples of devices that use this type of load are computers, phone chargers and any device that uses switched-mode power supplies.

We avoided testing our sensor with non-linear loads because they produce a highly distorted current waveform. As shown here:

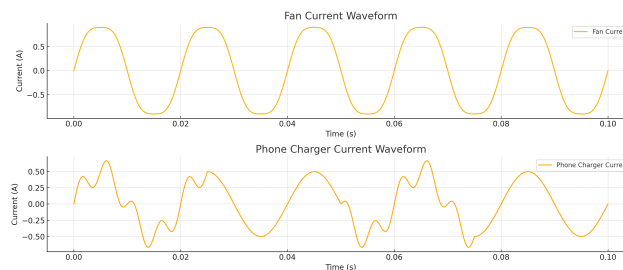


Figure 2.1. Comparison between the current waveforms of a fan and phone charger

The current waveform of the fan is sinusoidal with some minor variations, while the waveform of the phone charger is characterized by high-frequency switching

pulses. This causes the waveform to be less smooth with irregular pulses.

The nature of this current waveform would have been difficult to capture accurately because of the narrow pulses with high peak current combined with the overall distortion of the waveform. Due to these concerns, we only tested the current sensor using resistive and inductive loads, like incandescent light bulbs and fans.

2.3 Current sensor

There are many ways to measure current flowing through a conductor.

The simplest approach is applying Ohm's law $V = Ri$ to measure the current by using the voltage drop across a current-carrying conductor with known resistance (e.g., a shunt resistor). However as explained in (Crescentini 2021) [6], this configuration lacks galvanic isolation, which means that there is a direct path for current to flow through. In turn this leads to power loss which can be calculated by substituting Ohm's law in the formula for power $P = Vi$. The result is a power loss of $P = Ri^2$. Since it increases with the square of the current, this approach is not compatible with high-current applications as it would suffer from extreme loss of power.

Magnetic field sensors are a slightly more complex alternative (Ziegler 2009) [7]. They are non-invasive and do not cause heat dissipation. They are also safer as they provide an inherent electrical isolation between the conductor and the measuring system. These sensors can be deployed in two configurations:

- **open-loop**, with a magnetic field sensor placed close to a current carrying conductor. It is simple and inexpensive, however it requires calibration and it is susceptible to interference from external magnetic fields. They can be shielded from those thanks to winding, however they still suffer from magnetic offset, which determines a constant offset voltage on the output signal, leading to lowered precision. This kind of sensor also suffers from losses due to hysteresis and eddy currents.
- **closed-loop**, which uses the output signal to compensate the magnetization inside the core of the sensor to avoid thermal drift and lower the constant offset voltage. This is achieved using a secondary winding, where a current i_s passes and it generates a magnetic field that opposes the primary one. This complex setup makes these sensors more expensive and more complicated to build. However they don't suffer losses from eddy currents or hysteresis.

There are many types of magnetic field sensors like Hall Effect sensors, Fluxgate sensors and Magneto Resistance Effect sensors.

2.3.1 Hall Effect sensors

A Hall Effect magnetic field sensor is a device that measures the magnetic field generated by a current flowing through a conductor. This effect happens when a current I flows through a thin sheet of conductive material that is penetrated by a magnetic flux density B after which a voltage V is generated perpendicular to both the current and field. This voltage is proportional to the strength of the magnetic field and, consequently, to the current passing through the conductor:

$$V_H = \frac{I \cdot B}{n \cdot q \cdot d}$$

Where:

- V_H : Hall voltage (V)
- I : Current through the conductor (A)
- B : Magnetic flux density (T)
- n : Charge carrier density (m^{-3})
- q : Charge of the carrier (C)
- d : Thickness of the conductor (m)

These kind of sensors are widely used as they are simple, cheap and provide galvanic isolation. They offer great precision in closed-loop configurations, while the precision in open-loop configurations is good at best.

Fluxgate and Magneto Resistance sensors are more complex and won't be used in this project.

2.4 Operational Amplifier

Most hall effect current sensors include an operational amplifier in the circuit. An operational amplifier is a linear electronic component used in analog circuits [8].

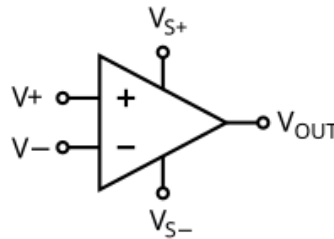


Figure 2.2. Operational Amplifier

As shown in 2.2, an operational amplifier has 5 terminals:

- V_+ , non-inverting input
- V_- , inverting input
- V_{out} , output
- V_{S+} , positive power
- V_{S-} , negative power

An operational amplifier works as a voltage amplifier by amplifying the difference in voltage between the two inputs V_+ and V_- thanks to its high gain A . The resulting amplified signal is then routed on the output terminal, according to:

$$V_{out} = A(V_+ - V_-)$$

It is important to note that the amplified output signal cannot exceed the power supply voltage, otherwise the output signal starts clipping, which means that it gets

distorted and cut off. In the configuration of figure 2.2 the amplifier is operating in *open-loop mode*, which means that since the gain is high, even a small difference between the two input voltages can saturate the output. Due to this behaviour, most of the time an operational amplifier is used in *closed-loop mode*, like this:

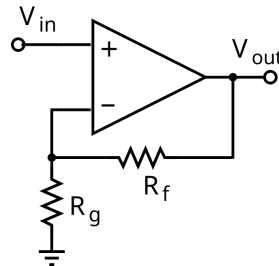


Figure 2.3. Closed loop operational amplifier

In closed-loop mode the output is either added or subtracted to one of the inputs through a feedback loop. In the case of figure 2.3, the feedback loop is on the negative terminal, so the output signal is subtracted to the input signal. This way the output saturation typical of the open-loop configuration is avoided at the price of a moderate loss of gain. There are also other benefits to negative feedback, like reducing the distortion of the signal and widening the bandwidth.

There is another kind of feedback called positive feedback, however it is outside the scope of this project.

2.4.1 Clipping and saturation

Clipping and saturation occur when the output signal of an operational amplifier exceeds its supply voltage limits.

Clipping refers to the phenomenon where the output signal is *cut off* or *clipped* because the amplifier cannot provide a voltage beyond its supply rails. This results in a distorted signal with flattened peaks. Clipping typically occurs when the input signal is too large for the amplifier's set gain and supply voltage.

Saturation happens when the operational amplifier reaches its maximum or minimum output voltage, often near the supply voltage limits. When in saturation, the amplifier can no longer linearly amplify the input signal, leading to a constant output near the positive or negative supply rail.

Both clipping and saturation distort the output signal and can lead to significant performance issues in analog circuits. One way to avoid output signal distortion due to clipping or saturation is to use the highest possible supply voltage. A higher supply voltage increases the dynamic range, reducing the likelihood of the output reaching the supply limits, as illustrated. This provides a larger voltage window for the output signal.

However, using a higher supply voltage comes with certain trade-offs:

- **Power Consumption and Heat Dissipation:** as the supply voltage increases, power consumption rises, given by $P = VI$. This generates more heat, which

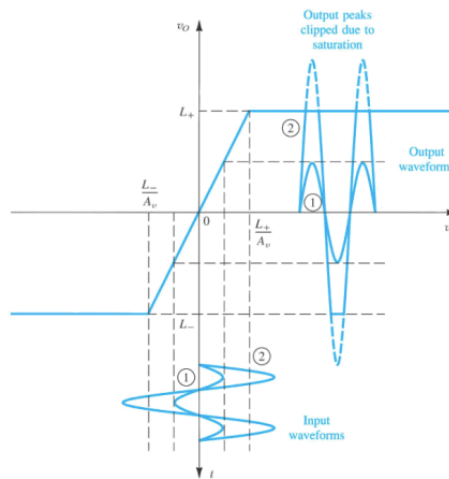


Figure 2.4. Clipping

can affect the performance and lifespan of the operational amplifier.

- **Efficiency and Power Supply Design:** higher supply voltages require more robust power supplies, which may reduce the overall system efficiency. This is particularly important in battery-powered systems, where higher voltages lead to faster battery depletion.
- **Noise and Signal Integrity:** increasing the supply voltage can also increase the noise levels in the system, which may degrade signal integrity. Careful design is needed to ensure a balance between increased dynamic range and acceptable noise levels.
- **Operational Amplifier Selection:** most operational amplifiers have a specified supply voltage range. Exceeding this range can damage the amplifier or cause it to behave unpredictably.
- **Balancing Headroom and Efficiency:** while maximizing the supply voltage helps prevent clipping, it is not always necessary to use the highest possible voltage. So it is important to choose a supply voltage that matches the expected signal range to avoid unnecessary power consumption.

In conclusion, while using a higher supply voltage reduces the risk of clipping and saturation, it is important to account for the associated power, thermal, and noise considerations. The choice of supply voltage should balance the need for dynamic range with overall system efficiency and reliability.

2.5 Data Interpretation

2.5.1 Nyquist-Shannon Sampling Theorem

The Nyquist-Shannon Sampling theorem (Shannon 1949)[9], is a fundamental principle in signal processing that helps in deciding how frequently a continuous signal needs to be sampled in order to reconstruct it digitally in an accurate way. The theorem states that:

If a function $f(t)$ contains no frequencies higher than W cps, it is completely determined by giving its ordinates at a series of points spaced $1/2W$ seconds apart.

In our project, the Hall effect sensor outputs a voltage signal proportional to the magnetic field generated by the flowing current. This means that the voltage signal will vary with time according to the current. According to the Nyquist-Shannon theorem, to accurately capture all the information in a variable signal without losing any components, the signal must be sampled at a rate that is at least twice the highest frequency present in the signal.

Since we need to sample an alternating current signal at 50Hz, a natural interpretation of Nyquist-Shannon's theorem would suggest that a sampling rate like this one would be sufficient:

$$f_{sampling} \geq 50Hz \times 2 = 100Hz$$

However, as explained in this article [10], Nyquist-Shannon's theorem provides a $f_{sampling}$ rate that represents the bare minimum to sample the signal. Therefore since we want to capture the signal without losing too much information, we need more than the bare minimum $f_{sampling}$ rate [12].

Before analyzing this signal any further we need to explain what a harmonic of a wave is. It refers to a sinusoidal wave with a frequency that is a positive integer multiple of the fundamental frequency of the wave, which in turn represents the lowest frequency also called the base frequency. In our case harmonics refers to the unwanted higher frequencies on the fundamental frequency which create a distorted wave pattern.

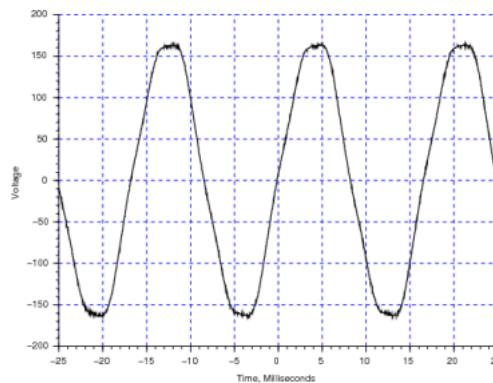


Figure 2.5. Power line voltage

The distortion can clearly be seen in the peaks of the wave. It is clear that the waveform has important content up to the 5th harmonic at 250 Hz. This means that due to Nyquist-Shannon's theorem we need to sample at:

$$f_{\text{sampling}} \geq 250\text{Hz} \times 2 = 500\text{Hz}$$

Using any sampling rate below 500 Hz will result in a loss of the signal's components. Even then, sampling up to the 5th harmonic would be barely adequate. The best way to avoid loss of information on the signal is to have a 2-times overhead on the 5th harmonic, which can be achieved with a sampling rate of:

$$f_{\text{sampling}} \geq 500\text{Hz} \times 2 = 1000\text{Hz}$$

2.5.2 Root Mean Square

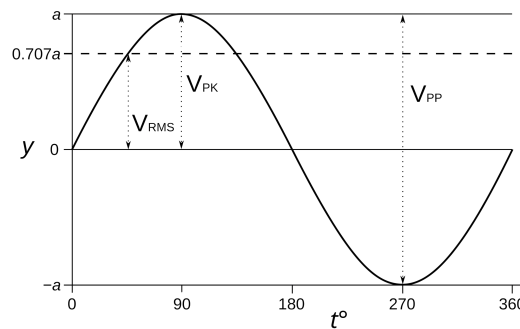
Since the input signal is a sinusoidal wave, we will need to calculate its Root Mean Square (*RMS*).

The RMS is a statistical measure of the magnitude of a varying quantity. It is especially useful in situations where the signal varies over time, such as in alternating current (AC) signals. The alternating nature of these signals means that the instantaneous current fluctuates between positive and negative peaks. This means that the average of these instantaneous current values turns out to be zero, which obviously does not provide useful information about the signal.

The Root Mean Square value of an alternating current is defined as the direct current (DC) value that dissipates the same amount of energy in a resistor that is dissipated by the actual alternating current [11]. It provides useful information on the signal's properties. Keeping in mind that V_{PK} is the peak value of the signal, the RMS value of a sinusoidal signal is equal to:

$$\text{RMS} = \frac{V_{\text{PK}}}{\sqrt{2}}$$

We use the root mean square of the signal because the peak-to-peak value gives the maximum swing of the signal but doesn't accurately represent the signal's average power or energy. Instead the RMS provides a more accurate representation of the signal's overall effect because it reduces the impact of spikes or anomalies, meaning the analysis of the signal is more reliable. The graph of a sine wave voltage shows these concepts in action:



2.5.3 Least Squares Regression

Regression analysis is a tool for studying the relationship between a dependent variable and one or more independent variables (Daniya 2020) [13]. This is mainly done for prediction purposes.

Least squares regression is a statistical method used to find the best-fitting line or curve through a set of data points. It minimizes the sum of the squared differences between the observed values (data points) and the predicted values (on the regression line). By minimizing these squared errors, the regression line is positioned as close as possible to all the points, providing an estimate of the relationship between the dependent variable (the outcome) and one or more independent variables (the predictors). While it is easy to apply and understand, it can also run into some issues because there are limitations in the shapes that linear models can assume over long ranges and poor extrapolation [14]. It is also very susceptible to nonlinear data patterns and outliers.

We used linear least squares regression, which can be used to fit the data of any function that is in the form:

$$f(\vec{x}; \vec{\beta}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots$$

The term *linear* is used, even though the function itself may not be a straight line, because if we treat the unknown parameters as variables and the explanatory variables as known coefficients associated with these parameters, the problem becomes a system of linear equations that can be solved for the unknown parameters.

2.6 Docker

Docker [15] is an open-source platform that automates the deployment, scaling, and management of applications using containerization. Containers are standalone units that package an application along with its dependencies, libraries, and configuration files, ensuring that the application runs consistently across different environments. Unlike virtual machines, containers share the host system's operating system kernel, which makes them more efficient in terms of resource usage and performance. The main advantage of using Docker is its ability to provide a consistent and reproducible environment for development, testing and deployment. Two key tools that are used to ensure this are Dockerfiles and Docker Compose.

A Dockerfile is a simple text file that contains a series of instructions on how to build a Docker image. It automates the process of assembling an image, which is a self-contained package that includes everything needed to run an application, such as the code, runtime, libraries, and environment settings. Dockerfile make it easy to reproduce the same setup across a wide range of machines.

Docker Compose is a tool for defining, building and running multi-container Docker applications. It uses a `docker-compose.yml` file to describe services, networks and volumes required by the application. It greatly simplifies the process of managing multiple containers.

There are several mechanisms to share data between containers. One of these is shared docker volumes, which are a form of persistent data storage that containers can read from and write to. The data in shared volumes is obviously independent of container lifecycles.

2.7 Instruments

The only instruments that we used in this project are a multimeter and an oscilloscope. We used the multimeter to check the connections between the components of the system. We also used its current clamp as a benchmark for the current sensor. The oscilloscope was used primarily to calibrate the sensor and make sure that it was not faulty.

Chapter 3

My Contribution

In this chapter, we describe the system's components and detail the process of assembling them.

3.1 Hardware

From a hardware standpoint, the system is composed of three parts. The most important one is the **Arduino ATmega2560**, which has a **Current Sensor Board** plugged in. The Arduino itself is connected to a **Linux machine**, which shows the data. Optionally, the current sensor can be powered by a **9V battery**.

3.1.1 System wiring with 5 Volt power supply

The sensor gets power from the 5 Volt pin of the Arduino. The analog signal coming from the sensor gets sent to port A0, where it will be read and interpreted. Only one of the two ground pins provided by the sensor is used. Here is a diagram:

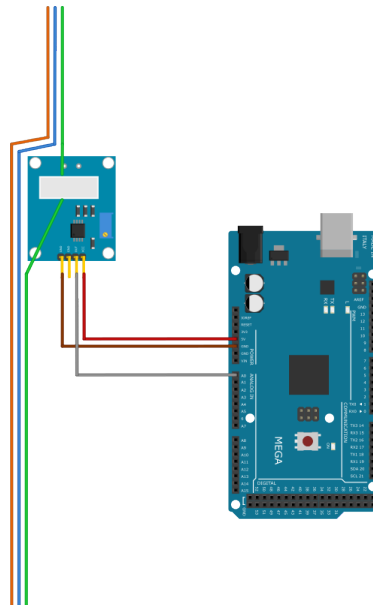


Figure 3.1. 5 Volt configuration

3.1.2 System wiring with 9 Volt power supply

In this version of the system, the sensor gets power from a 9V battery, which is grounded to the sensor ground pin. The other ground pin of the sensor is then connected to the ground pin of the Arduino. The data pin is connected to port A0 like in the 5V version.

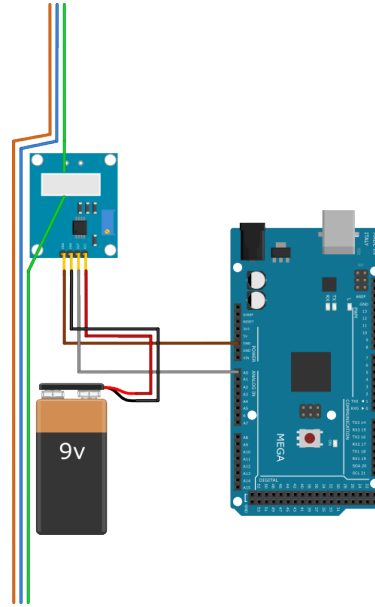


Figure 3.2. 9 Volt configuration

3.1.3 Arduino ATmega2560

The central part of the system is the Arduino ATmega2560. The Arduino board we used has an ATmega 2560 microcontroller and a good number of input/output pins that gave us a lot of flexibility in this project. We did not use the digital I/O pins, because analog pins were enough. The integrated Analog to Digital converter [2.1.4](#), has a 10-bit resolution.

The Arduino is connected through USB to a Linux-based receiver program. This program can interpret the data that the Arduino sends. The receiver is then responsible of sending the data to the web interface.

3.1.4 Current Sensor Board

The board contains various components:

- Current Sensor
- Operational Amplifier
- Potentiometer

Here is a simplified diagram of the board:

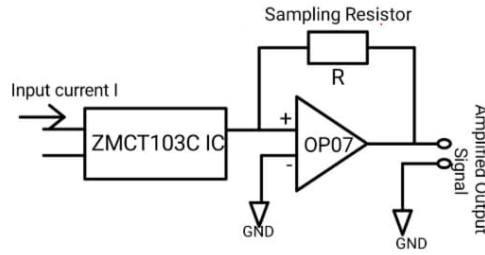


Figure 3.3. Diagram

3.1.5 Current sensor

As explained in 2.3 we will be using a Hall effect current sensor. The sensor used in this project is based on the ZMCT103C current transformer. The sensor itself is encapsulated with epoxy and according to the specifics [18], it can handle input currents between 0 and 10 A. It also has a high isolation voltage of 4500V and a low permissible error, meaning it's both accurate and safe to use.

3.1.6 Operational Amplifier

The sensor board uses a OP07 operational amplifier [19]. In this case it is being used to amplify the small voltage output from the ZMCT103C sensor for further processing by the Arduino board. It has:

- low input offset voltage, at around $75 \mu\text{V}$. It is important for maintaining accuracy when measuring low currents because it helps minimize measurement errors.
- low noise performance, to ensure that the signal remains clean.
- high stability, particularly in regards to temperature and long-term drift ($1.5 \mu\text{V}/\text{Month}$).
- wide supply and input voltages, as it supports a voltage range of $\pm 3\text{V}$ to $\pm 18\text{V}$ and an input voltage range of $\pm 14\text{V}$.

Like we explained in 2.4.1, operational amplifiers can get saturated. To improve the accuracy of our readings we also tried powering the sensor with a 9V power source, as shown in 3.1.2. We measured the current usage of a normal fan to test whether the accuracy of the system is better when using a 9V power supply over a 5V one, like in 3.1.1:

5V power supply	9V power supply	Multimeter Value
32mA	0mA	0mA
32mA	0mA	0mA
101mA	150mA	150mA
103mA	150mA	150mA
105mA	152mA	150mA
99mA	150mA	150mA

When comparing these values using a multimeter as reference, it is clear that the system is way more precise when using a 9V power supply.

3.1.7 Potentiometer

A potentiometer is a resistor with three terminals. The first and second terminals are connected to the ends of a resistive element and since our potentiometer is rotating, the third terminal (wiper) is movable. This means that it can be set to any point on the resistive element, thus changing the resistance between the wiper and the terminal. This mechanism allows us to control the gain of the operational amplifier by varying the resistance of the feedback loop. For example in our case the potentiometer replaces the R_f resistance shown in figure 2.3.

Since the gain of a non inverting operational amplifier is [8]:

$$A_V = 1 + \frac{R_f}{R_{in}}$$

And the gain of an inverting amplifier is:

$$A_V = -\frac{R_f}{R_{in}}$$

In both cases a potentiometer allows us to change the resistance dynamically. After some trial and error using an oscilloscope, we settled on a resistance value of 70k Ω for the on-board potentiometer. This value allowed us to maximise the detail of the wave while also avoiding clipping.

3.2 Software

For the software side of this project, there is a program called *meter* that runs on the Arduino and a receiver program called *receiver* which runs on the Linux host. The complete repository can be found at [16].

3.2.1 Meter - Arduino side

The program running on the Arduino is responsible for:

1. interfacing with the current sensor,
2. interpreting the raw data coming from the sensor using the analog to digital converter,
3. sending the correct current reading to the Linux side receiver.

3.2.2 Communication between meter and receiver

The communication between the Arduino and the receiver script hinges on the device file `/dev/ttyUSB0`, which serves as a serial interface.

As explained in the course material for *Sistemi Operativi*[2], everything in Linux is considered a file, including devices. Device files are located in the `/dev` directory and they provide an interface for interacting with hardware devices like hard drives and keyboards as well as virtual devices like pseudoterminals 2.1.6. Unlike regular files that are meant to store data, device files represent input/output devices, allowing user programs and the kernel to communicate with hardware through standard

file operations. This abstraction greatly simplifies the process of handling devices.

The meter and receiver exchange data in a struct called `amp_value`. Since this struct is sent in binary form, we need to use the `packed` attribute to make sure that the data is always aligned in the same way. This means that it can be easily read by the meter and the receiver without issues:

```
typedef struct __attribute__((packed)) amp_value {
    float current;
    uint16_t timestamp;
} amp_value;
```

3.2.3 Receiver - Linux side

The main tasks of the receiver program running on the Linux host are:

1. opening an interface to interact with the Arduino,
2. waiting for user input on how to read the data,
3. reading the data sent on this interface.

The receiver needs to notify the user of errors, otherwise both writing code and using the program can be complicated. For example we check for errors while opening the file descriptor `fd` for the device file:

```
int serial_open(const char* name) {
    int fd = open (name, O_RDWR | O_NOCTTY | O_SYNC );
    if (fd < 0) {
        printf ("error %d opening serial, fd %d\n", errno,
            fd);
    }
    return fd;
}
```

We also notify the user when there is an error while reading the `amp_value` struct:

```
amp_value UART_read_amp(int fd) {
    int bytes_read = 0;
    int total_bytes_read = 0;
    amp_value amp = {0, 0};

    bytes_read = read(fd, &amp, sizeof(amp_value));
    if (bytes_read == sizeof(amp_value)) {
        total_bytes_read += bytes_read;
    } else {
        perror("read");
        printf("Expected to read %lu bytes, but got %d
            bytes\n", sizeof(amp_value), bytes_read);
    }

    return amp;
}
```


3.2.4 Makefile

The Makefiles in this project are structured in a tree-like way, with a main Makefile that calls the Makefiles for the receiver and the meter:

```
.phony: clean all

all:
    make -C meter
    make -C receiver
clean:
    make -C meter clean
    make -C receiver clean
```

In turn, the Makefiles for the meter and the receiver call other Makefiles:

```
BINS=meter.elf

OBJS=my_uart.o misc.o

HEADERS=my_uart.h

include ../avr_common/avr.mk

CC = gcc
CFLAGS = -Wall -Wextra

receiver: receiver.o misc.o
    $(CC) $(CFLAGS) -o receiver receiver.o misc.o

receiver.o: receiver.c receiver.h
    $(CC) $(CFLAGS) -c receiver.c

misc.o: misc.c
    $(CC) $(CFLAGS) -c misc.c

clean:
    rm -f receiver receiver.o misc.o
```

We employed Makefiles in this project for their simplicity and ease of use, much like we showed in [2.1.5](#).

3.3 Docker

In this project, both the receiver script and web interface are docker-based, because as we explained in section [2.6](#), it allowed us to create a stable environment that can be easily reproduced anywhere.

There are two Dockerfiles, one for the receiver and one for the web interface. The receiver Dockerfile uses an Ubuntu image with the necessary packages. Then it compiles the code using make and it runs the C code for the receiver:

```
FROM ubuntu:22.04

RUN apt-get update && apt-get install -y \
    build-essential \
    avr-libc \
    gcc-avr \
```

```

    make \
    avrdude \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY . /app

RUN make

CMD ["/receiver/receiver"]

```

The Dockerfile for the web interface is much simpler, because it is built on top of a Nginx image which gets the HTML files and hosts them on port 80:

```

FROM nginx:latest

COPY ./ /usr/share/nginx/html

EXPOSE 80

```

The docker-compose file builds the receiver and webpage services based on their Dockerfiles. It also assigns them volumes for permanent storage, ports for web communication and passes through any device file needed:

```

services:
  receiver:
    build:
      context: ./linux_receiver
      dockerfile: Dockerfile
    container_name: receiver
    devices:
      - "/dev/ttyUSB0:/dev/ttyUSB0"
    volumes:
      - shared_data:/app/receiver/data
    command: ["/receiver/receiver", "/dev/ttyUSB0"]
    stdin_open: true
    tty: true

  webpage:
    build:
      context: ./webpage
      dockerfile: Dockerfile
    container_name: webpage
    ports:
      - "5000:80"
    volumes:
      - ./webpage:/usr/share/nginx/html
      - shared_data:/usr/share/nginx/html/data

volumes:
  shared_data:

```

We passed the appropriate device file (*/dev/ttyUSB0*) to the **receiver** container to allow it to interact with the Arduino board. This allows the container to receive data as if they were connected directly. We also had to force the **receiver** container to have the standard input communication channel *stdin* open, so that it could communicate with the Arduino board.

The **shared_data** volume is used to share the sensor data between the two containers. Even though there are other ways to share data between containers like Inter-Process Communication or HTTP communication, this was the easiest and in some ways most reliable approach for our project.

Chapter 4

Use cases

In this chapter, we demonstrate a complete run of the system, providing an overview of the code execution and explaining how each component functions in practice. The complete codebase of our project is on GitHub [16].

4.1 Boot up

There are two ways to boot the Linux side of the system. They both leverage Makefiles to compile the C code, however the docker way also hosts a web interface to show the data coming from the sensor.

1. terminal-based interface, which can be run using:

```
make  
./receiver /dev/ttyUSB0
```

2. docker-based interface, which can be booted up through docker-compose 3.3:

```
docker-compose up -d
```

As for the Arduino side, the code for the meter needs to be flashed to the Arduino board using Make and Avrdude:

```
make
```

4.2 Meter

When the Arduino board is powered on it automatically starts executing the Meter program.

4.2.1 Setup

The first thing it does is setup various variables for UART communication, timers and Interrupt Service Routines (ISR). We decided to use interrupts and ISRs instead of polling since interrupt-driven code is more efficient than code that relies on polling.

As we explained in 2.1.3, the code in interrupt service routines must be simple and efficient. To comply with this rule, in our code interrupt service routines are only tasked with changing the value of a counter or a flag. The flags are used to

indicate timer overflow, while counters are used to know how many measurements have been taken.

```

1 //UART Global Variables
2 volatile uint8_t mode;
3 volatile uint8_t uart_flag = 0;
4
5 //Timer Global Variables
6 volatile uint8_t online_flag = 0;
7 volatile uint8_t timer_flag = 0;
8 volatile uint16_t measurement_count = 0;
9 volatile uint8_t sensor_flag = 0;
10
11 ISR(TIMER1_COMPA_vect) { //1000hz timer for sampling
12     sensor_flag = 1;
13 }
14
15 ISR(TIMER3_COMPA_vect) { //1sec timer
16     timer_flag = 1;
17     measurement_count++;
18 }
19
20 ISR(TIMER5_COMPA_vect) { //variable timer for online mode
21     online_flag = 1;
22     measurement_count++;
23 }
24
25 ISR(USART0_RX_vect) { //interrupt when pc sends data
26     uart_flag = 1;
27     mode = UDR0; //read byte from UART representing MODE
28 }

```

Then we initialize various utilities like the UART function for communication, the analog to digital converter and the sampling timer responsible for sampling the signal:

```

1 UART_init();
2 adc_init();
3 sampling_timer_init();

```

The function to initialize the analog to digital converter sets the V_{ref} to our desired value, which as we explained in 2.1.4 is fundamental since the ADC would not work correctly without doing this:

```

1 void adc_init(void) {
2     // Select Vref=AVcc
3     ADMUX |= (1 << REFS0);
4     ADMUX &= ~(1 << REFS1);
5
6     // Set ADC prescaler to 128 for 16 MHz clock (125 kHz ADC
7     // clock)
8     ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
9
10    // Enable ADC
11    ADCSRA |= (1 << ADEN);

```

The timer is initialized with a sampling frequency of 1000Hz, for reasons we explained in 2.5.1. After this, system interrupts are enabled and we also initialize the arrays where the current measurements will be stored:

```

1 void sampling_timer_init(void){
2     TCCR1A = 0;
3     TCCR1B = (1 << WGM12) | (1 << CS11) | (1 << CS10); // set
        up timer with prescaler = 1024
4     uint16_t ocrval = (uint16_t)(15.625); //1000Hz
5     OCR1A = ocrval;
6
7     TIMSK1 |= (1 << OCIE1A); // enable timer interrupt
8 }

```

Now the program measures the current and then stores this data. Afterwards it goes into sleep while waiting either for user input coming from the receiver on the Linux side or for the next current measurement:

```

1 while (1) {
2     if(uart_flag){ // received data from linux
3         ...
4     }
5     else{ // did not receive data
6         ...
7         sleep_cpu();
8     }
9 }

```

4.2.2 Sensor data interpretation

When the Arduino does not receive data from Linux it measures the current every second. We initialize a timer that changes the value of a flag every second (line 2). Then we measure the value coming from the sensor every 1000Hz (line 7), then this value is ran through a simple algorithm to save the minimum and maximum values over a 1 second interval (lines 7-16). Then when the 1 second interval ends, those maximum and minimum values are used to compute the current (line 18) and save it in an `amp_value` struct. Then the min and max values are reset (lines 26-28):

```

1 else{ // did not receive data
2     detached_mode_timer_init();
3     float max_val = 0;
4     float min_val = 0;
5     float new_val = 0;
6     while(uart_flag == 0){ //serial not connected
7         if(sensor_flag){ //measuring every 1000hz
8             new_val = adc_read();
9             if (new_val > max_val) { //get max value
10                max_val = new_val;
11            }
12            if (new_val < min_val || min_val == 0) { //get min value
13                min_val = new_val;
14            }
15            sensor_flag = 0; //reset flag
16        }
17        if(timer_flag){
18            float current = calculate_current(min_val, max_val);
19
20            amp_value amp = {0, 0};
21            amp.current = current;
22            amp.timestamp = measurement_count;
23        }
24    }
25 }

```

```

24     update_time_arrays(amp, last_seconds, last_minutes,
25                          last_hours, last_days, last_months);
26     max_val = 0;
27     min_val = 0;
28     timer_flag = 0;
29 }
30 sleep_cpu();
31 }

```

The function used to compute the current is a relatively simple one. It uses the V_{ref} (5) and the ADC resolution of the sensor (1024) to get an averaged value that is then used to calculate the RMS of the signal. We used the RMS for reasons that were explained in 2.5.2. After that, we compute the final value using some constants that we obtained through least square regression in 4.2.3. Finally we run it through a simple check (lines 5-7) to account for the sensor's sensitivity.

```

1 float calculate_current(float min_val, float max_val){
2     float sample = ((max_val - min_val)*5)/1024;
3     sample = sample * 0.707;
4     float calibrated_sample = sample * CALIBRATION1 +
5       CALIBRATION2;
6     if (calibrated_sample < 0.03) {
7         calibrated_sample = 0;
8     }
9     return calibrated_sample;
10 }

```

4.2.3 Least Squares Regression

Since the datasheet of the sensor used in this project does not include a calibration constant, we have to compute it ourselves. It is needed to make the readings as accurate as possible. To compute the calibration constants CALIBRATION1 and CALIBRATION2, we gathered data coming from the sensor while we measured the real value using a multimeter. Those values are then stored in the numpy arrays. Then we used the linear regression model from the `sklearn` module on our data, like we explained in 2.5.3:

```

1 import numpy as np
2 from sklearn.linear_model import LinearRegression
3
4 # Sensor and real-life values
5 sensor_values = np.array([0.231, 0.255, 0.252, 0.269,
6   0.10, 0.007]).reshape(-1, 1)
7 real_values = np.array([0.11, 0.12, 0.12, 0.13, 0.00,
8   0.00])
9
10 #Create and fit the linear regression model
11 model = LinearRegression()
12 model.fit(sensor_values, real_values)
13
14 #Get the parameters of the linear model
15 CALIBRATION1 = model.coef_[0]
16 CALIBRATION2 = model.intercept_
17
18 print(CALIBRATION1, CALIBRATION2)

```

4.3 Receiver

4.3.1 Setup

When the receiver on the Linux machine starts running, it sets up a signal handler for *SIGINT*, which is a signal that is triggered when CTRL+C is pressed. We used it because there might be errors while opening the file descriptor if a user stopped the program with CTRL+C without this signal handler:

```

1 signal(SIGINT, signal_handler);
2 void signal_handler(int signum){
3     if(signum == SIGINT){
4         fprintf(stderr, "Exiting program after CTRL+C \n");
5         close(fd);
6         fd = serial_open(serial_device);
7         close(fd);
8         exit(EXIT_SUCCESS);
9     }
10 }
```

Then we get the name of the serial device file from terminal, we check it for errors, then we initialize the serial connection between meter and receiver. We set the serial to blocking, because it is more reliable in cases where the receiver or the meter need to wait for data; it is also simpler to implement. The baudrate is set to 19200, because even though higher speeds are fine they are not required for this project. Using a lower baudrate also helps improve reliability, since it means that the signal is slower so it is also more resistant to electrical noise and interference. It also results in reduced distortion and simpler error handling:

```

1 serial_device = argv[1];
2 fd = serial_open(serial_device);
3 serial_set_interface_attribs(fd, BAUDRATE, 0);
4 serial_set_blocking(fd, blocking_status);
```

Where:

```

1 int serial_open(const char* name) {
2     int fd = open (name, O_RDWR | O_NOCTTY | O_SYNC );
3     if (fd < 0) {
4         printf ("error %d opening serial, fd %d\n", errno,
5             fd);
6     }
7     return fd;
8 }
```

In the `serial_set_interface_attribs` function, we initialize a `termios` struct, then we choose the baudrate and we set it. Then we set flags to enable reading, shut off parity and specify that we use 8-bit chars. Afterwards we check for errors and we clear the input and output buffers. This is a critical step because without it the buffers may contain some data from previous connections, meaning that the serial connection will not work.

```

1 int serial_set_interface_attribs(int fd, int speed, int
2     parity) {
3     struct termios tty;
4     memset (&tty, 0, sizeof tty);
5     if (tcgetattr (fd, &tty) != 0) {
6         printf ("error %d from tcgetattr", errno);
7         return -1;
8     }
9 }
```



```

8  switch (speed){
9      ...
10 }
11 cfsetospeed (&tty, speed);
12 cfsetispeed (&tty, speed);
13 cfmakeraw(&tty);
14
15 tty.c_cflag &= ~(PARENB | PARODD);
16 tty.c_cflag |= parity;
17 tty.c_cflag = (tty.c_cflag & ~CSIZE) | CS8;
18
19 if (tcsetattr (fd, TCSANOW, &tty) != 0) {
20     ...
21 }
22
23 if (tcflush(fd, TCIOFLUSH) != 0) {
24     perror("tcflush");
25     return -1;
26 }
27
28 return 0;
29 }

```

Then the user gets asked to choose one of three possible modes:

- online mode, where the user chooses how many seconds will pass between each sample.
- query mode, where the user can see the overall statistics.
- cleaning mode, where the user deletes all the statistics.

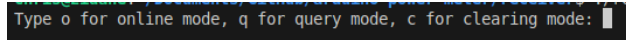


Figure 4.1. Choose mode

After the user has chosen, the receiver sends a special message to the meter containing the chosen mode. This is done using a simple write on the device file representing the Arduino. We also included a simple error check to spot if there is a problem with the UART connection:

```

1 void UART_send_special_message(int fd, char msg) {
2     ssize_t bytes_written = write(fd, &msg, sizeof(char));
3     if (bytes_written < 0) {
4         printf("Error writing to serial port\n");
5         return;
6     }
7 }

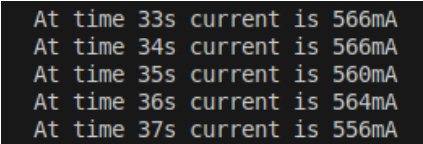
```

4.3.2 Online mode

The user gets asked for the sampling interval, which fits into a single byte since it can only go up to 60. Then the receiver waits for data coming from the meter in the form of an `amp_value` struct, reads it through a simple read on the device file and finally it gets printed on the interface:

```
1 amp_value UART_read_amp(int fd) {
2     int bytes_read = 0;
3     int total_bytes_read = 0;
4     amp_value amp = {0, 0};
5
6     bytes_read = read(fd, &amp, sizeof(amp_value));
7     if (bytes_read == sizeof(amp_value)) {
8         total_bytes_read += bytes_read;
9     }
10    else {
11        perror("read");
12        printf("Expected to read %lu bytes, but got %d\n",
13               sizeof(amp_value), bytes_read);
14    }
15    return amp;
16 }
```

The data is interpreted the same way as it was in 4.2.2, with the only difference being that the interval of time between measurements is set by the user so it is not necessarily equal to 1 second. Here is an example:



```
At time 33s current is 566mA
At time 34s current is 566mA
At time 35s current is 560mA
At time 36s current is 564mA
At time 37s current is 556mA
```

Figure 4.2. Online mode output

4.3.3 Query and Clearing mode

When the user chooses query mode, the terminal shows various statistics on the current that was measured in the last hour, last day, last month and last year. If the user chooses clearing mode, the arrays containing those statistics are deleted:

```

Last Minute:
0mA | 430mA | 438mA | 430mA | 408mA | 438mA |
428mA | 426mA | 428mA | 420mA | 432mA | 511mA |
420mA | 426mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |

-----

Last Hour:
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |

-----

Last Day:
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |

-----

Last Month:
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |

-----

Last Year:
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |

```

Figure 4.3. Query mode output

```

Type o for online mode, q for query mode, c for clearing mode: c
Are you sure you want to clear the array? (Y/n): Y
Memory cleared
Type o for online mode, q for query mode, c for clearing mode: q
Last Minute:
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |

-----

Last Hour:
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |

-----

Last Day:
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |

-----

Last Month:
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |

-----

Last Year:
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |
0mA | 0mA | 0mA | 0mA | 0mA | 0mA |

```

Figure 4.4. Clearing mode output

4.4 Docker and Website

There is a web-based way to check out the data coming from the Arduino board through a web interface, available at `localhost:5000` :

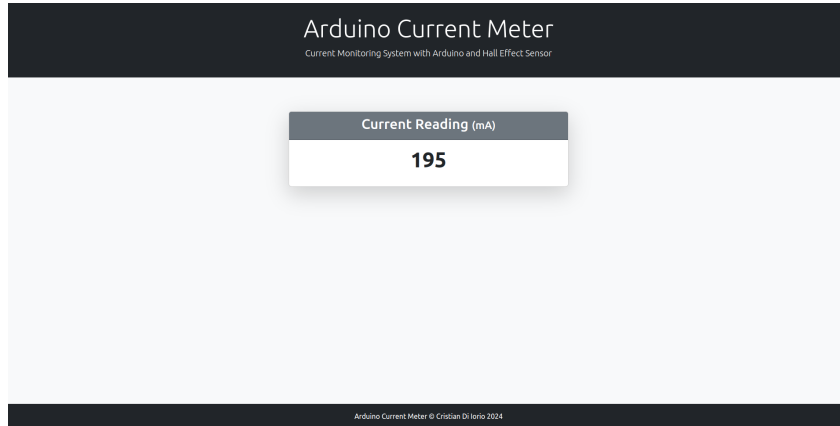


Figure 4.5. Website interface

The website has a simple interface. It uses a JavaScript function to read the file where the measurements are written. This file (`data.txt`) resides in a shared volume called `shared_data` 2.6. The volume is accessible to both the receiver and webpage containers, which grants both of them read/write privileges on its contents.

```
1 async function fetchLastValue() {
2   try {
3     const response = await fetch('/data/data.txt');
4     const data = await response.text();
5
6     // Split by newlines and get the last non-empty line
7     const lines = data.trim().split('\n');
8     const lastValue = lines[lines.length - 1];
9
10    // Display the last value
11    document.getElementById('data-container').textContent =
      lastValue;
12  } catch (error) {
13    console.error('Error fetching the last value:', error);
14  }
15 }
16
17 // Update the last value every second
18 setInterval(fetchLastValue, 1000);
```

Chapter 5

Conclusions

In this project, we set out to develop a low-cost, easy-to-build current monitoring system using an Arduino microcontroller, a Hall effect current sensor, and a Linux computer. Our goal was to provide a solution that is adaptable, efficient and accessible for various applications. Through the integration of these components, we successfully demonstrated a system capable of precise and real-time current measurement.

By leveraging the Hall effect sensor, we showed how non-intrusive and accurate current monitoring can be achieved without the complexity and cost associated with traditional methods. The use of the Arduino provided a flexible and user-friendly platform for handling sensor data and transmitting it to the Linux computer. The Linux computer, in turn, served as an essential tool for programming the Arduino, debugging the system, and displaying the current readings in real time. Docker is another helpful tool that played a vital role in this project.

Each step of the design and implementation was thoroughly detailed, fulfilling our commitment to explaining both the *how* and the *why* behind the system's components and operation. We also highlighted the underlying principles from physics, electronics, and signal theory that influenced our choices.

In conclusion, we have successfully built a reliable and straightforward current monitoring system that fulfills the objectives outlined in the introduction. Our system offers a practical, low-cost alternative for current measurement that can be easily replicated and adapted for a wide range of applications, ensuring both safety and efficiency in electrical monitoring.

Bibliography

- [1] GCC Team, "*AVR Libc Documentation*", 2024. [AVR-GCC Documentation](#)
- [2] Giorgio Grisetti, "*Slide del corso di Sistemi Operativi*", 2024. [GitLab](#)
- [3] Kamran Jalilinia, "*Analog to Digital Conversion - Sampling and quantization*", 2024. [Article](#)
- [4] GNU Operating System, "*GNU Make Manual*", 2024. [GNU manual](#)
- [5] Ubuntu Documentation Team, "*pty(7), pseudoterminal interfaces*", 2023. [man\(7\)](#)
- [6] Marco Crescentini, Sana Fatima Syeda and Gian Piero Gibiino, "*Hall-Effect Current Sensors: Principles of Operation and Implementation Techniques*", IEEE Sensors Journal, Vol. 22, No. 11, June 1, 2022. DOI: [10.1109/JSEN.2022.3119766](#)
- [7] Silvio Ziegler, Robert C. Woodward, Herbert H.C. Iu and Lawrence J. Borle, "*Current Sensing Techniques: A Review*", IEEE Sensors Journal, Vol. 9, No. 4, pp. 354–376, Apr. 2009. DOI: [10.1109/JSEN.2009.2013914](#)
- [8] Adel Sedra, Kenneth Smith, "*Circuiti per la Microelettronica*", EdiSES edizioni, 1996.
- [9] Claude Shannon, "*Communication in the Presence of Noise*", Proceedings of the IRE, Vol. 37, No. 1, January 1949. DOI: [10.1109/JRPROC.1949.232969](#)
- [10] Tim Wescott, "*Sampling: What Nyquist Didn't Say, and What to Do About It*", 2018. [Article](#)
- [11] Raymond Serway, John Jewett, "*Physics for Scientists and Engineers*", Cengage Learning. [Chapter 21](#)
- [12] Jim Karki, "*Understanding Operational Amplifier Specifications*", [Report](#)
- [13] T. Daniya, Dr. Kumar, and R. Cristin, "*Least Square Estimation of Parameters for Linear Regression*", International Journal of Control and Automation, vol. 13, pp. 447–452, Apr. 2020. [Paper](#)
- [14] National Institute of Standards and Technology, "*Handbook of Statistical Methods*", 2024. [Handbook](#)
- [15] Docker Documentation Team, "*Docker Documentation*", 2024. [Documentation](#)
- [16] Arduino Current Meter [Project Repository](#) on GitHub
- [17] Ubuntu Documentation Team, "*ioctl(2), control device*", 2024. [man\(2\)](#)

-
- [18] Qingxian Zeming Langxi Electronic, "*ZMCT103C Current Transformer*", 2024. [Manufacturer's spreadsheet](#)
- [19] Analog Devices, "*Ultralow Offset Voltage Operational Amplifier*", 2024. [Manufacturer's spreadsheet](#)

Acknowledgements

Dedico questa tesi a tutti quelli che mi hanno sostenuto durante questi tre anni. A mia sorella Virginia, mia madre Serena e mio padre Pietro. E anche a tutti i miei amici.