Universitat Politècnica de Cataluña

Course: Power electronics
Students: Guillen Mañe Fernandez
Raul Luque
Miguel Sánchez
Date: November 6, 2025

Entrega 1

The report is structured as follows: first, the implementation of the current measurement system is described, including the main software routines developed and explained by the professors during the laboratory session for signal acquisition, voltage-to-current conversion, and RMS calculation. Then, the collected measurements are presented, followed by a brief analysis and discussion of their accuracy and validity. Finally, a preliminary project proposal is outlined, in which the concepts introduced throughout the course are applied to the design of an energy-aware heating control system.

Contents

List of Figures

1 Load Measurement-Implemented code

Arduino code in C

```
#include <Wire.h>
#define ADS1115_ADDRESS 0x48
//==========================================================================
// VARIABLE DECLARATIONS
//==========================================================================
// Define the pins of the Arduino ADC where the current sensor
//is measured
const int SensorPin = A1, RefPin = A2;
// Define the data from the current sensor
const int Rshunt = 33.3;
// Resistance of the transformer: Model 50 A: 20 ohms,
//Model 30 A: 33.3 ohms
double n_trafo = 1000;
//Number of turns between primary and secondary
// Variables to calculate every millisecond
unsigned long time_now = 0;
unsigned long time_ant = 0,
difTime = 0, act_time = 0,
reading_time = 0, dif_reading_time = 0, timer1 = 0, timer2 = 0;
// Define variables to calculate the RMS of a power cycle
double quadratic_sum_v = 0.0;
double quadratic_sum_rms = 0.0;
// This variable accumulates the quadratic sum of
//instantaneous currents
const int sampleDuration = 20;
// Number of samples that determine how often the RMS
//is calculated
int quadratic_sum_counter = 0;
// Counter of how many times values have been accumulated
//in the quadratic sum
double freq = 50.0;
// Define the frequency of the power cycle
// Define variables to calculate an average of the current
double accumulated_current = 0.0;
// Accumulator of RMS values for averaging
const int sampleAverage = 250;
// Number of samples that determine how often the RMS average
//is calculated
int accumulated_counter = 0;
// Counter of how many times RMS values have been accumulated
bool first_run = true;
double v_calib_acum = 0;
double v_calib = 0;
int i = 0;
byte writeBuf[3];
```

Arduino code in C

```
//==========================================================================
// Helper functions: Function created to partition the problem
//in smaller parts
//==========================================================================
   void config_i2c(){Wire.begin(); // begin I2C
   // ASD1115// set config register and start conversion
   // ANC1 and GND, 4.096v, 128s/
   writeBuf[0] = 1;
   // config register is 1
   writeBuf[1] = 0b11010010;
   // 0xC2 single shot off <== ORIGINAL - single conversion/
   //AIN1 & GND/ 4.096V/ Continuous (0)
   // bit 15 flag bit for single shot
   // Bits 14-12 input selection:
   // 100 ANC0; 101 ANC1; 110 ANC2; 111 ANC3
   // Bits 11-9 Amp gain. Default to 010 here 001 P19
   // Bit 8 Operational mode of the ADS1115.
   // 0 : Continuous conversion mode
   // 1 : Power-down single-shot mode (default)
   writeBuf[2] = 0b11100101; // bits 7-0  0x85 //869 SPS
   // Bits 7-5 data rate default to 100 for 128SPS
   // Bits 4-0  comparator functions see spec sheet.
   // setup ADS1115
   Wire.beginTransmission(ADS1115_ADDRESS);  // ADC
   Wire.write(writeBuf[0]);
   Wire.write(writeBuf[1]);
   Wire.write(writeBuf[2]);
   Wire.endTransmission();
   delay(500);}
   float read_voltage(){//unsigned long start = micros();
   // Read conversion register
   Wire.beginTransmission(ADS1115_ADDRESS);
   Wire.write(0x00); // Conversion register
   Wire.endTransmission();
   Wire.requestFrom(ADS1115_ADDRESS, 2);
   int16_t result = Wire.read() << 8 | Wire.read();
   // Mount the 2 byte value
   Wire.endTransmission();
   //unsigned long end = micros();
   //Serial.print("ADC␣Read␣Time␣(us):␣");
   //Serial.println(end - start);
   // Convert result to voltage
   float voltage = result * 4.096 / 32768.0;
   // Raw adc * reference voltage configured /
   return voltage;  // Voltage in V}
```

Arduino code in C

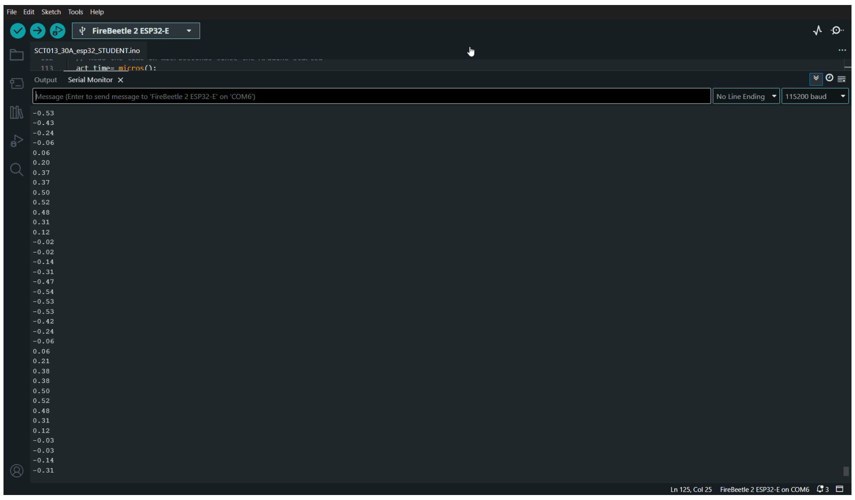
```
//=============================================================
// setup Function: Function that runs once on startup
//=============================================================
void setup() {
  // Initialize serial communications
  Serial.begin(115200);
  // Initialize IIC communications
  config_i2c();}
```

Arduino code in C

```
//=====================================================================
// loop Function: Function that runs cyclically indefinitely
//=====================================================================
void loop() {
  // Read the time in microseconds since the Arduino started
  act_time= micros();
  // Calculate the time difference between the current time
  //and the last time the instantaneous current was updated
  difTime=act_time-time_ant;
   if (difTime>=1000) {
    // Update the time record with the current time
    time_ant=act_time;
    // Read the voltage from the sensor check the 1.65
    //measuring in the circuit with AO2
    double Vinst = read_voltage() - 1.65;
    Serial.println(Vinst);
    // Convert voltage in shunt to current measurement
     double Iinst=Vinst*30; // Accumulate cuadratic sum
    quadratic_sum_rms += Iinst*Iinst*difTime/10000000;
    quadratic_sum_counter++;}
  // EVERY POWER CYCLE (20 ACCUMULATED VALUES), CALCULATE RMS
if (quadratic_sum_counter>=20) {
    // Take the square root to calculate the RMS of the
    // last power cycle
    double Irms = sqrt(50*quadratic_sum_rms);
    // Reset accumulation values to calculate the RMS of
    // the last power cycle
    quadratic_sum_counter=0, quadratic_sum_rms=0;
    // Filter base error
    if (Irms<=0.1){
      Irms = 0;}// Accumulate RMS current values to calculate
    // the average RMS
    accumulated_current+=Irms, accumulated_counter++;}
  // EVERY 250 POWER CYCLES (approximately 5 seconds),
  //CALCULATE THE AVERAGE RMS
  if (accumulated_counter >=250) {
    // Calculate the average of the RMS current
double Irms_flt=accumulated_current/((double)accumulated_counter);
    // Reset accumulation values to calculate the average RMS
    Irms_flt=Irms_flt;// Print the filtered current
    //Serial.println(Irms_flt);
    }}
```

1.1 Output

Throughout the experiment, the execution of the program was monitored at several key points using the Arduino Serial Monitor. Intermediate variables, such as instantaneous voltage and calculated RMS current, were printed in rea


File Edit Sketch Tools Help
FireBeetle 2 ESP32-E
SCT013_30A_esp32_STUDENT.ino
Output
Serial Monitor
No Line Ending
115200 baud
Ln 125, Col 25 FireBeetle 2 ESP32-E on COM6

```
act_time= micros();
```

```
-0.53
-0.43
-0.24
-0.06
0.06
0.20
0.37
0.37
0.50
0.52
0.48
0.31
0.12
-0.02
-0.02
-0.14
-0.31
-0.47
-0.54
-0.53
-0.53
-0.42
-0.24
-0.06
0.06
0.21
0.38
0.38
0.50
0.52
0.48
0.31
0.12
-0.03
-0.03
-0.14
-0.31
```

Figure 1: Cheking of every step of the code

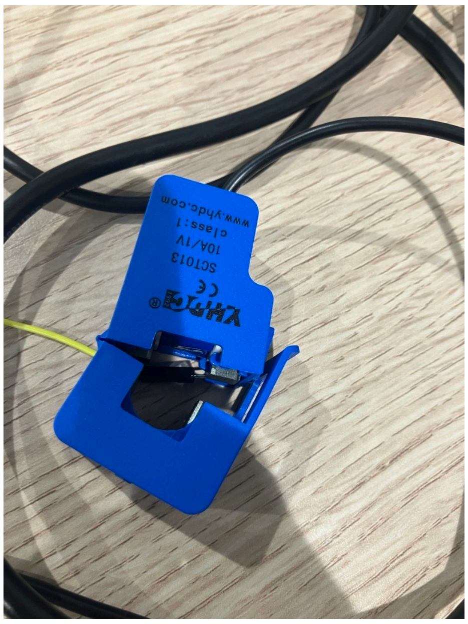


Figure 2: Sensor used

1.2 Discussion

The results obtained during the session demonstrate that the developed algorithm successfully measures AC current and provides a stable RMS value after averaging.

The RMS estimation was based on a quadratic accumulation approach over one electrical cycle (20 samples), followed by long-term averaging over 250 cycles. This strategy reduced noise and transient fluctuations, resulting in consistent measurements. Although small offsets were observed in the raw sensor voltage, the filtering stage mitigated this issue

Overall, the practice validated the proposed measurement architecture as a suitable approach for basic current monitoring, with the potential to be extended toward the final project presented in the next section

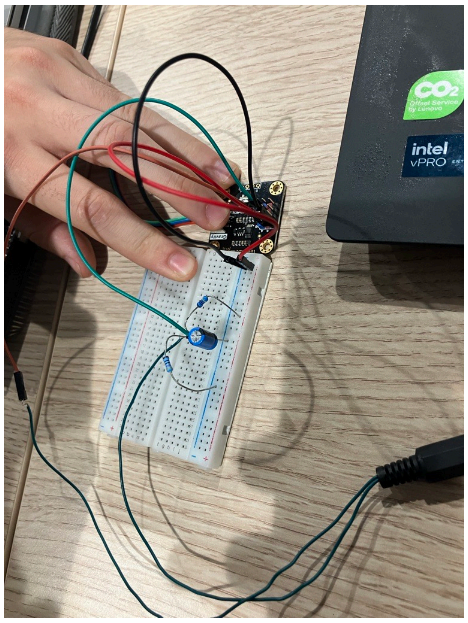


Figure 3: Circuit used

2 Project

2.1 Project description

The objective of the project is to automatically optimize the energy use of a heating load based on its operating environment and the CO_2 emissions associated with the peninsular electrical system. The load consists of a resistive heating system that warms a room using hot air. The software automatically measures the room temperature to determine whether heating is critical, slightly below the target, or unnecessary. Additionally, the system measures the power consumption and considers the daily CO_2 emission levels of the electrical grid, so that when emissions are high and the temperature is not critical, the heating load remains off.

2.2 Hardware Implementation

The hardware architecture is designed to measure the electrical consumption of the heating system, monitor the ambient temperature, obtain CO_2 emission data from the cloud, and control the power supply of the load accordingly. The system consists of the following main components:

- **ESP32 microcontroller**: Acts as the central processing unit. It acquires the temperature and current sensor data, performs local decision-making, and establishes Wi-Fi communication to request CO_2 emission data from online services using HTTP. Its integrated ADC is used for current measurement, while GPIO pins control the relay.

- **Hall-effect current sensor**: Installed around the phase conductor of the heating load. It provides an analog voltage proportional to the instantaneous current. The signal is read by the ADC of the ESP32 to compute the electrical power and energy consumption.

- **Temperature sensor**: Measures the ambient temperature of the room. It communicates analoglly with the ESP32 and allows precise measurement to compare with the temperature setpoint.

- **Power relay**: An electromechanical relay capable of switching the mains voltage and current of the heating system. The relay is driven through a transistor or driver circuit from an ESP32 GPIO pin. It connects or disconnects the phase conductor of the heater according to the control logic.

- **Power supply and wiring**: A 5 V isolated power supply provides energy to the ESP32 and sensors. The heating load is connected through a Schuko socket or terminal block, including phase, neutral, and protective earth. A fuse is placed in series with the phase conductor to protect against overcurrent. All equipment is mounted in a protective enclosure to ensure electrical safety and physical separation between low-voltage electronics and mains wiring.

2.3 Software Implementation

The objective of the software is to control the relay contacts to energize or de-energize the load based on CO_2 conditions and temperature thresholds. A main task will continuously read the sensors and control the relay. A separate task will periodically perform HTTP requests, parse the data, and store the result in a shared variable.

2.3.1 Relay Activation Conditions

- Energize the circuit whenever the temperature is below T_{min}.
- Energize the circuit when the temperature is between T_{min} and T_{target}, provided that the CO_2 emissions (tCO_2/MWh) are below a defined limit.
- De-energize the circuit whenever the temperature exceeds T_{target}.

Additionally, implementing remote activation/deactivation of the load and a notification system through a Telegram bot is considered an optional objective. Whether these features are included will depend on the available development time. If implemented, they will follow the same design philosophy as the CO_2 acquisition task, meaning they will run in a separate task to avoid blocking the main control loop.

Similarly, an optional weekly activity report may be generated, summarizing maximum power,average daily energy consumption, weekly and accumulated CO_2, and the number of relay switching events. If this reporting feature is implemented, the system will store the required accumulated values in non-volatile memory to preserve data between reboots.

The firmware will be implemented in C++ using the Arduino framework on the ESP32 platform.

3 Project Planning, Communication and Roles

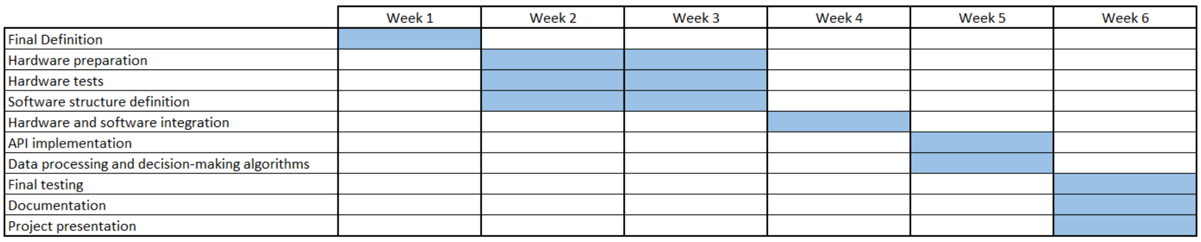
3.1 Implementation Plan

The estimated duration of the project is six weeks. The schedule is structured to allow parallel development of hardware and software, followed by integration and validation. The revised implementation plan is as follows:

- **Week 1:** Definition of system requirements, review of available components, confirmation of hardware/software architecture, and distribution of roles.
- **Week 2:**
  - Hardware setup: wiring of current and temperature sensors, relay integration, ESP32 power supply and connections.
  - Software foundation: creation of basic code structure for sensor acquisition and relay control logic.

- **Week 3:**
  - Initial hardware testing: validation of sensor readings and correct relay activation.
  - Implementation of internal decision logic based on temperature thresholds and heating control.
- **Week 4:**
  - Integration of cloud CO_2 data: API communication, parsing and storing emission levels.
  - Merging environmental logic with CO_2-based optimisation for heating control.
- **Week 5:** Final validation, optimisation of control behaviour, safety checks, documentation and preparation of the final presentation.

The gantt plan is as follows:



	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6
Final Definition	🟦					
Hardware preparation		🟦	🟦			
Hardware tests		🟦	🟦			
Software structure definition		🟦	🟦			
Hardware and software integration				🟦		
API implementation					🟦	
Data processing and decision-making algorithms					🟦	
Final testing						🟦
Documentation						🟦
Project presentation						🟦

Figure 4: project planing

The team will meet weekly during class sessions, where progress will be reviewed, responsibilities reassigned if necessary, and questions discussed with professors. Continuous communication will also be maintained through email and WhatsApp for coordination and problem solving.

GitHub will be used for code management. There will be two main directories: one for documentation and another for code, regardless of whether the code is in C or Python. Additional directories may be created as needed. Parallel implementation branches will be used, and the repository leader will coordinate them to ensure consistent integration and prevent merge conflicts.

GitHub repository can be found here: Project Repository

3.2 Team Roles

- **Miguel Sánchez Team Leader**
  Miguel will act as the project leader, coordinating both hardware and software development. Due to his broader technical background and experience, he will oversee the global workflow, ensure deadlines are met, and support decision-making. In addition, he will contribute to programming tasks, assisting both with the sensor-control implementation and the API/data processing to maintain consistency between all modules.
- **Raúl Luque Hardware Lead and Sensor Integration**
  Raúl is responsible for the hardware implementation of the system. This includes the integration of sensors (temperature and current), relay module, ESP32 connections, wiring, and power supply. He will also program the parts of the software related to sensor data acquisition and signal conditioning, ensuring that the hardware readings are correctly transferred into the control logic.

- **Guillem Mañé Data and Software Integration**
  Guillem will focus on the software side related to external data acquisition. His main task is to implement the API communication to retrieve real-time CO_2 emission data, process it, analyse trends, and make it available to the decision-making algorithm. He will work closely with the rest of the team to ensure proper integration of cloud data with the control logic of the ESP32.

Although each member has a specialised role, all three will collaborate in software development to guarantee proper integration between sensor data, API information, and rela