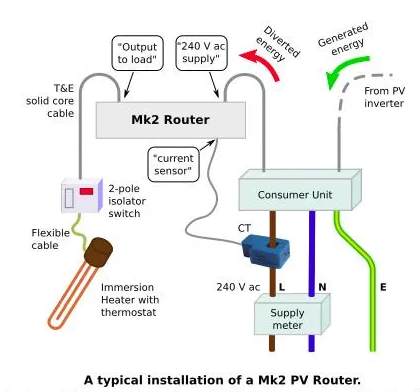
**Mk2PVRouter**

J. Lemaire

Pierrefeu February 2022

# Objective

This document summerizes the description, the assembly and the test for a simple version of the photovoltaic router, **Mk2PVRouter**, created and distributed (kit) by **Robin Emley**, to divert the PV surplus power toward a resistive load, on displaying the total diverted energy (kWh).



Version :

* Main PCB 2.1a, 3.3V powered
* 230V Monophased current
* 1 load
* Display module
* No RF module and no pin saving hardware
* French context => faster control with normal mode only

Documentation :

* Mk2PVRouter[[1]](#footnote-2)
* PV diversion[[2]](#footnote-3)
* Videos Le Profes’Solaire (in french) :
  + 5 - Routeur solaire autoconsommation[[3]](#footnote-4)
  + 18 - Routeur photovoltaïque (partie 2)[[4]](#footnote-5) [[5]](#footnote-6)
  + 21 - Tuto montage Mk2PVRouter[[6]](#footnote-7)
* Le Profes’Solaire : câblage avec horloge ou contacteur heures creuses[[7]](#footnote-8)

The documentation provided by Robin Emley is very complete and instructive, but sometimes not easy (for me !) to use because it deals with any other versions : so I opted to restrict this document to my context and to deal with the essential informations for that version  (schematics, references of the main components, crucial tests, etc.) on adapting the Robin Emley’s documents.

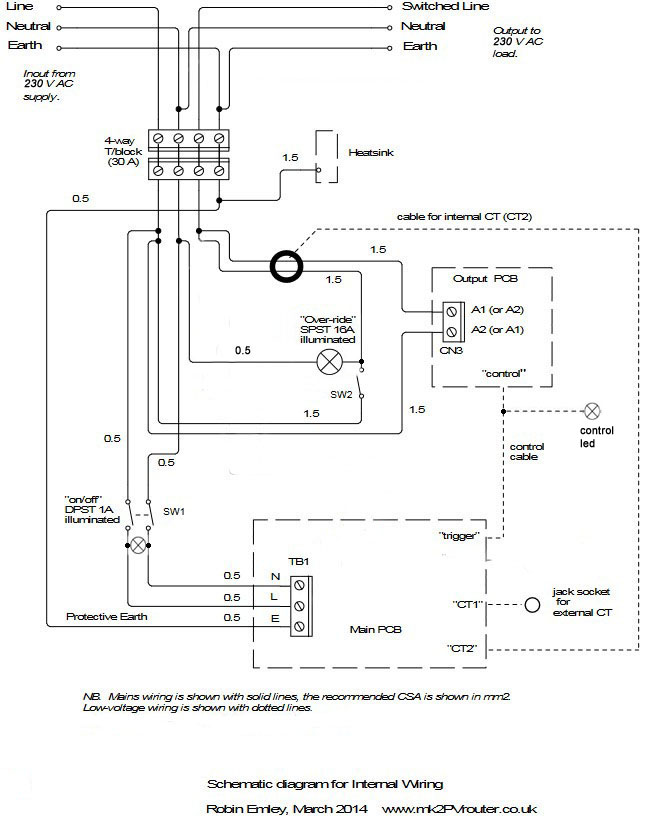
But I also tried to explain simply but precisely « how the router works » and to structure its main Arduino sketch, using OOP classes.



# Making the router

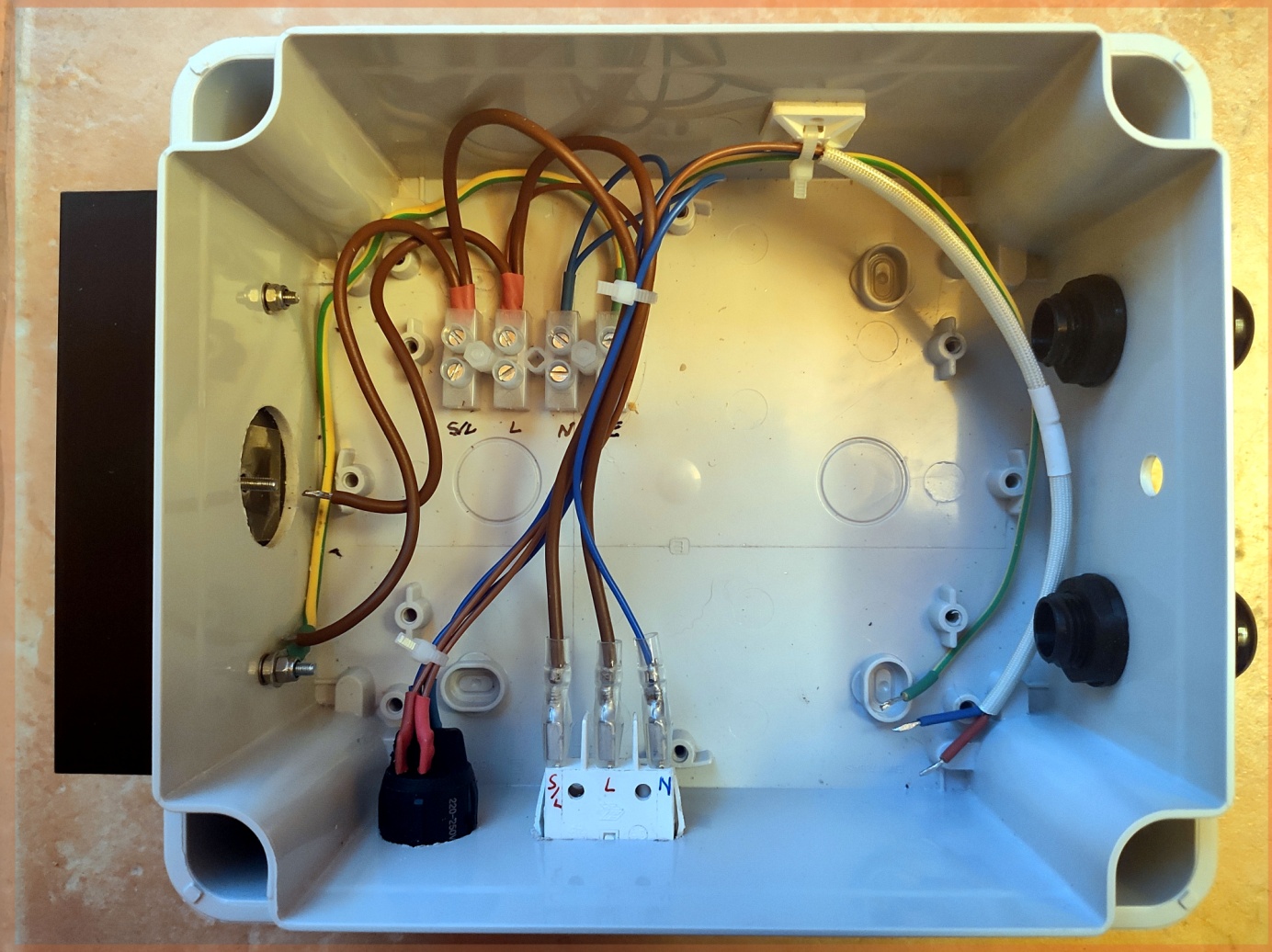
## 230V part

### Schematic[[8]](#footnote-9)



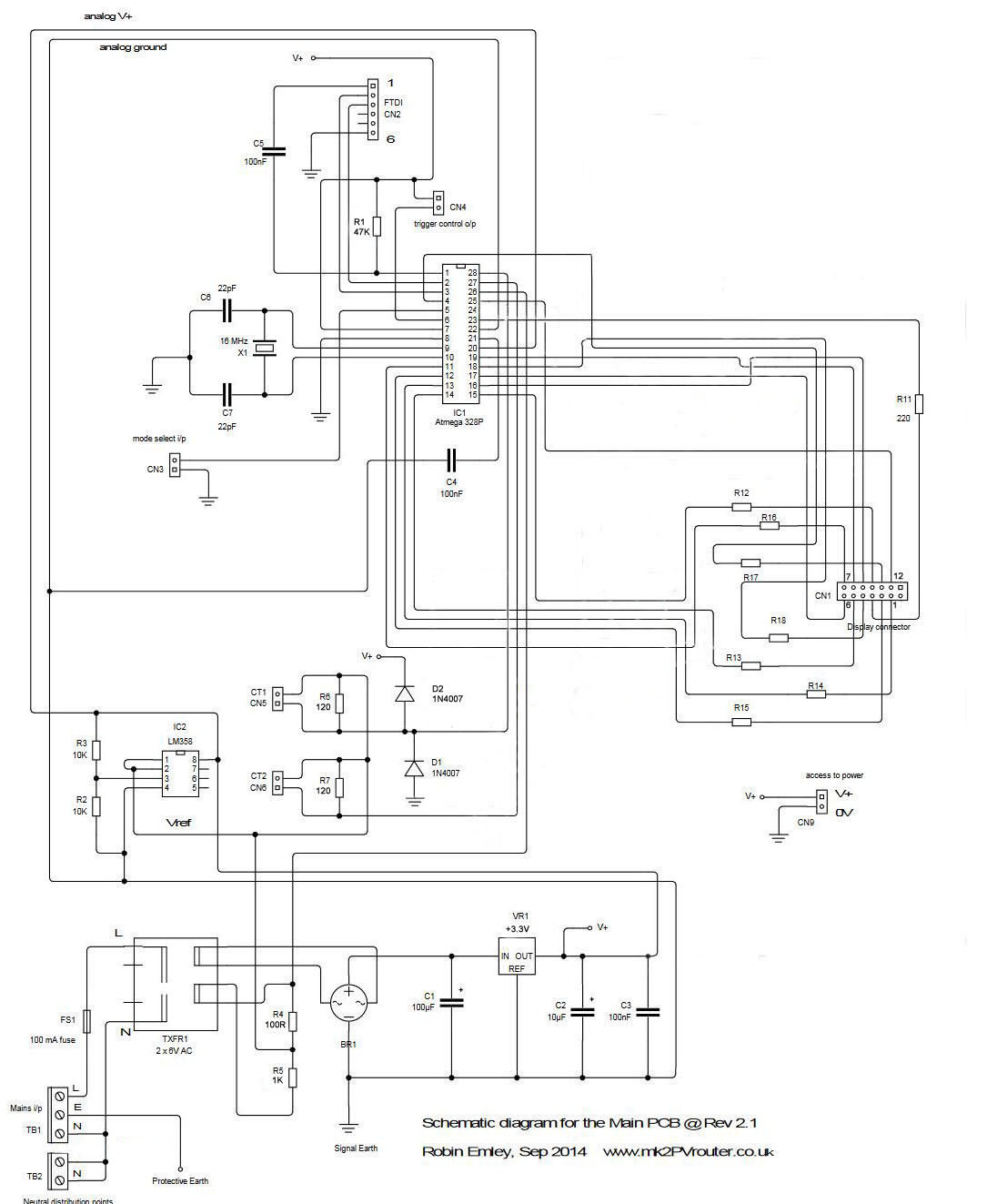
### Assembly

Here the heatsink is located at the top of the box.

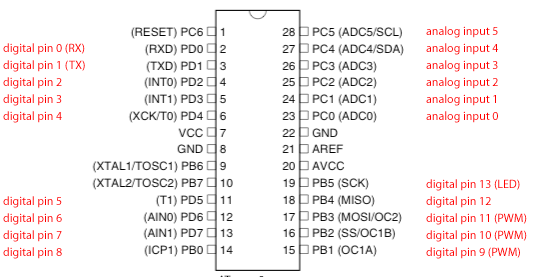


## Main PCB

### Schematic[[9]](#footnote-10)

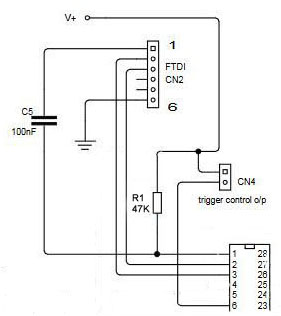


The « **brain part** » is a 28 pins microcontroller **ATmega328P**[[10]](#footnote-11); it can be programmed with Arduino sketches, using the digital[[11]](#footnote-12) or analog numbering for that pins :

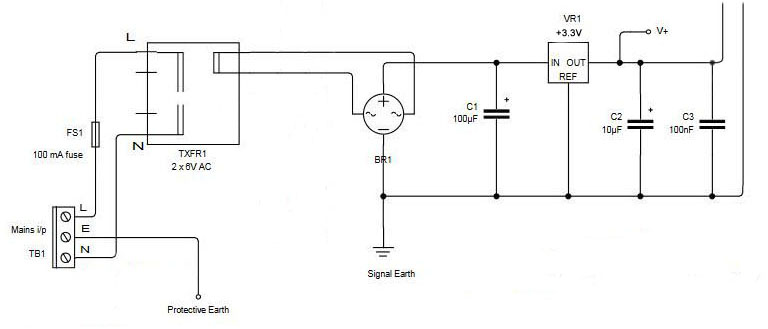


|  |  |  |  |
| --- | --- | --- | --- |
| # ATmega 328P pin | Function | To | # pin |
| 1 | Reset | FTDI | 1 |
| 2 | RX | FTDI | 2 |
| 3 | TX | FTDI | 3 |
| 4 | Display | CN1 | 11 |
| 5 | Mode[[12]](#footnote-13) | CN3 | 1 |
| 6 | Trigger | CN4 | 2 |
| 7 | VCC | V+ (3,3V) |  |
| 8 | GND | GND |  |
| 9 | XTAL 16MHz | X1 | 1 |
| 10 | XTAL 16MHz | X1 | 2 |
| 11 | Display | CN1 | 7 |
| 12 | Display | CN1 | 2 |
| 13 | Display | CN1 | 1 |
| 14 | Display | CN1 | 5 |
| 15 | Display | CN1 | 10 |
| 16 | Display | CN1 | 9 |
| 17 | Display | CN1 | 6 |
| 18 | Display | CN1 | 4 |
| 19 | Display | CN1 | 8 |
| 20 | AVCC | V+ (3,3V) |  |
| 21 | AREF | C4 => GND |  |
| 22 | GND | GND |  |
| 23 | Display | CN1 | 3 |
| 24 | NC |  |  |
| 25 | Display | CN1 | 12 |
| 26 | Grid voltage | TXFR1 | Coil 1 |
| 27 | Diverted current | CT2 | 2 |
| 28 | Grid current | CT1 | 2 |

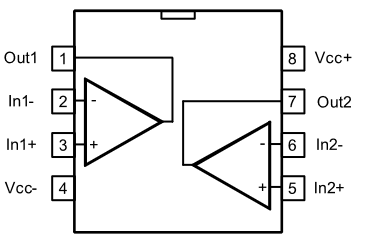
The microcontroller can be programmed using an **FTDI** 6-pins connector (CN2) where only the pins #1 (DTR), 2 (TXD), 3 (RXD) and 6 (GND) are cabled and the **trigger** connector (CN4) has its negative pin connected to the pin #6 of the ATmega328P[[13]](#footnote-14) :



The **power part** uses the 1st secondary coil of a 240V => 6V transformer (TXFR1), a bridge rectifier **W005G**[[14]](#footnote-15) (BR1), and an unique 3.3V linear regulator **Ua78m33C**[[15]](#footnote-16) (VR1), with 3 capacitors :



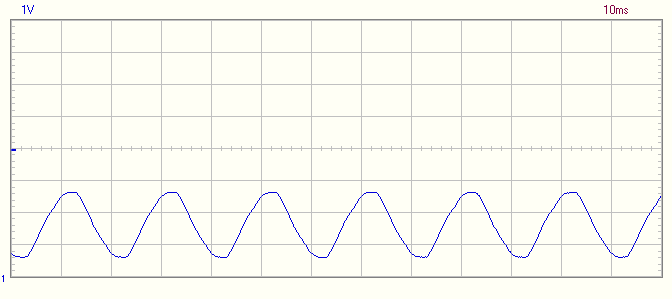
The **current (grid and diverted) measures** are performed by the analog input #28 (A5) and #27 (A4) with an OP ampli of the **LM358** integrated circuit[[16]](#footnote-17) :



used here as an **VRef** ≅ 1.65V voltage follower[[17]](#footnote-18), 2 x current transformer 100A => 50mA (CT1, CT2) and 2 x 120Ω burden resistors :

The **voltage** (grid) measure is performed by the analog input #26 (A3) and the 2nd 6V coil of the voltage transformer 230V[[18]](#footnote-19) => 2 x 6V, with a resistor divider 1kΩ-100Ω in which the **VRef** voltage is introduced again :

This signal can be traced using an oscilloscope :



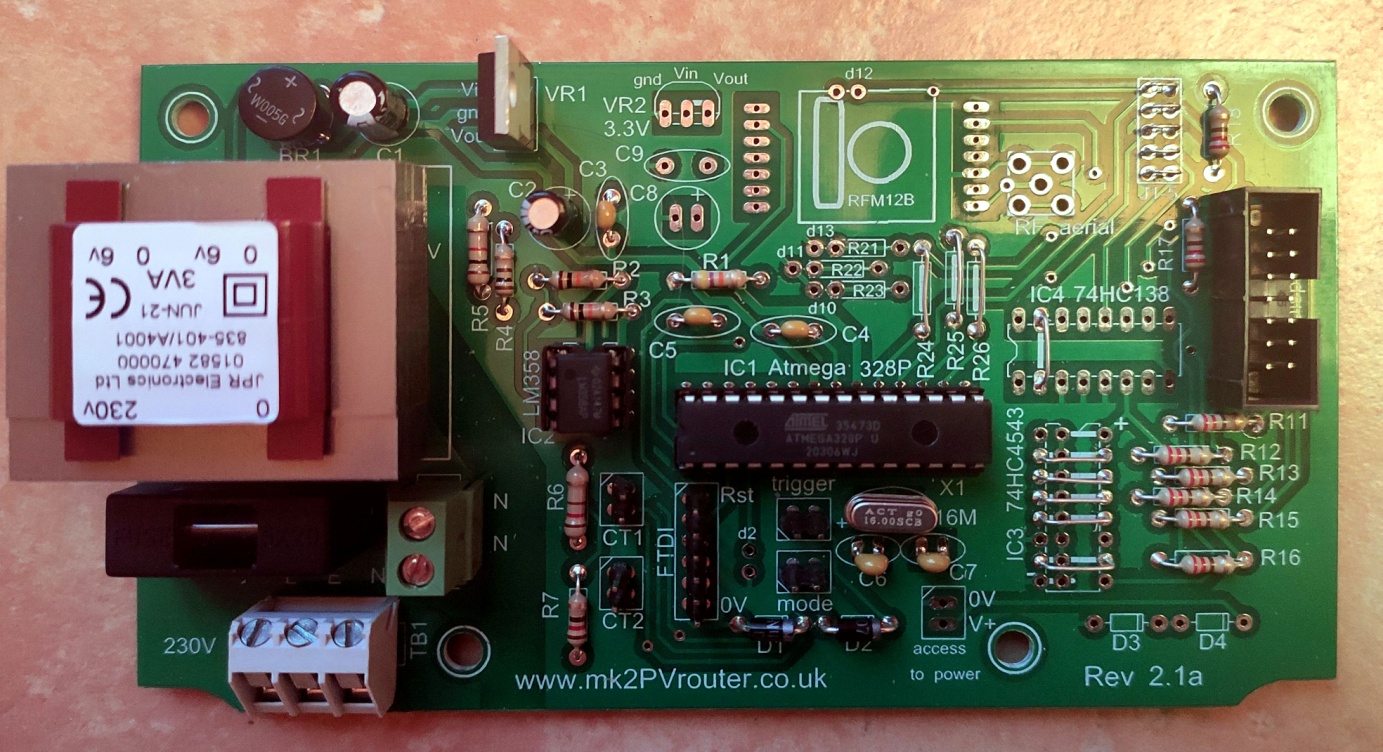
The Falstad[[19]](#footnote-20) simulator is also a very simple tool to understand this analog part :



(We observe here a little dephasis between and , caused by the transformer).

Most of the other pins are used to **display** the diverted energy (as digital pins) : we shall precise it latter.

### Assembly



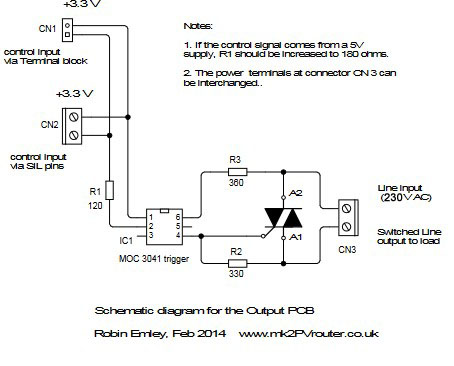
Components to solder :

|  |  |  |
| --- | --- | --- |
| **Label** | **Type** | **Characteristics** |
| R1 | Resistor 0.25W | 47kΩ |
| R2 | Resistor 0.25W | 10kΩ |
| R3 | Resistor 0.25W | 10kΩ |
| R4 | Resistor 0.25W | 100Ω |
| R5 | Resistor 0.5W | 1kΩ |
| R6 | Resistor 0.5W | 120Ω |
| R7 | Resistor 0.25W | 120Ω |
| R11 | Resistor 0.25W | 220Ω |
| R12 | Resistor 0.25W | 220Ω |
| R13 | Resistor 0.25W | 220Ω |
| R14 | Resistor 0.25W | 220Ω |
| R15 | Resistor 0.25W | 220Ω |
| R16 | Resistor 0.25W | 220Ω |
| R17 | Resistor 0.25W | 220Ω |
| R18 | Resistor 0.25W | 220Ω |
| C1 | Capacitor (polarized) | 100µF 25V |
| C2 | Capacitor (polarized) | 10µF 25V |
| C3 | Capacitor | 100nF |
| C4 | Capacitor | 100nF |
| C5 | Capacitor | 100nF |
| C6 | Capacitor | 22pF |
| C7 | Capacitor | 22pF |
| TXFR1 | Transfo | 240V => 6V 3W |
| fuse | Fuseholder | 100mA |
| BR1 | Bridge rectifier | W005G |
| VR1 | 3.3V regulator | Ua78m33C |
| IC1 | Microcontroller (DIL socket) | ATMega328P |
| IC2 | Op ampli (DIL socket) | LM358 |
| X1 | Crystal | 16MHz |
| D1 | Diode | 1N4007 |
| D2 | Diode | 1N4007 |
| CN5 | Header 2 pins | CT1 |
| CN6 | Header 2 pins | CT2 |
| TB2 | Terminal block 2-way[[20]](#footnote-21) | Neutral |
| TB1 | Terminal block 3-way | 230V (L, E, N) |
| CN4 | Header 2-way | Trigger |
| CN3 | Header 2-way[[21]](#footnote-22) | Mode |
| CN2 | Header 6-way[[22]](#footnote-23) | FTDI |
| CN1 | Boxed header 14-way | Display |
| IC3 | Links 2-15, 3-14,4-13,5-12,7-10 |  |
| IC4 | Link 2-15 |  |
| R24 | Link |  |
| R25 | Link |  |
| R26 | Link |  |
| J1-5 | Links 1-10, 2-9, 3-8, 4-7, 5-6 |  |

## Output PCB

It acts as a solid state relay.

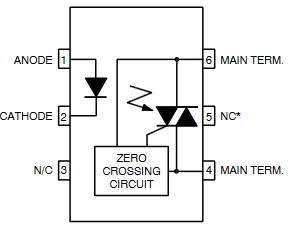
### Schematic[[23]](#footnote-24)



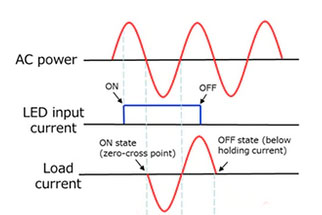
|  |  |
| --- | --- |
|  | * A1 : S/L * A2 : L * G : control |

The **triac** BTA41[[24]](#footnote-25) acts as a controlled switch, maintained until 0 crossing.

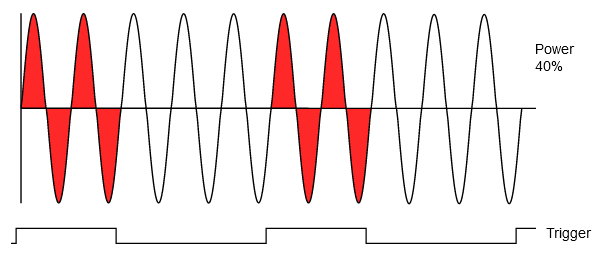
An **optocoupler** MOC3041[[25]](#footnote-26) isolates the 3.3V control from the 230V :



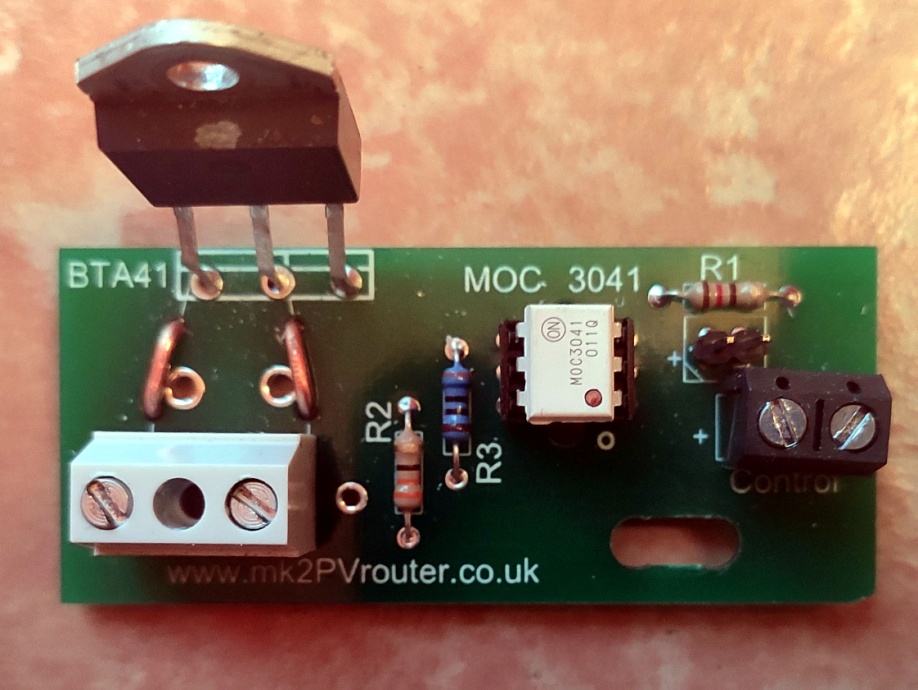
It contains an internal triac again, which controls the previous one, with a **0-crossing detection**[[26]](#footnote-27).



Hence, the load diverted current will begin and stop only at a 0 voltage levels (for resistive loads), which elimitate harmonics, interferences and reduce heating. It explains the so called « **Burst fire mode** » of the router[[27]](#footnote-28) [[28]](#footnote-29):



### Assembly

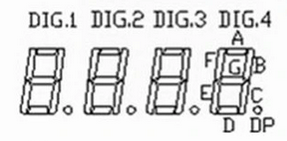


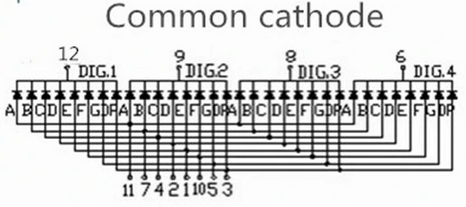
|  |  |  |
| --- | --- | --- |
| **Label** | **Type** | **Characteristics** |
| R1 | Resistor 0.25W | 120Ω |
| R2 | Resistor 0.25W | 330Ω |
| R3 | Resistor 0.25W | 360Ω |
| MOC 3041 | Optocoupler (socket) | MOC3041 |
| BTA41 | Triac | BTA41 |
|  | Terminal Block 2-way 24A | S/L & L |
| Control | Terminal block 2-way 1A |  |
| Control | Header 2-way |  |

## Display PCB



Characteristics[[29]](#footnote-30)

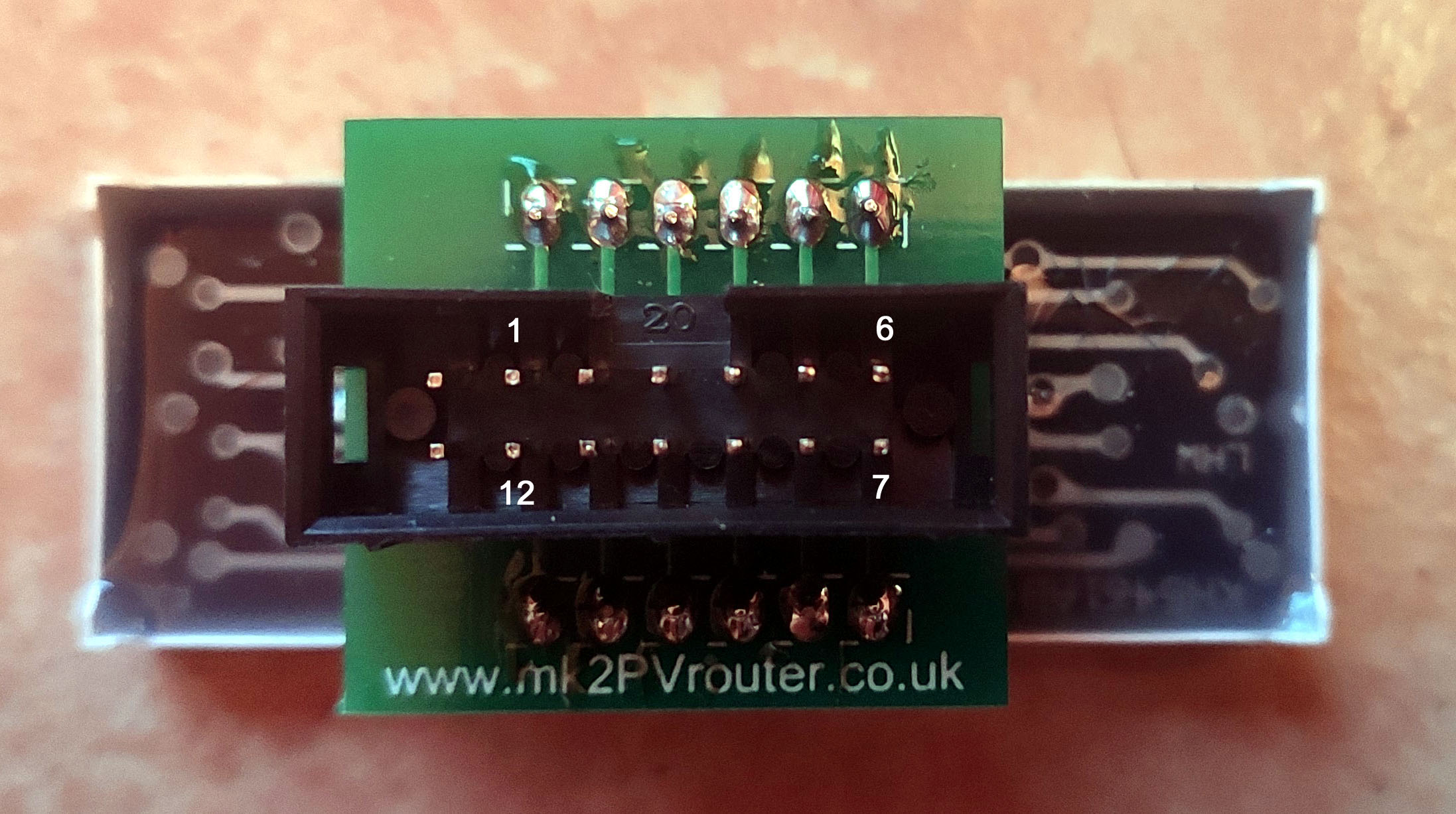




Connection to ATmega 328P pins[[30]](#footnote-31) :

|  |  |  |
| --- | --- | --- |
| **# display pin** | **# ATmega 328P pin** | **# digital Arduino pin** |
| 1 | 13 | 7 |
| 2 | 12 | 6 |
| 3 | 23 | 14 |
| 4 | 18 | 12 |
| 5 | 14 | 8 |
| 6 | 17 | 11 |
| 7 | 11 | 5 |
| 8 | 19 | 13 |
| 9 | 16 | 10 |
| 10 | 15 | 9 |
| 11 | 4 | 2 |
| 12 | 25 | 16 |

Only 12pins of the boxed header are directly connected to the display module whose dot points are up :



## Cables

### CT1

Green and yellow 0.2mm2 cable

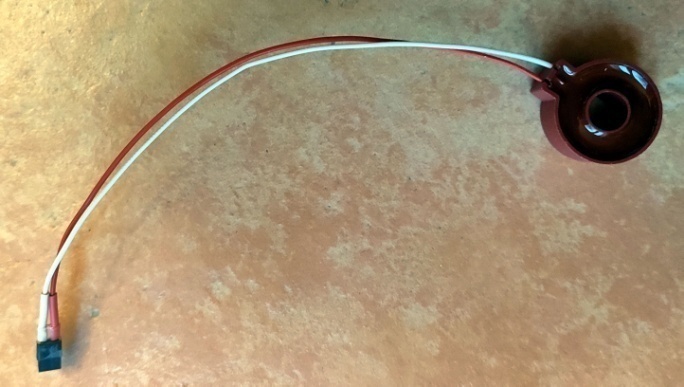


YHDC SCT013 100A :50mA[[31]](#footnote-32)



### CT2

White and red 0.2mm2 cable



### Control

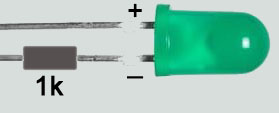
Green and red 0.2mm2 cable



### Led

* Green and red 0.2mm2 cable
* 1kΩ serie resistor on the cathode





### FTDI[[32]](#footnote-33)

Lines (ATmega328P) :

* 1 – DTR
* 2 – TXD
* 3 – RXD
* 6 - GND



### Display

Provided cable :

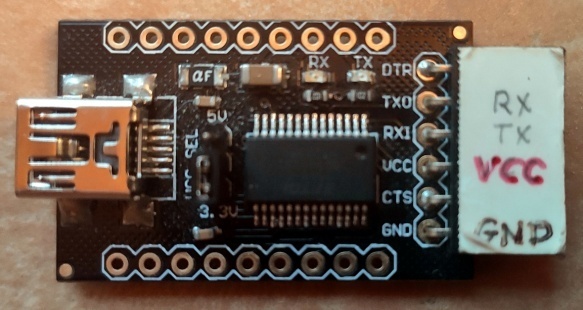


# Testing the router

## The programming tools

### USB⬄Serial TTL converter

* From DfRobot[[33]](#footnote-34)
* Used lines
  + 1 – DTR
  + 2 – TXD (=RX on the converter)[[34]](#footnote-35)
  + 3 – RXD (=TX on the converter)
  + 6 - GND



### IDE Arduino 1.8.13

* Board : Arduino Uno
* COMx port (115200b/s)

## Tests

### Voltage regulation

* Without any ICs
* +3.3V between V+ and 0V pins of CN9 (access to power)

### LM358

* Without the ATmega328P and the MOC3041
* +1.65V between the up pin of CN5 (CT1) and the 0V pin of CN9 (access to power)

### ATmega328P

* Without the MOC3041
* Sketch **Blink\_dig4.ino**
* The green control led blinks with a 3s period (1s on, 2s off)[[35]](#footnote-36)

### Display module

* Without the MOC3041
* Sketch **SegCheck\_bothDisplays.ino**
* Each digit counts successively from 0 to 9 on the display screen

### Voltage and current raw measures

* Without the MOC3041
* Sketch **RawSamplesTool\_2chan.ino**
* CT (CT1) clic-on around a 800W load[[36]](#footnote-37)
* Over-ride switch closed (to power the load)
* Serial monitor :

-------------------------------------

Sketch ID: RawSamplesTool\_2chan.ino

>>free RAM = 706

millis() now = 5039

recordingMayStartAt 10039

4

3

2

1

0

No of cycles recorded = 1

cycleCount 252, samplesRecorded 53

| 21v |

| 2 . 1 v |

| 2 . 1 v |

| 2 . 1 v |

| 2 . 1 v |

| 2 . 1 v |

| 2 . 1 v |

| 2 . 1 v |

| 2 . 1 v |

| 2 . 1 v |

| 2 . 1 v |

| 2 . 1 v |

| 2 .1 v |

| 2. |

| v 1 .2 |

| v 1 . 2 |

| v 1 . 2 |

| v 1 . 2 |

| v 1 . 2 |

| v 1 . 2 |

| v 1 . 2 |

| v 1 . 2 |

| v 1 . 2 |

| v 1 . 2 |

| v 1 . 2 |

| v 1 . 2 |

| v 12 |

min\_V 202, max\_V 820

min\_I1 419, max\_I1 602

min\_I2 420, max\_I2 603

Raw data from stored cycle: <Vsample>, <I1sample>, <I2sample>[cr]

53, <<< No of sample sets

535, 521, 508

573, 533, 495

602, 543, 485

632, 553, 476

658, 562, 466

685, 570, 457

711, 577, 448

739, 587, 439

768, 594, 431

790, 600, 426

801, 602, 423

809, 602, 421

815, 602, 421

818, 601, 420

819, 601, 420

820, 600, 420

819, 599, 420

816, 595, 424

793, 588, 431

762, 579, 439

731, 571, 446

699, 564, 454

663, 553, 463

624, 542, 473

585, 532, 483

546, 520, 496

506, 508, 508

470, 496, 519

439, 486, 529

410, 476, 540

382, 466, 550

356, 457, 559

329, 449, 567

302, 441, 577

273, 432, 586

246, 425, 593

228, 421, 598

219, 419, 599

211, 419, 601

207, 420, 601

204, 420, 603

203, 421, 602

202, 422, 602

203, 424, 601

210, 428, 596

235, 435, 590

265, 443, 582

296, 452, 575

329, 461, 567

367, 471, 557

405, 481, 547

443, 492, 536

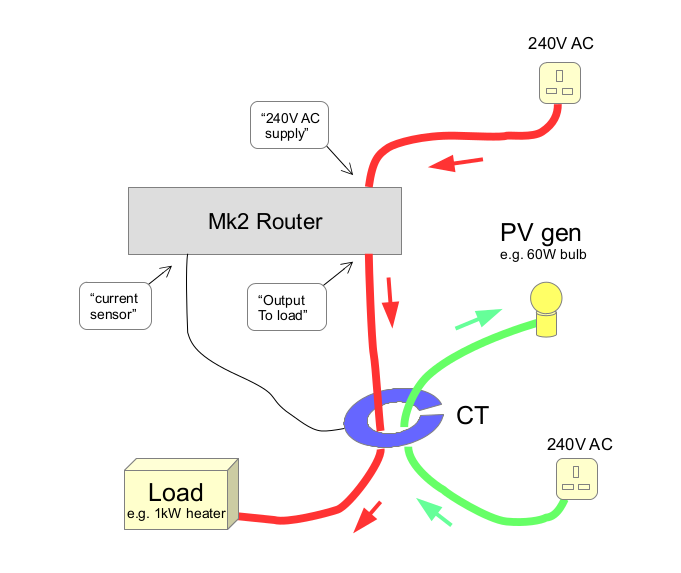
483, 504, 524

### Output PCB

* With the MOC3041 now
* Sketch **Blink\_dig4.ino**
* A bulb on the load blinks as the green LED

### Load control

* Sketch **BalanceCheck\_1kW.ino**
* Use a 60W bulb and a >1kW resistive load (here a 1750W kettle)
* Schematic :



* The load is cyclically powered in such a way that the net flow of current in CT tends to zero.
  + The led blinks[[37]](#footnote-38) and a wattmeter on the load indicates 1350W[[38]](#footnote-39).
  + The « diverted » energy is displayed[[39]](#footnote-40).

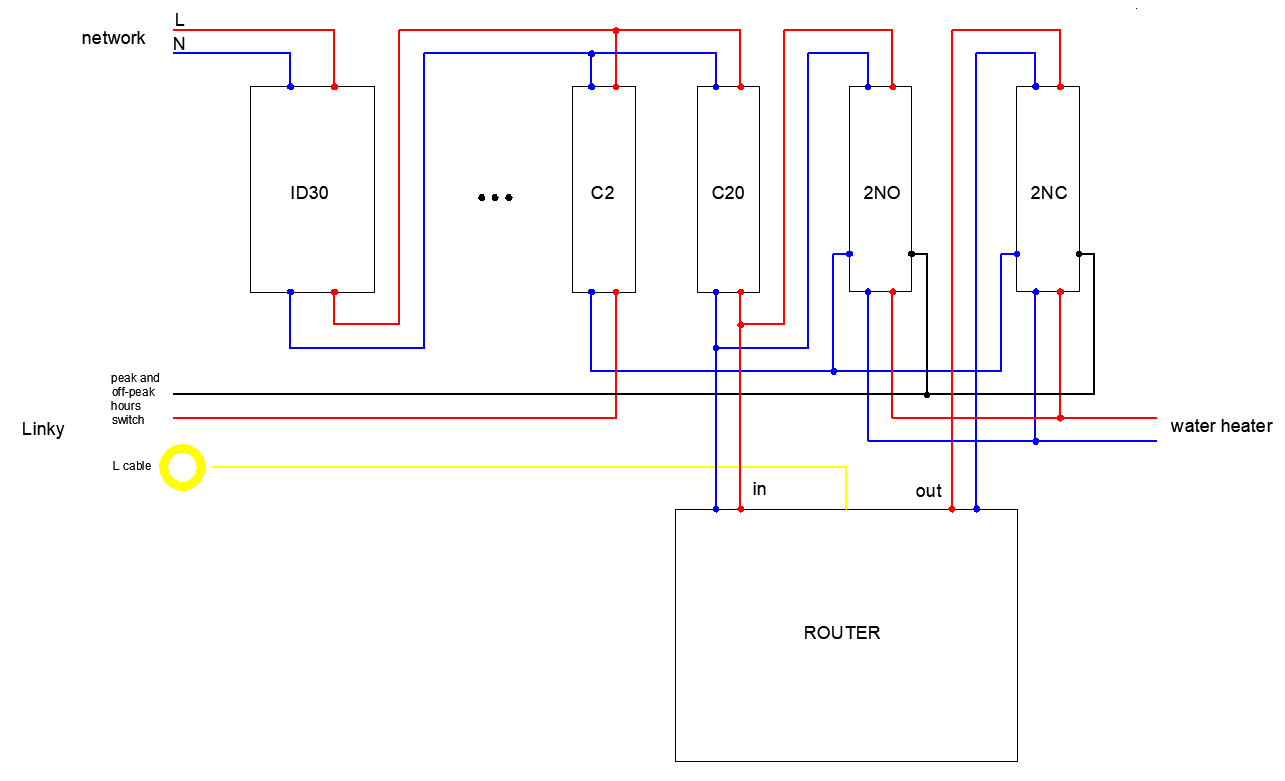
### CTs calibration

* To adjust **powerCal\_grid** and **powerCal\_diverted**
* Sketch **Cal\_BothDisplays\_2.ino**
* Here uses a 60W bulb load with a wattmeter[[40]](#footnote-41) which indicates 58W
* Channel select with the #195 instruction : powerChannel = … ; (GRID or DIVERTED[[41]](#footnote-42))
* The router displays the measured power which has to be tiled with the wattmeter output
* Results[[42]](#footnote-43) :
  + **powerCal\_Grid** = **0.057**
  + **powerCal\_Diverted** = **0.056**

# Connecting the router

The router can be connected to an AC subdistribution board, with[[43]](#footnote-44) a 30mA differential cut-off switch (ID30), and any thermal-magnetic circuit breakers (Cx)[[44]](#footnote-45), specific to the photovoltaic components : solar pannels, inverter(s), battery, etc.

Here the load is a water heater, normally powered during off-peak hours, using a 2NO contactor[[45]](#footnote-46), driven by the Linky counter. Another 2NC contactor[[46]](#footnote-47) can be added to switch off the current coming from the router in this situation[[47]](#footnote-48). And the the load breaker also protects the router :



# Programs

Here we try to structure and to simplify and the Robin Emley’s code using OOP.

Structurations :

* The **Displayer** class to encapsulate the displays
* The **PowerMonitor** class to encapsulate the RMS voltage, RMS current and real power estimations

Simplifications :

* 1-dimension table to define the segments in the Displayer class
* **SetSym** methods
* Straightforward **timer1** settings without using the **TimerOne** library
* Shorter names for the variables
* Shorter code

Addings :

* Circular buffer to store the digitized values (voltage, current CT1, current CT2)

## The Displayer class

### Role

This class implements the display function, for this specific displayer (8-segment 4-digit with common cathode)[[48]](#footnote-49), restricted to numbers. The 2-dimension byte array defined by Robin Emley has been replaced by a 1-dimension byte array and the **SetSym** methods allow to define the digits very simply. The duration of an effective refresh operation is the same (60 µs approximatively).

### Header

**// Displayer.h**

#pragma once

#include <Arduino.h>

#include <elapsedMillis.h>

#define DIGIT\_PERIOD 5 // In ms

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class Displayer

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Simple class to manage a 8-segment 4-digit with a common cathode displayer for numbers

Inspired from :

- Robin Emley codes (https://mk2pvrouter.co.uk/downloads.html)

- Dean Reading SevSeg library (https://github.com/DeanIsMe/SevSeg)

500001 500001 500001 500001

5 1 5 1 5 1 5 1

5 1 5 1 5 1 5 1

466662 466662 466662 466662

4 2 4 2 4 2 4 2

4 2 4 2 4 2 4 2

433332 7 433332 7 433332 7 433332 7

NB

- 23 possible symbols only for each digit : "0", "1", ..., "9", "0.", "1.", ..., "9.", " ", ".", "-"

- Acceptable numbers : -999 <= x <= 9999 ("...." displayed if not)

- Driven by 12 digital pins : 8 => segments and 4 => digits

- A segment is ON/OFF when its pin is HIGH/LOW

- A digit is ENABLED/DISABLED when its pin is LOW/HIGH

- Each digit is ENABLED during DIGIT\_PERIOD ms

\*/

class Displayer

**{**

public**:**

Displayer**(**byte \_tSegPin**[],** byte \_tDigPin**[]);** // Constructor

void SetSym**(**byte sym**,** byte dig**);** // Set a symbol for a digit and " " for the other digits

void SetSym**(**double x**);** // Set symbols for a double

void SetSym**(**int x**);** // Set symbols for an integer

void Refresh**();** // To call in the loop() function

private**:**

byte**\*** tSegPin**;** // Segment pins (size = 8)

byte**\*** tDigPin**;** // Digit pins (size = 4)

byte tSym**[**4**];** // Symbols (#) to display

byte curDig**;** // Current digit

elapsedMillis tmrDig**;** // Timer for displaying a digit

**};**

### Arduino sketches

**// Displayer1.ino**

// Test 1 of the displayer

// Successively display "0", "1", ..., "9" on each digit

#include <Displayer.h>

#include <elapsedMillis.h>

#define DISPLAY\_PERIOD 500 // For displaying a symbol (ms)

byte tSegPin**[]** **=** **{**2**,** 5**,** 12**,** 6**,** 7**,** 9**,** 8**,** 14**};**

byte tDigPin**[]** **=** **{**16**,** 10**,** 13**,** 11**};**

Displayer disp**(**tSegPin**,** tDigPin**);**

byte dig**,** sym**;**

elapsedMillis tmr**;**

void setup**()**

**{**

**for** **(**int i **=** 0**;** i **<** 4**;** i**++)** disp**.**SetSym**(**20**,** i**);** // Display none

dig **=** 0**;**

sym **=** 0**;**

disp**.**SetSym**(**sym**,** dig**);**

**}**

void loop**()**

**{**

disp**.**Refresh**();**

**if** **(**tmr **>** DISPLAY\_PERIOD**)**

**{**

tmr **=** 0**;**

sym**++;**

**if** **(**sym **>** 9**)**

**{**

sym **=** 0**;**

disp**.**SetSym**(**20**,** dig**);**

dig **=** **++**dig **%** 4**;**

**}**

disp**.**SetSym**(**sym**,** dig**);**

**}**

**}**

**// Displayer2.ino**

// Test 2 of the displayer

// Display different integer or real numbers

#include <Displayer.h>

#include <elapsedMillis.h>

#define DISPLAY\_PERIOD 1000 // For displaying a number (ms)

#define NO\_NUMBERS 10

byte tSegPin**[]** **=** **{**2**,** 5**,** 12**,** 6**,** 7**,** 9**,** 8**,** 14**};**

byte tDigPin**[]** **=** **{**16**,** 10**,** 13**,** 11**};**

Displayer disp**(**tSegPin**,** tDigPin**);**

elapsedMillis tmr**;**

double tNumber**[**NO\_NUMBERS**]** **=** **{**213**,** **-**213**,** **-**1234**,** 9999**,** 10240**,** 3.1416**,** **-**3.1416**,** 0.00123**,** 5E+3**,** **-**5E+3**};**

byte id**;**

void setup**()**

**{**

id **=** 0**;**

disp**.**SetSym**(**tNumber**[**0**]);**

**}**

void loop**()**

**{**

disp**.**Refresh**();**

**if** **(**tmr **>** DISPLAY\_PERIOD**)**

**{**

tmr **=** 0**;**

id **=** **++**id **%** NO\_NUMBERS**;**

disp**.**SetSym**(**tNumber**[**id**]);**

**}**

**}**

**// Displayer3.ino**

// Test 3 of the displayer

// Time to refresh

#include <Displayer.h>

#include <elapsedMillis.h>

#define PERIOD 500

byte tSegPin**[]** **=** **{**2**,** 5**,** 12**,** 6**,** 7**,** 9**,** 8**,** 14**};**

byte tDigPin**[]** **=** **{**16**,** 10**,** 13**,** 11**};**

Displayer disp**(**tSegPin**,** tDigPin**);**

elapsedMillis tmr**;**

void setup**()**

**{**

Serial**.**begin**(**115200**);**

disp**.**SetSym**(**3.14**);**

**}**

void loop**()**

**{**

unsigned long duration **=** micros**();**

disp**.**Refresh**();**

duration **=** micros**()** **-** duration**;**

**if** **(**tmr **>** PERIOD**)**

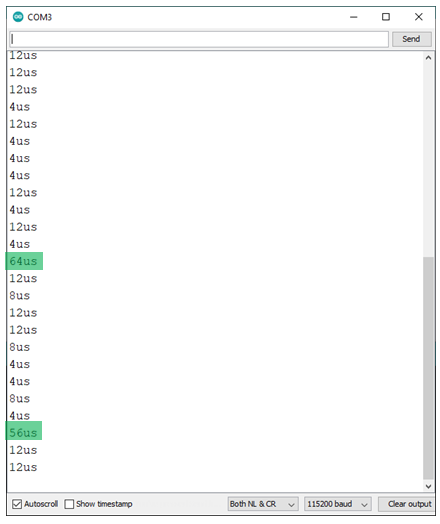
**{**

tmr **=** 0**;**

Serial**.**print**(**duration**);** Serial**.**println**(**"us "**);**

**}**

**}**



## The PowerMonitor class

### Role

Estimate continuously the RMS voltage, RMS current(s) and real power(s), using many channels : the first for the voltage, the other(s) for the current(s).

Inspired from :

* The **Cal\_bothDisplays\_2.ino** sketch, from Robin Emley[[49]](#footnote-50)
* The **EmonLib**[[50]](#footnote-51) and the **EmonLibCM**[[51]](#footnote-52) Arduino libraries

### A bit of theory

For a -periodic AC current , with and for instantaneous voltage and current :

* RMS voltage (V) :
* RMS current (A) :
* Real[[52]](#footnote-53) power (W) :
* Apparent power (VA) :
* Power factor :

In these formulas, is a small sampling period and is the number of sampled values in the interval of time containing complete cycles. In practice, will correspond to the begining of a positive alternance.

The Cauchy-Schwarz inequality ensures that :

and the Ohm law that , with a pure resistive load.

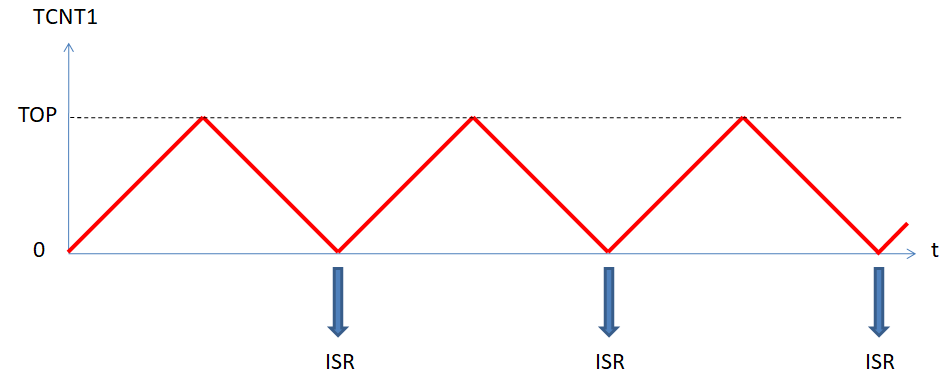
Moreover, with any type of load, but in the special case of a perfect AC sinusoid current, it can be proven that :

Where is the phase shift between voltage and current.

### Timer for the sampling

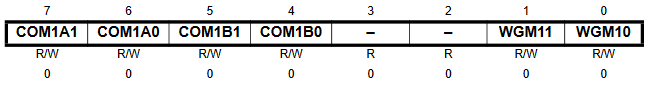
Following the Robin Emley’s **Cal\_bothDisplays\_2.ino** code, this evaluation is performed continuously, using interrupts generated with a fixed period, when the timer **Timer1** overflows.

As in the **TimerOne** library[[53]](#footnote-54) from Paul Stoffregen, the **phase correct and frequency PWM mode, with TOP in ICR1 and TOV1 flag set on BOTTOM[[54]](#footnote-55)** is selected here for this timer, which counts repeatedly from 0 to TOP then from TOP to 0, and so will call the **ISR(TIMER1\_OVF\_vect)** after **2xTOP[[55]](#footnote-56)** steps :

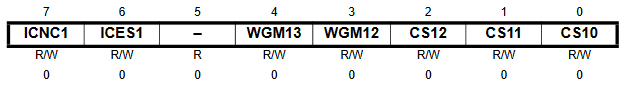


Without any prescaler and a 16MHz system clock frequency, each timer step takes (1/16)µs ; so, a value of 8xT in TOP will generate ISR calls with a Tµs period. Moreover, to encapsulate all the staff in a C++ class, we need to define a static pointer in the constructor, because the ISR(TIMER1\_OVF\_vect) function requires static variables : hence, only one instance of the class may be defined at a time, but it’s not a problem here, because only one timer is needed.

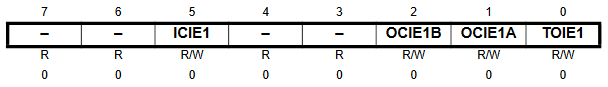
The **TCCR1A** control register is set to 0 for disabling any PWM outputs on pins number 9 and 10[[56]](#footnote-57) :



In the **TCCR1B** control register, only the bits **WGM13** and **CS10** are set, for selecting the previous mode, with no prescaling[[57]](#footnote-58) :



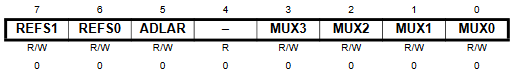
And finally the timer OVF interrupts are enabled on setting to 1 the **TOIE1** bit in the **TIMSK1** mask register[[58]](#footnote-59) :



### Analog to digital conversions

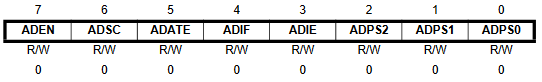
The second specialized circuit used here is the Analog to Digital Converter (**ADC**) component of the ATMega328P[[59]](#footnote-60) which can convert an analog signal in [0, VRef] to a digital value in {0, 1, …, 1023}. It processes iteratively, using a specific clock which period define its precision.

Fisrtly, we need to set the voltage reference, VRef, choosen here equal to AVcc (3.3V), on setting the **REFS0** bit in the **ADMUX** register [[60]](#footnote-61):



The 4 lowest bits allow the source selection : latter we shall precise the voltage or current source, using them .

Secondly, we need to precise the ADC clock period, on defining the prescaler, using the **ADPS** bits in the **ADCSRA** control and status register [[61]](#footnote-62):



Here, we choose a prescaler value equal to 128 ( **ADPS** = 0b111), which corresponds to the maximum **ADC** clock period value, 128/16 = 8µs, and guaranties the best precision in the conversions. Because, each conversion takes 13 cycles[[62]](#footnote-63) and is performed in parallel (without consuming any CPU cycles), a new conversion can be available every 104µs. Then, to convert successively a voltage and 2 current sources, 312µs will be requested at least, because there is only one **ADC** component in the ATMega328P. In practice, we used the Robin Emley’s period for the timer, 125 µs, to allow the supplementary operations in the ISR function.

Observe again that this register can enable the converter, using the **ADEN** bit, and start a new conversion, using the **ADSC** bit, which is cleared when the conversion is complete.

### Filtering

To estimate the voltage and the current values, we have to center their digitized raw values. In theory the means are equal to 511.5, but in practice, they can vary a little.

So it’s necessary to estimate these means and the simplest way to proceed, consists in using an exponential moving average, because it can be evaluated recursively, very simply :

* **Digitized raw values** :

where (voltage) or (current)

* **Exponential moving average** :

given[[63]](#footnote-64)

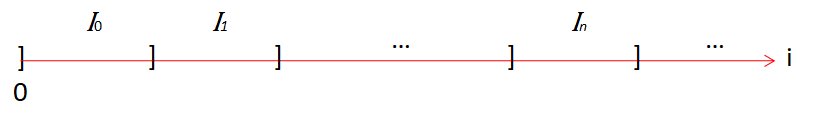
Centered raw values :

for every

This is the filtering technic used in the **emonLib**[[64]](#footnote-65) library, with . It requires a new mean estimation calculus for each sampling,. But Robin Emley observes that it was not really necessary and that an estimation calculus for each cycle only is sufficient. This technic is based on the following formula obtained on adding the previous recurrence formulas :

* **Exponential moving average by bloc** :

Here, the indices are supposed to be grouped in successive intervals :



and the following formulas are used :

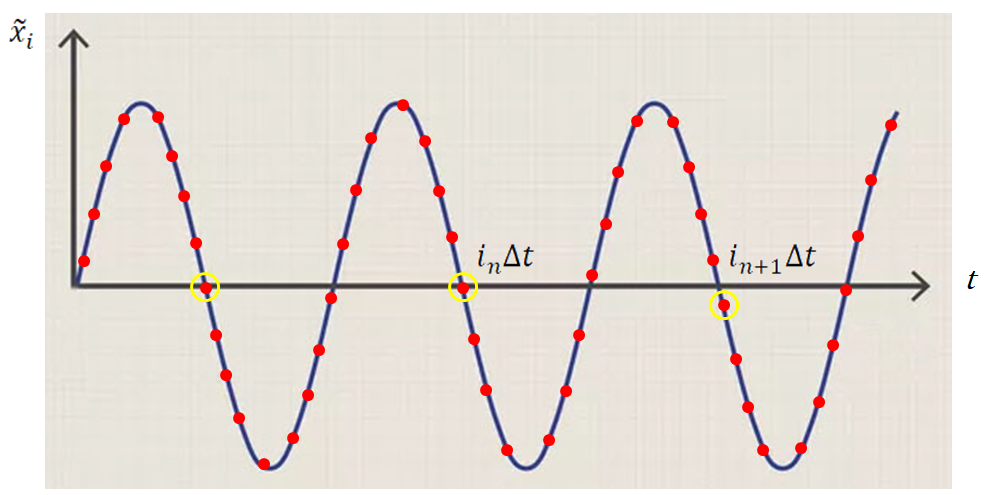
given

Centered raw values :

when

Robin Emley chooses and starts a new interval at the begining of each negative alternance, for the centered digitized raw values of the voltage conversions :

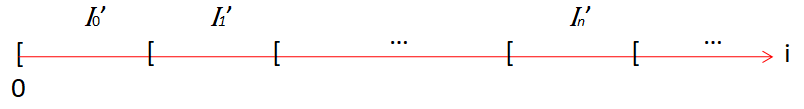
and



For instance, with a 50Hz AC current and , each intervals contains approximatively elements.

### Rms voltage, rsm current and real power estimations

In a similar way, they are performed, on successive groups of cycles, starting now at the begining of a positive alternance, in view to equilibrate the treatments :



Using the digitized centered raw values for the voltage and the current , as previously defined, these estimations are constructed on evaluating the 3 sums :

### Phase shifting for the voltage

In practice, little phase shifts are observed between the voltage and the current digital values, probably because their evaluations are time shifted and because the current transformers (CTx) are inductive loads. As mentionned previously, they cause the inequalites  , even with resistive loads.

If necessary[[65]](#footnote-66), they can be reduced using the phase correction algorithm[[66]](#footnote-67) ; in place of the third sum, it consists in evaluating :

where ( near 1) :

When , no phase shift is applied.

### Calculus with int values

Without a floating-point coprocessor unit, the ATMega328P is not rapid to realize operations on **float** or **double** numbers. Because speed is essential here, it’s a good option to realize the operations using integer values.

The raw digitized values[[67]](#footnote-68) are multiplied by and stored in **long** typed variables[[68]](#footnote-69). Hence, the estimated means are multiplied by and stored in **long** typed variables again.

This 256 (=28) factor and the 4096 (=212) coefficient used to update the means are choosen because their usage in a multiplication or in a division with an integer variable is equivalent to a **bit shift** :

B = A\*2x ⬄ B = A<<x and B = A/2x ⬄ B = A>>x [[69]](#footnote-70)

Hence, if **digV** is an **int** variable containing a digitized raw value for the voltage and **offV** is a **long** variable containing the actual mean estimation for the voltage, the instruction :

V **=** **(((**long**)**tDig**[**0**])<<**8**)-** offV**;**

will define a long variables with the corresponding filtered value[[70]](#footnote-71).

We proceed in a same way for the current channels, but using here arrays, **tC[.]**

At each sampling, these values are added :

sumV **+=** V**;**

to update the means at the begining of each negative alternance[[71]](#footnote-72) :

offV **+=** **(**sumV**>>**12**);**

For the sums considered in the §5.2.5, a reduction (division by 4 at least) is necessary to avoid capacity overflows :

V **=** **(**V**>>**2**);**[[72]](#footnote-73)

sumVV += ((V\*V)>>6);[[73]](#footnote-74)

We proceed in the same way for the current channels, using arrays.

For the real powers, the phase correction algorithm is applied with a specific coefficient for each current channel. All these coefficients are in fact integers, obtained on multiplying the values by 256 and stored in the **tPha** array ; hence the >>8 to implement the §5.2.7 formula :

phaV = preV + (((V - preV)\*tPha[i])>>8);

tSumVC[i] += ((phaV\*C)>>6);

So, with these rescalings, we just have to apply again the >>6 (⬄ /64.) scaling, on their final values, at the start on a new positive alternance, to obtain the RMS and real power estimations from the saved end values of these sums and their counts, on the previous group of cycles, as if we used the centered digitized raw values without any scaling ; for instance, an estimation of can be obtained with the following formula :

tCal[0]\*sqrt(((double)savSumVV)/(64.\*nSavSum));

where **tCal[0]** is a constant which converts a digitized raw value in Volt, for the measured AC current.

Equivalent formulas are used for the measured currents (grid or diverted here).

### The ring buffer

The ISR treatment, implemented in the **FuncISR()** method, need to be as rapid as possible because it interrupts all other treatments in the microcontroler and is called very frequently if the sampling period is low. So another **Process()** method is necessary to implement the previous treatments on the digitized raw values ; this method shall be called in the Arduino **loop()** function as frequently as possible.

To communicate asynchronously between these 2 methods, we opted to store the digitized values in a First In First Out (FIFO) circular buffer, which elements are arrays of **int**,containing the digitized raw values (the fisrt for voltage, the others for currents) : the **FuncISR()** method pushes such arrays and the **Process()** method pops them[[74]](#footnote-75).

There exist several circular buffer libraries for the Arduino, but we opted to tailor one, very simple, for our problem, using a template.

The class **RingBuffer** defines and implements such a circular buffer, with an one dimension array of MN integers  and systematic pointer operations to optimize the treaments :

**// RingBuffer.hpp**

#pragma once

#include <Arduino.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class RingBuffer<M, N>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Circular Buffer FIFO data structure, with M arrays of N int elements

\*/

template **<**int M**,** int N**>** class RingBuffer

**{**

public**:**

// Constructor

RingBuffer**()** **{ … }**

// Number of elements

int Size**()** **{return** size**;}**

// Operations

void Push**(**int t**[]) { … }**

int Pop**(**int t**[]) { … }** // Before test if Size()>0

private**:**

int buffer**[**M**\***N**];**

int **\***pHead**,** **\***pTail**,** **\***pStart**,** **\***pEnd**;**

int size**;**

**};**

Its use is then obvious :

* Declare a ring buffer with a a capacity of M arrays of N integers :

RingBuffer**<**M**,** N**>** rb**;**

* Push an array of N integers :

int t**[**N**];**

…

t**[**0**]** **=** … **;**

**…**

t**[**N-1**]** **=** … **;**

**…**

rb**.**Push**(**t**);**

* Pop an array of N integers :

int s**[**N**];**

…

**if** **(**rb**.**Size**()** **>** 0**)**

**{**

rb**.**Pop**(**s**);**

// then use s[0],…, s[N-1]

**}**

### Testing code

We added a code to display :

* The maximum value of the circular buffer size
* The number of samples in the group of cycles
* The maximum processing time (µs) to treat a sample

To use it, uncomment the define instruction :

// #define TESTING

### Header

**// PowerMonitor.h**

#pragma once

#include <Arduino.h>

#include "RingBuffer.hpp"

#define M 100 // Buffer size <256

#define N 3 // Number of channels : V, C1, C2, etc.

#define NM1 2 // N-1 = number of current channels

#define TESTING

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class PowerMonitor

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Rms voltage, rms current(s) and real power(s) estimations for a voltage and one

or many current channels, the voltage channel being the first

Inspired from :

Robin Emley's Cal\_bothDisplays\_2.ino sketch ( https://www.mk2pvrouter.co.uk )

EmonLib library ( https://github.com/openenergymonitor/EmonLib )

EmonLibCM library ( https://github.com/openenergymonitor/EmonLibCM )

Constructor parameters :

\_period = sampling period for the channels (in us and 105<.<8192)

\_nCycl = number of cycles to construct the estimations (>=1)

\_tPin[N] = Arduino pins array for the channels

\_tCal[N] = Calculus coefficient array to convert a digitized value

\_tPha[NM1] = Phase shift coefficient array of the voltage

NB : 256 => no shift, <256 => right shift, >256 => left shift

\*/

enum Polarities **{**POSITIVE**,** NEGATIVE**};** // Alternance polarities

extern "C" void TIMER1\_OVF\_vect**();**

class PowerMonitor

**{**

public**:**

// Constructor

PowerMonitor**(**int \_period**,** int \_nCycl**,** byte \_tPin**[],** double \_tCal**[],** int \_tPha**[]);**

void Init**();** // Initialisation (to be called in the setup)

void Process**();** // Process calculus outside the ISR (to be called in the loop()

double RmsVoltage**();** // Vrms (V)

double RmsCurrent**(**int i**);** // Irms (A) with i=0 => CT1, i=1 => CT2, etc.

double RealPower**(**int i**);** // Pr (W) with i=0 => CT1, i=1 => CT2, etc.

#ifdef TESTING

byte MaxSize**();** // Maximum size of the circular buffer, on entering Process()

int NbSamples**();** // Number of samples in estimations

long MaxProcTime**();** // Max time to treat a sample (us)

#endif

private**:**

unsigned int period**;** // Sampling period for the channels

int nCycl**,** iCycl**;** // Number of cycles and index of the cycle

byte**\*** tPin**;** // Arduino pins array for the channels

double**\*** tCal**;** // Calculus coefficients array to convert the digitized vales

int**\*** tPha**;** // Phase shift coefficients array of the voltage

int iChan**;** // Index of the channel in the digitalization

RingBuffer**<**M**,** N**>** rbDig**;** // Circular buffer for the digitized values

int tDigISR**[**N**];** // Digitized values array pushed in the ISR

int tDig**[**N**];** // Digitized values array poped in the Process method

long V**;** // Voltage value (<<8 and centered)

long tC**[**NM1**];** // Current values array (<<8 and centered)

Polarities pol**;** // Polarity of the alternance

long offV**;** // Offset voltage for centering

long tOffC**[**NM1**];** // Offset currents array for centering

long sumV**;** // Sum of the voltage values

long tSumC**[**NM1**];** // Sum of the current values array

long sumVV**;** // Sum of the squared voltage values (>>10)

long savSumVV**;** // Saved sum of the squared voltage values (>>10)

long tSumCC**[**NM1**];** // Sum of the squared current values array (>>10)

long tSavSumCC**[**NM1**];** // Saved sum of the squared current values array (>>10)

long tSumVC**[**NM1**];** // Sum of the VC products array (>>10)

long tSavSumVC**[**NM1**];** // Saved sum of the VC products array (>>10)

long preV**;** // Previous voltage value (>>2)

long phaV**;** // Phase shifted voltage value (>>2)

int nSum**;** // Number (saved) of values in the sums VV, CC and VC

int nSavSum**;** // Number (saved) of values in the sums VV, CC and VC

void FuncISR**();** // ISR function

#ifdef TESTING

byte maxSize**;** // Maximum size of the circular buffer, on entering Process()

long maxProcTime**;** // Maximum of processing time

#endif

friend void TIMER1\_OVF\_vect**();** // To use the private FuncISR() in the ISR

**};**

### Arduino sketch

**// PowerMonitor.ino**

// Test the PowerMonitor class

#include "PowerMonitor.h"

#include <elapsedMillis.h>

byte tPin**[**N**]** **=** **{**3**,** 5**,** 4**};**

double tCal**[**N**]** **=** **{**1.01**,** 0.055**,** 0.055**};**

int tPha**[**NM1**]** **=** **{**312**,** 279**};** // {256, 256}

PowerMonitor pm**(**125**,** 2**,** tPin**,** tCal**,** tPha**);**

elapsedMillis tmr**;**

double Vrms**,** Irms**,** Pr**,** Pa**;**

void setup**()**

**{**

Serial**.**begin**(**9600**);**

pm**.**Init**();**

**}**

void loop**()**

**{**

// delay(30);

pm**.**Process**();**

**if** **(**tmr**>**2000**)**

**{**

tmr **=** 0**;**

#ifdef TESTING

Serial**.**print**(**"MaxSizeBuffer = "**);** Serial**.**print**(**pm**.**MaxSize**());**

Serial**.**print**(**" NbSamples = "**);** Serial**.**print**(**pm**.**NbSamples**());**

Serial**.**print**(**" MaxProcTime = "**);** Serial**.**print**(**pm**.**MaxProcTime**());** Serial**.**println**(**"us"**);**

#else

Vrms **=** pm**.**RmsVoltage**();**

Irms **=** pm**.**RmsCurrent**(**0**);**

Pr **=** pm**.**RealPower**(**0**);**

Pa **=** Vrms**\***Irms**;**

Serial**.**print**(**"Vrms = "**);** Serial**.**print**(**Vrms**);** Serial**.**print**(**"V"**);**

Serial**.**print**(**" Irms = "**);** Serial**.**print**(**Irms**);** Serial**.**print**(**"A"**);**

Serial**.**print**(**" Pr = "**);** Serial**.**print**(**Pr**);** Serial**.**print**(**"W"**);**

Serial**.**print**(**" Pa = "**);** Serial**.**print**(**Vrms**\***Irms**);** Serial**.**print**(**"VA"**);**

Serial**.**print**(**" F = "**);** Serial**.**println**(**abs**(**Pr**)/**Pa**);**

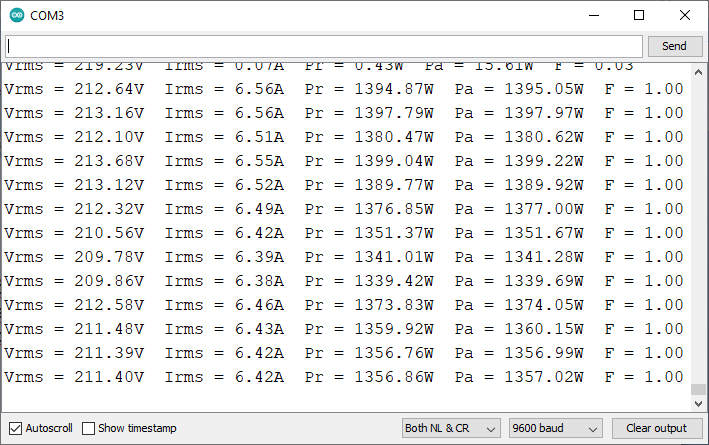
#endif

**}**

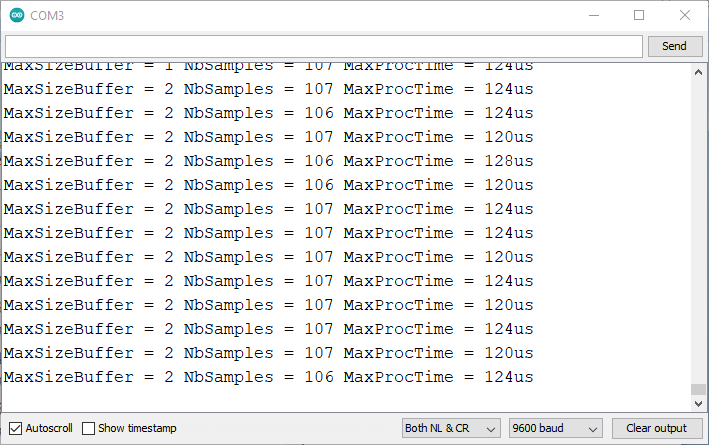
**}**

Here the calculus and shift phased coefficients are tuned, using a low cost wattmeter :



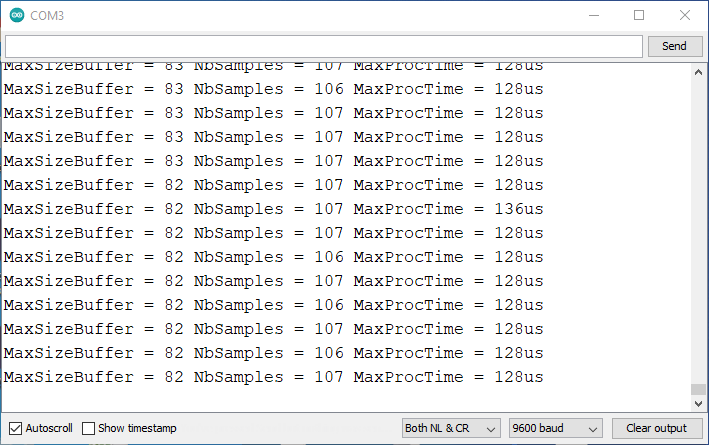


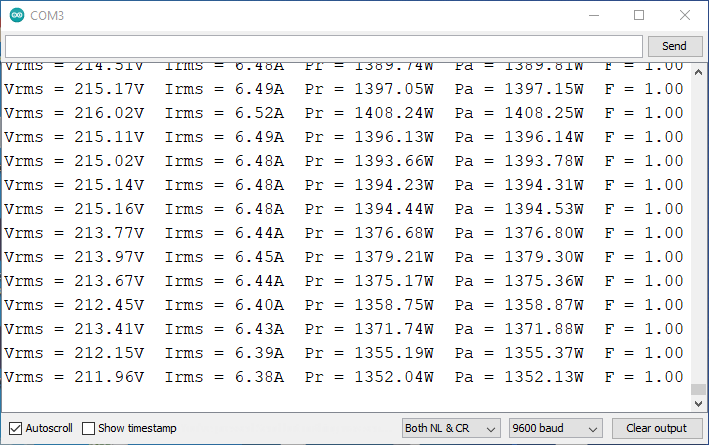
On defining **TESTING**, we obtain :



This justifies the circular buffer (**MaxSizeBuffer** = 2) and displays a correct value for the **NbSamples**: with a 50Hz AC current, and a sample each 3x125=375µs, there are 20000/375≅53.33 samples in each cycle, hence 106 or 107 for 2 cycles. Notice that the **MaxProcTime** for processing one sample is much more lower than 375µs : this lets a great place for other treatments in the **Process()** function.

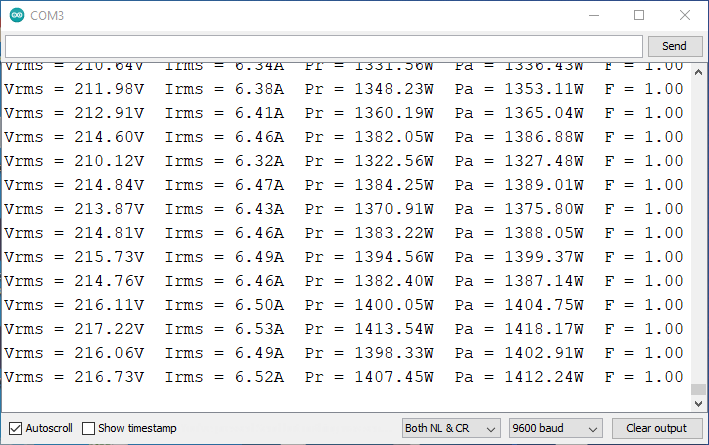
The need of a circular buffer appears more clearly if we add a 30ms delay, in the **loop()** function :

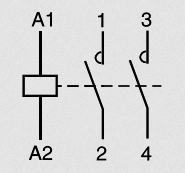
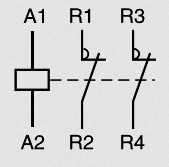




Hence, this buffering technic offers a great tolerance in the **Process()** calling frequency : for instance, no problem to use the **Displayer** class here.

Finally, we can confirm the Robin Emley choice to do not use a shift phased correction, with the neutral **256** values in the **tPha** array, in place of the tuned ones :



1. <https://mk2pvrouter.co.uk/> [↑](#footnote-ref-2)
2. <https://learn.openenergymonitor.org/pv-diversion/introduction/choosing-an-energy-diverter.md> [↑](#footnote-ref-3)
3. <https://www.youtube.com/watch?v=_XvB7SIO4-w> [↑](#footnote-ref-4)
4. <https://www.youtube.com/watch?v=UtEi39gQp8s> [↑](#footnote-ref-5)
5. NB : with a wrong description of the Burst fire mode [↑](#footnote-ref-6)
6. <https://www.youtube.com/watch?v=iSK7ZSSW8GI> [↑](#footnote-ref-7)
7. <https://sites.google.com/view/le-professolaire/horloge> [↑](#footnote-ref-8)
8. Adapted from the Robin Emley’s schematic [↑](#footnote-ref-9)
9. Adapted from Emley schematic [↑](#footnote-ref-10)
10. <https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061B.pdf> [↑](#footnote-ref-11)
11. The analog pins can be used as digital pins with number 14 (A0), …, 19 (A5) [↑](#footnote-ref-12)
12. To disable the **anti-flicker** processing : not used in this version. [↑](#footnote-ref-13)
13. Hence, to power the load, the pin #6 should be set LOW [↑](#footnote-ref-14)
14. <https://docs.rs-online.com/c0dc/0900766b80dd0628.pdf> [↑](#footnote-ref-15)
15. <https://www.ti.com/lit/ds/symlink/ua78m.pdf> [↑](#footnote-ref-16)
16. <https://www.onsemi.com/pdf/datasheet/lm358-d.pdf> for instance [↑](#footnote-ref-17)
17. <https://www.allaboutcircuits.com/video-tutorials/op-amp-applications-voltage-follower/> [↑](#footnote-ref-18)
18. 230V is a RMS voltage => [↑](#footnote-ref-19)
19. <https://falstad.com/circuit/circuitjs.html> [↑](#footnote-ref-20)
20. Not used here [↑](#footnote-ref-21)
21. Not used here [↑](#footnote-ref-22)
22. Only pin 1 (RST), 2 (TX), 3 (RX) and 6 (GND) are connected [↑](#footnote-ref-23)
23. Adapted from Robin Emley schematic [↑](#footnote-ref-24)
24. <https://www.st.com/resource/en/datasheet/bta41.pdf> [↑](#footnote-ref-25)
25. <https://www.onsemi.com/pdf/datasheet/moc3043m-d.pdf> [↑](#footnote-ref-26)
26. <https://toshiba.semicon-storage.com/eu/semiconductor/knowledge/faq/opto/opto-054.html> [↑](#footnote-ref-27)
27. <https://learn.openenergymonitor.org/pv-diversion/mk2/switchdev> [↑](#footnote-ref-28)
28. NB : the mode connector DOES NOT select the **Phase angle mode**! [↑](#footnote-ref-29)
29. <https://fut688.en.made-in-china.com/product/yNtnvlTGhhVW/China-0-36-Inch-4-Digit-LED-Display-7-Seg-Segment-Common-Cathode-Red.html> [↑](#footnote-ref-30)
30. With 220Ω resistor for A, B, …, G, DP pins, and no resistor for the DIGx pins (green) [↑](#footnote-ref-31)
31. <https://en.yhdc.com/comp/file/download.do?id=941> [↑](#footnote-ref-32)
32. Added to program the microcontroller, without having to open the router box. [↑](#footnote-ref-33)
33. <https://www.robotshop.com/media/files/PDF/datasheet-dfr0065.pdf> [↑](#footnote-ref-34)
34. Here TXD and RXD indicate the FTDI pins TX and RX on the microcontroller [↑](#footnote-ref-35)
35. NB : pin #6 of the ATmega328P LOW to illuminate the LED [↑](#footnote-ref-36)
36. Reverse the CT if the 2 current curves are superimposed [↑](#footnote-ref-37)
37. If not, reverse the CT1 connection [↑](#footnote-ref-38)
38. When the router is overriden, it indicates 1750W [↑](#footnote-ref-39)
39. If not, reverse the CT2 connection [↑](#footnote-ref-40)
40. <https://uk.banggood.com/SINOTIMER-DDS109L-EU-or-US-or-UK-or-FR-or-AU-AC-110V-or-220V-Plug-Socket-Digital-Wattmeter-Meter-Power-Consumption-Watt-Energy-Meter-KWh-Electricity-Analyzers-Monitors-with-Backlight-p-1816558.html> [↑](#footnote-ref-41)
41. In this case, the override need to be selected [↑](#footnote-ref-42)
42. To use in place of 0.0435 in the programs [↑](#footnote-ref-43)
43. According to the NF C 15-100 electrical standard in France [↑](#footnote-ref-44)
44.  (2 poles but with only the Line protected) [↑](#footnote-ref-45)
45.  [↑](#footnote-ref-46)
46.  [↑](#footnote-ref-47)
47. Maybe useless… [↑](#footnote-ref-48)
48. Cf. 2.4 [↑](#footnote-ref-49)
49. [www.Mk2PVrouter.co.uk](http://www.Mk2PVrouter.co.uk) [↑](#footnote-ref-50)
50. <https://github.com/openenergymonitor/EmonLib> [↑](#footnote-ref-51)
51. <https://github.com/openenergymonitor/EmonLibCM> [↑](#footnote-ref-52)
52. « Active » power also [↑](#footnote-ref-53)
53. <https://github.com/PaulStoffregen/TimerOne> [↑](#footnote-ref-54)
54. Mode number 8 in the data sheet (<https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061B.pdf>, chap 16, p 141) [↑](#footnote-ref-55)
55. 0→1, …, TOP-1→TOP, TOP→TOP-1, …, 1→0 [↑](#footnote-ref-56)
56. Data sheet p 140 [↑](#footnote-ref-57)
57. Data sheet p 142 [↑](#footnote-ref-58)
58. Data sheet p 144  [↑](#footnote-ref-59)
59. Data sheet (<https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061B.pdf> ) chap. 24 [↑](#footnote-ref-60)
60. Data sheet p 257 [↑](#footnote-ref-61)
61. Data sheet p 258 : prescaler = 2^ADPS [↑](#footnote-ref-62)
62. Excepted the first which takes the twice [↑](#footnote-ref-63)
63. For instance 512 [↑](#footnote-ref-64)
64. <https://github.com/openenergymonitor/EmonLib> [↑](#footnote-ref-65)
65. Robin Emley proposes to do not use this correction, considering the precision sufficient without using it. [↑](#footnote-ref-66)
66. <https://learn.openenergymonitor.org/electricity-monitoring/ctac/explanation-of-the-phase-correction-algorithm> [↑](#footnote-ref-67)
67. In [↑](#footnote-ref-68)
68. Integer values in [↑](#footnote-ref-69)
69. For a signed integer variable A, A>>x preserves the sign of A and return the quotient of the integer division of A by 2x. But A<<x preserve the sign of A iff there is no overflow ; for instance :

    int x = 16000; int y = x>>2; // store the value -4000 in y

    int x = -16000; int y = x>>2; // store the value -4000 in y

    int x = 30; int y = x>>5; // store the value 0 in y

    int x = -30; int y = x>>5; // store the value -1 in y

    int x = -16000; int y = x<<1; // store the value -32000 in y

    int x = -16000; int y = x<<2; // store the value 1536 in y [↑](#footnote-ref-70)
70. Integer values in normally [↑](#footnote-ref-71)
71. Apparently, the tests that 412x256 ≤ offV ≤ 612x256 proposed by Robin Emley are not necessary here [↑](#footnote-ref-72)
72. Integer values in [↑](#footnote-ref-73)
73. Integer values in because there are less than 128 = 27 terms in the sum [↑](#footnote-ref-74)
74. This mode of operation requires to stop the interrupts when poping. [↑](#footnote-ref-75)