Power Measurement
Energy consumption acquisition and processing

B. VAN WALSTIJN, R. STAUTTENER, R. BOLDING
Amsterdam University of Applied Sciences
March 5, 2019

-Overview

The power consumption of a whole Raspberry Pi cluster was measured, processed and sent to a visualisation interface. Every individual Raspberry Pi has an external hardware interface which measures physical quantities and communicates to the Raspberry Pi. The acquisited data is processed and forwarded to a monitoring agent. The monitoring agent collects all data, processes these and hosts a server. All data from this server is requested for by visualisation front-end software for the end user to monitor. In this paper we will discuss the implementations used and thought for.

-Hardware interface

The hardware interface has been made for the Raspberry Pi. The function of the hardware interface is to measure the voltage and current consumed by the Raspberry Pi. The hardware interface is made so that it can be placed on top of the Raspberry Pi, connecting via the GPIO pins. For designing the hardware interface we have used EAGLE, an application intended for designing electronic schematics and printed circuit boards (PCB's). The hardware consists of the following components:

- Power connector
- Current measurement circuitry
- Filtering

- GPIO connection pins

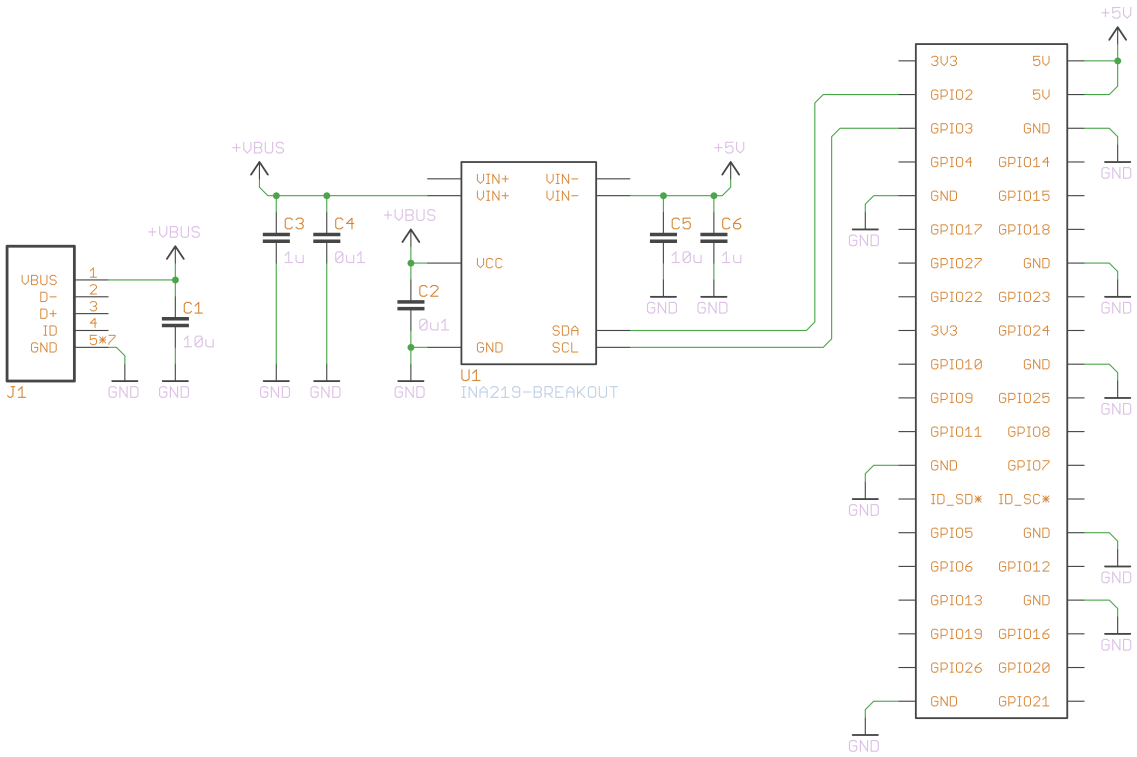

Figure 1: Hardware schematics

Let's start off with the power connector. We chose a micro USB connector because this resembles the Pi's. The placement of the connector is as close to the original USB connector so this gives the most compatibility. As current measurement IC we chose the INA219 chip by Texas Instruments. This chip has a very high measurement accuracy (max. 0.5%) and quick sampling time (0.5ms). It also features an I2C interface which is very suitable for the Raspberry Pi, since it has no internal ADCs. We have also implemented some simple filtering with the capacitors on the input and output of the current sensoring IC. This makes for a smooth measurement without too much ripple in the output data. And last but not least there are 2x20 pins GPIO pin headers for the connection with the Raspberry Pi.

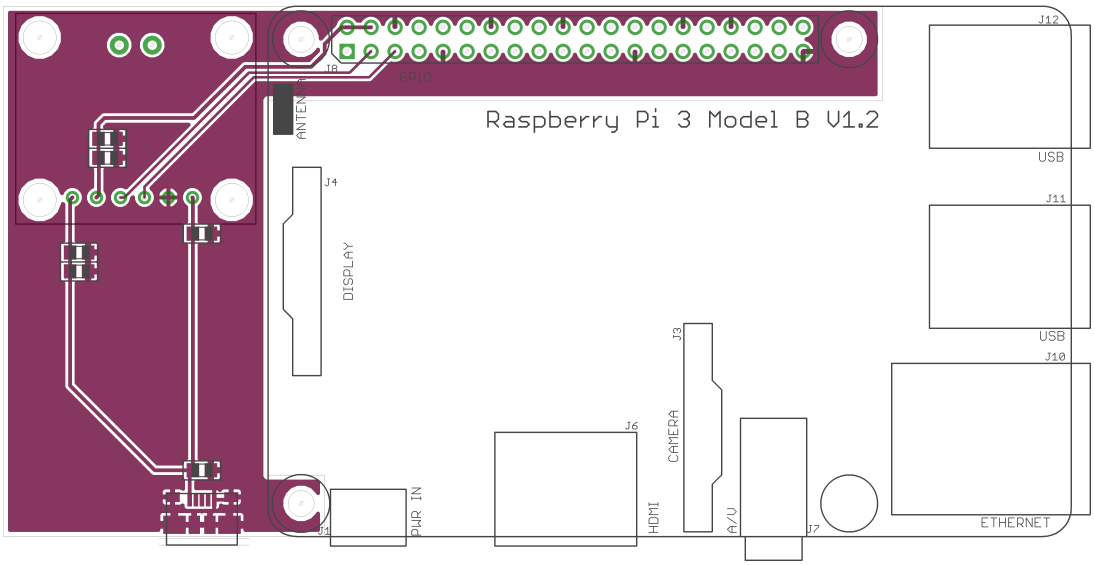

Figure 2: Hardware board

-Software

The software for getting the required data consists of the following components:

- I2C library
- Prometheus Client
- Psutil
- Filtering

The software uses the I2C library from Adafruit. This means that no further code is neccesary for using this communication channel. Having imported the 'INA219' library, we are able to create a variable using the INA219 constructor. This way, we have an instance which can be used to communicate directly.
Aside from the power measurement data, we have also implemented some CPU an RAM diagnostics. By importing Psutil we can achieve these and send them through.
Once the data is acquired, the software can start to do some processing. We know that the measurement interval is not infinitesmal, so there'll have to be some sort of smoothing and calculating. If not so, the sampling speed

could turn out lower than changes in the physical quantity, which leads to the well-known Nyquist theorem that is not met. So, we implemented a filter.
We have created three different types of filtering. One saves the gathered data in an array, moves them one to the left, appends the new value at the right and calculates the average (running average method). Optionally, the new measured value can be replaced by the calculated average value, which gives a FIR-filter implementation method, leading to a faster converging algorithm by reducing overshoot. The third algorithm is a simpler implementation of a running average and calculates the weight of the previous average value, adds this up to the new value and calculates the average. All these methods run in an infinite loop.
Currently, the third method is used. This method, however, could be further examined and implemented with inclusion of the time interval constant to determine the weight value. This way an even more exact output can be generated which is more reliable.
As soon as the data is processed, it can be sent to the next part. This is where Prometheus comes in. Because Prometheus does not run the software itself, and so does not re-run the software multiple times, the connection with the Prometheus Client can be run within the infinite loop. And so it does. Every 'interval' time, all data is sent to the Prometheus Client via its function. The HTTP-server that has been set up is now used as a server to keep a bunch of data present and presented. The data will be converted and then sent to the server. The server is running on port 8000 and can be accessed by the Prometheus Server (discussed in chapter Prometheus).

The code is as follows:

metric_INA219.py:

```
#!/usr/bin/env python

from __future__ import division
from prometheus_client import start_http_server, Summary, Gauge

from ina219 import INA219, DeviceRangeError
from subprocess import PIPE, Popen
import psutil
import time
```

```
class Filtervalue:
  def __init__(self, initialvalue):
    self.averagevalue = initialvalue

  # def __init__(self, initialvalue, sumcount):
    # self.sumcount = sumcount
    # self.lastvalues = []

    # for x in range(0, self.sumcount+1):  #0-100
      # self.lastvalues[x] = self.averagevalue


def updatevalue(self, newvalue, avgweight):  # three different
types of filter
  # sumofvalues = 0

  # self.lastvalues[self.sumcount] = self.newvalue #100

  # for x in range(0, self.sumcount):  #0-99
    # self.lastvalues[x] = self.lastvalues[x+1]
    # sumofvalues += self.lastvalues[x]

  # self.averagevalue = sumofvalues / self.sumcount  # filter
  no. 2

  # self.lastvalues[sumcount] = ((sumofvalues * (1-avgweight)) +
   (self.newvalue * (avgweight))) / sumcount  #extra filtering

  self.averagevalue = (self.averagevalue * (1-avgweight)) + (
  newvalue * avgweight)


collect_time = 1


inamodule = INA219(0.1, 2.0)
inamoduleg.configure(inamodule.RANGE_16V, inamodule.
GAIN_8_320MV, inamodule.ADC_12BIT, inamodule.ADC_12BIT)


REQUEST_TIME = Summary('request_processing_seconds',
```

```
'Time spent processing request')

prometheus_current = Gauge('energy_current', 'Filtered current
of the system')
prometheus_voltage = Gauge('energy_voltage', 'Filtered voltage
of the system')
prometheus_power = Gauge('energy_power', 'Filtered power of the
 system')

prometheus_cpu_percent = Gauge('cpu_percent', 'Filtered CPU
percentage')
prometheus_memory_mb = Gauge('memory_mb', 'Filtered RAM
consumption')


#initialize variables with Filtervalue constructor
filtered_current = Filtervalue(inamodule.current())
filtered_voltage = Filtervalue(inamodule.voltage() * 1000)
filtered_power = 0.0

filtered_cpu_percent = Filtervalue(psutil.cpu_percent())
filtered_memory_mb = Filtervalue(0.0)


if __name__ == '__main__':
  # Start server for Prometheus to scrape from
  start_http_server(8000)

while True:

  # Filter every /interval/ time
  start_time = time.time()
  while time.time() < start_time + collect_time:
    filtered_current.updatevalue(inamodule.current() , 0.1)
    filtered_voltage.updatevalue(inamodule.voltage() * 1000,
    0.1)

  filtered_cpu_percent.updatevalue(psutil.cpu_percent(), 0.1)
  ram = psutil.virtual_memory()
  filtered_memory_mb.updatevalue(ram.used / 1048576, 0.1)  # b
  to mb
```

```
filtered_power = filtered_current.averagevalue *
filtered_voltage.averagevalue / 1000

# post metrics
prometheus_current.set(filtered_current.averagevalue)
prometheus_voltage.set(filtered_voltage.averagevalue)
prometheus_power.set(filtered_power)

prometheus_cpu_percent.set(filtered_cpu_percent.averagevalue)
prometheus_memory_mb.set(filtered_memory_mb.averagevalue)
```

-Points of improvement

So our software works. However, there are still a few things that could be worked on. I will note them here:

- Power integration in t-domain to achieve energy
  What our Power monitoring does not do is calculation of the total energy consumption. As you might well know, energy is defined by the power multiplied by time. However, for changing values, energy is always calculated by integrating power over time. How can this be achieved? There are multiple answers to that question. One is to implement this is hardware. Design a PCB that filters the measured current and voltage, multiplies them and integrates them. This is difficult to do the right way, as a lot of designing skills come into view. One of the other options however, is an implementation in software. In this case it's necessary to have very fast data, very fast processing and very precise timing. I will explain more about timing in the next bullet point.

- Timestamps (snapshots) for gathered data.
  As I said in the previous bullet point, timing is very important. It's mandatory to have a very accurate measurement at a very specific, precise time. This is also called snapshots. When it is possible to create snapshots, a software implemented version of time integration can be done and precies results can be expected.

- Current measurement accuracy (not precision). It's very important to have a good current measurement. As we have seen with the Sparkfun breakout boards, probably because of the lack of designing skills of

the developers' side, these boards gave fluctuating outputs compared to one another. Their precision is high, but accuracy is low. In order to achieve representative measurement data, you will want to make a very solid current measurement. This however, is again a hardware thing. Consider predesigned alternatives with unpredictable outputs, or consult people from electronic design studies who know what they're talking about.

- Python software.
  So you'll notice that the Python software is not the fastest in the world. Communication with low level electronics is done by some GPIO libraries and their data is processed in Python. Because timing is so important, the software tries to do an infinite amount of measurements and calculations just to get the data right (the exact speed depends on the sampling speed in the INA219). The software now uses up a huge amount of CPU time just to do the filter calculations. It would be wise to make these less CPU-dependent or even better, as light as possible.