Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor Thesis

Fernando García Llorente

Implementation of a reading meter-bus data program and system, using a Raspberry Pi

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Fernando García Llorente

Implementation of a reading meter-bus data program and system, using a Raspberry Pi

Bachelor Thesis based on the examination and study
regulations for the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Supervising examiner: Prof. Dr.-Ing. Franz Schubert
Second examiner: M. Eng. Peter Lorenzen

Day of delivery March 20th 2017

Fernando García Llorente

Title of the Bachelor Thesis

Implementation of a reading meter-bus data program and system, using a Raspberry Pi

Keywords

Meter Bus, MBus, Raspberry Pi, Python, Wrapper, HTML, CSS, LibMBus, Library, PyCharm, C, Programming Language, RS232, Serial, Converter, GPIO, jQuery, JavaScript

Abstract

This thesis describes the first steps of the improvement of a heat transfer station in the CC4E building in Hamburg, in order to make this station a smarter station than it is.

To achieve that improvement in this thesis, the station is made accessible from any part of the building using low cost components, while this is done using the same programming language that is already used by the engineers of the center, as well as a program that allows the maximum flexibility and manageability possible to achieve such an outcome.

To achieve this flexibility and manageability using the same programming language, it’s created an own whole Wrapper that englobes up to 5 different programming languages.

Also, the hardware components are expensive, so a deep research in the functionality of the system is made to start developing an own hardware system that will allow to reduce the cost considerably.

This thesis is also made as the first step of a further development, so another’s future researchers of the C4DSI can continue improving the smart heat transfer station.

Fernando García Llorente

Thema der Bachelorarbeit

Implementierung eines Programmes zum Auslesen eines Meter-Bus mit einem Raspberry Pi.

Stichworte

Meter Bus, MBus, Raspberry Pi, Python, Wrapper, HTML, CSS, LibMBus, Library, PyCharm, C, Programming Language, RS232, Serial, Converter, GPIO, jQuery, JavaScript

Kurzzusammenfassung

Diese Arbeit beschreibt die ersten Schritte der Verbesserung einer Wärmeübertragungsstation im CC4E-Gebäude in Hamburg, um diese Station zu einer intelligenteren Station zu machen.

Um diese Verbesserung in dieser Arbeit zu erreichen, wird die Station von jedem Teil des Gebäudes mit kostengünstigen Komponenten zugänglich gemacht, während dies mit der gleichen Programmiersprache erfolgt, die bereits von den Ingenieuren des Zentrums verwendet wird. Um ein solches Ergebnis zu erreichen wird außerdem wird ein Programm verwendet, das eine maximale Flexibilität und Verwaltbarkeit ermöglicht.

Um diese Flexibilität und Verwaltbarkeit mit der gleichen Programmiersprache zu erreichen, entsteht ein eigener Wrapper, der 5 verschiedene Programmiersprachen umfasst.

Da die Hardwarekomponenten sehr teuer sind, wird die minimale Funktionalität ermittelt, um ein eigenes Hardwaresystem zu entwickeln, das es erlaubt, die Kosten erheblich zu reduzieren

Diese Thesis wird auch als erster Schritt Einer weiteren Entwicklung gemacht, so dass weitere zukünftige Forscher des C4DSI die intelligente Wärmeübertragungsstation weiter verbessern können.

Acknowledgment

First, and most important, I would like to thank the big support given by the M. Eng. Peter Lorenzen, who has followed all the steps made during the development of this thesis, as well as for his availability and patience to try to solve the doubts that I was arising along the research.

Also, I would like to thank the Prof. Dr.-Ing. Franz Schubert for giving me the chance to research in such an important research center as is the CC4E, with all the new and sophisticate equipment that it has.

I don’t want to forget to express my gratitude to the Informatic Engineer David González, who gave me a lot of useful information about the different programming languages and helped me learning a lot of these ones. Finally, I would like to thank to the entire C4DSI team for the long-term willingness to ask questions and for the warm welcome they have given to me, despite not knowing German. Specially to Ina, Sebastian and Matthias.

Content

Figures

Introduction

Continuing the explanation of the abstract, the aim of this thesis is to develop a smart station board that can be able to read a meter and then convert the data sent by this meter into data that can be read by a *Raspberry Pi*. Also, this data must be shown in any of the computers that englobes the network of the *CC4E*.

Also, the converter used to read the data can be replace by a new designed circuit, so the total circuit will be much cheaper and can reduce considerably the cost of future heat transfer stations.

MOTIVATIONS

The motivations that have led me to develop this project have not been few. The fact of being in one of the leading countries in engineering and one of the largest and most advanced cities, such as Hamburg, is motivation enough.

In addition, the opportunity to develop my research at the *Hamburg Energy Efficiency and Renewable Energy Competence Center*, which belongs to the University "*Hochschule für Angewandte Wissenschaften Hamburg*" (*HAW*), makes it very exciting and profitable to research with good engineers at my side and in very good facilities. Perfect conditions to research.

The barrier of the language and the fact that engineers here are specialists in different branches of engineering than mine is just another motivation to continue improving in my preparation as an engineer. The fact that I can help in a big project and the possibility that my small research project will help the future students that will realize their thesis here, is the best preparation to become a good engineer in the future.

Finally, the fact that I had no previous experience with the protocols and the languages that I use in my project, makes it more difficult but also made me more motivated.

APPROACH

To do the project as efficient as possible, 3 mains goals are exposed. Three main steps of the research that will help to do a better research. These three goals are the main structure of the thesis.

GOAL 1

The first goal is to implement and test all the new components of the system, in order to read the data that comes from the meter in a monitor, all using a meter bus library. A lot of research is needed in this goal to start correctly the project: knowing how the *RPi*, *MBus* protocol and *RS232* protocol works; installing the software needed in the *RPi* and computer; and connecting, testing, and looking for as much information as possible in order to implement it in the best way possible.

GOAL 2

Once the first goal is done, the main goal comes. This is the main goal because a lot of work is needed. With the system working correctly, creating and implementing a *Python Wrapper* in the *Raspberry Pi* that uses the same library to read the data sent from the meter and do whatever is wanted with the data, is the most difficult and main work of all the project. Also, some preparatory work is needed: developing the skills in *Python*, *JavaScript*, *HTML*, *CSS*; studying and understanding how the library works and prepare correctly the *Raspberry Pi* to work with the implemented *Wrapper*.

GOAL 3

With Goal 1 and Goal 2 complete, the main work of the project is done. However, some parts of the system can be improved in order to save some money in the future. Also, using the *USB* port to read the data is not the best solution. Replacing successfully the *USB-RS232* converter by another designed circuit is the third goal of this thesis.

To do that, the *RPi* has some *GPIO* Pins that can be used instead of the *USB* ports, and a new designed circuit in between is needed. Building, testing and implementing a new designed circuit are one of the main tasks of the implementation of the circuit. Also, configuring the *Raspberry Pi* to use correctly the *GPIO* pins to send and receive the information is another subtask of this last goal.

TECHNICAL BASICS

Some technical basics are explained in this part of the project. In the figure 1 it is described the main diagram of the goal 1, which is explained previously, with the images of the real components used in this project and with the protocols used in the process.

The meter used is an "Integral-V UltraLite" model made by *Allmess*, whose datasheet can be found in the reference 1. The physical data is measured by this meter and sent to the converter using the *Meter Bus* protocol. This data is converted into data that can be sent using the *RS232* protocol by the "MBus 10 Converter" made by *TechBase*, whose datasheet can be found in the reference 2.

This data is received by the *Raspberry Pi* using a RS232-USB Converter. This *RPi* uses a monitor connected through an *HDMI* cable in order to be able to see the shell command lines of the *RPi*.

Figure 1 – Main diagram of Goal 1. [1] [2] [3] [4] [5]

Meter Bus Protocol

MBus is a standard that allows reading the data of certain types of meters. This protocol is made for using it with only two wires, which makes it very profitable.

When the information is requested from the meter, it delivers the data collected to a common master, such as in this case is the *Raspberry Pi* which is connected to all the meters in the same building.

The wired *Meter Bus* has a bus topology where the common master mentioned above can communicate with up to 250 slaves. This communication of the master with the slaves happens thanks to some voltage changes, from 24V to 36V, while the communication of the slave with the master happens through current changes, from 1mA to 1.5mA.

Meter Bus devices can use a speed between 300 bauds and 38400 bauds. Most meters use 2400 bauds, as is the case of the meter used in this project.

Wired *M-Bus* differentiates between five different frame types:

- SND_NKE (send link reset)
- SND_UD (send user data)
- REQ_UD1 (request user data 1)
- REQ_UD2 (request user data 2)
- RSP_UD (respond user data)

But the most common message is the request/response service in which the common master sends a *REQ_UD2* frame addressed to a specific slave, such as a meter, and it responds with the *RSP_UD* message. This last message is the one containing the measurement data of the moment in which the request was made.
The *REQ_UD2* frame contains only the primary address of the slave to be read, which occupies 1 byte. This main address is explained below.

Primary Addressing

The primary address occupies only a single byte, and that allows values between 0 and 255.
Addresses from 1 to 250 are assigned to the slaves.

The other addresses have special purposes:

- 0 is used by unconfigured slaves.
- 251 and 252 are reserved.
- 253 indicates that secondary addressing is being used.
- 254 and 255 are broadcast addresses.

[6] [7]

Rs-232

“In telecommunications, *RS-232* is a standard for serial data transmission. It is commonly used in computer serial ports. The standard defines the electrical characteristics and the time of the signals, the meaning of the signals, and the physical size and pinout of the connectors.” [8]

In the figure 2 it’s possible to see how the Pin Outs are distributed physically, and in the table 1 every pin is described.

RS-232 Pin Outs (DB-9)

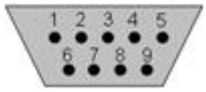


Figure 2 - A male DB-9 connector viewed from the front. [9]

DTE Pin Assignment (DB-9)			**DCE Pin Assignment (DB-9)**		
1	DCD	Data Carrier Detect	**1**	DCD	Data Carrier Detect
2	RxD	Receive Data	**2**	TxD	Transmit Data
3	TxD	Transmit Data	**3**	RxD	Receive Data
4	DTR	Data Terminal Ready	**4**	DSR	Data Set Ready
5	GND	Ground (Signal)	**5**	GND	Ground (Signal)
6	DSR	Data Set Ready	**6**	DTR	Data Terminal Ready
7	RTS	Request to Send	**7**	CTS	Clear to Send
8	CTS	Clear to Send	**8**	RTS	Request to Send
9	RI	Ring Indicator	**9**	RI	Ring Indicator

Table 1 – DB-9 Pin Assignment [9]

Raspberry Pi

The *RPi* is a board computer with the size of a wallet. All models have a *Broadcom System* on a chip (*SoC*), which includes a central processing unit (*CPU*) compatible with *ARM* and a graphics processing unit on chip.
The RPi model used in this project is the *Raspberry Pi* 1 Model B revision 1.2 which has 512 MB of *Ram*, two *USB* ports and 100 MB of *Ethernet* port.

[10]

In the figure 3 it is described every component of the hardware of the *Raspberry Pi*.

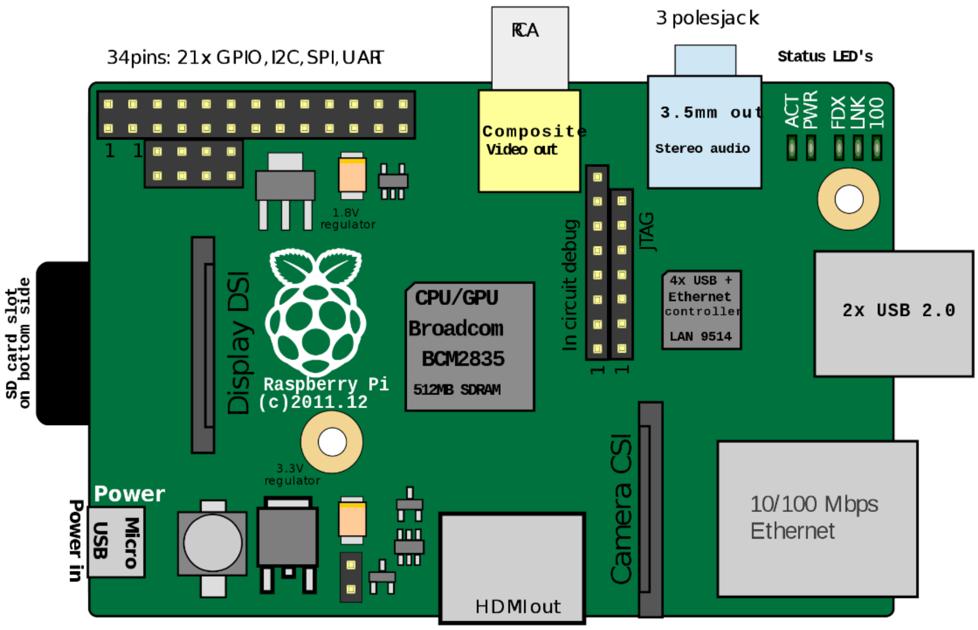


Figure 3 – Raspberry Pi 1 Model B revision 1.2. [11]

GPIO

General-Purpose Input/Output is a generic pin on a computer board, in this case on the *Raspberry Pi*; whose behavior is controllable by the user.

GPIO pins do not have a predefined purpose and are not used by default.

[12]

In the figure 4 it's described every pin of the *RPi* that it is used in this project.

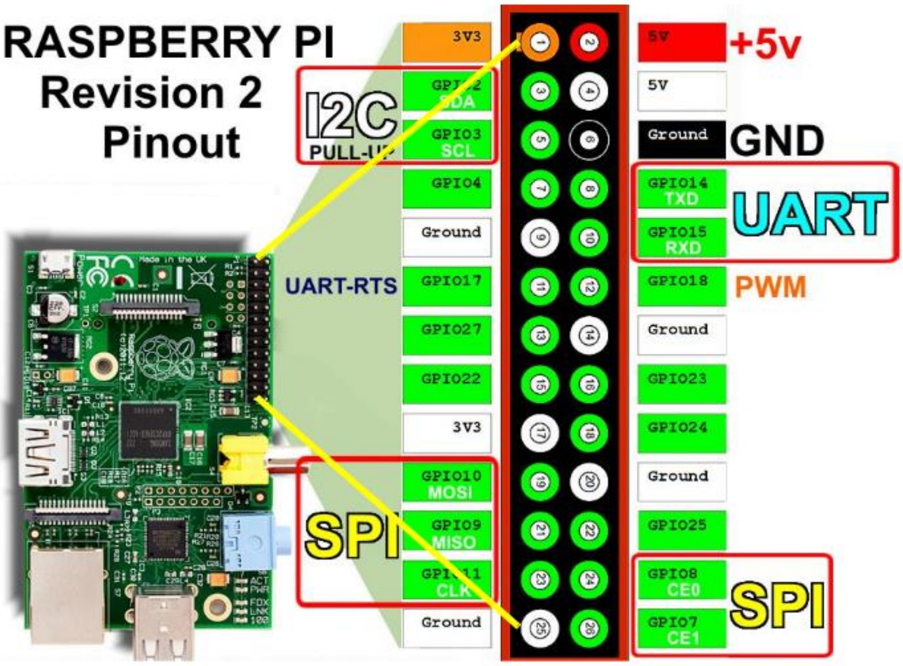


Figure 4 – GPIO RPi Pins. [13]

Goal Diagrams

Some diagrams are made in order to understand better the three main goals of this thesis, and in order to create a more detailed explanation.

First of all, in the figure 5 is represented the diagram of how the heat transfer station was working before the start of this thesis. The meter and the *MBus* are the same models used in this project, whose datasheets can be found in the references 1 and 2. As explained before this meter sends the physical data measured using MBus protocol, using voltage levels between 24 and 36 volts. It also uses current levels between 1 and 1.5 milliamps. This data is received by the converter in order to send the data using RS232 that uses voltage levels between 3 and 15 volts. In this first diagram is the *PLC* the one who reads the data and send it to the computer through *TCP*.

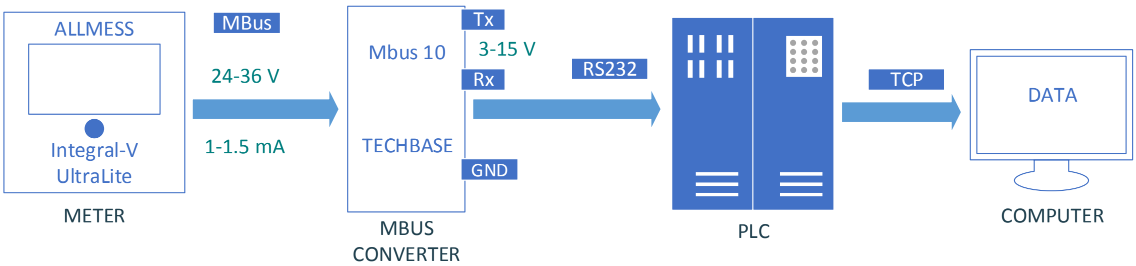


Figure 5 – Detailed diagram of the previous heat meter station.

In the figure 6 is represented the diagram of the goal 1. In this goal the *PLC* is replaced for the *Raspberry Pi* and a *RS232* converter needed to connect the *RPi* to the converter. This connection is made through a serial *RS232* cable, and using one of the 2 *USB* ports of the *RPi*. The *RPi* needs a monitor in order to see the shell command line of the *Linux* software installed on it, and this connection is made through and *HDMI* cable.
As it can be seen at the top of the *RPi* of this diagram, this *Linux* software uses a *Meter Bus* library called *libmbus*, that makes the reading of the *MBus* data possible.

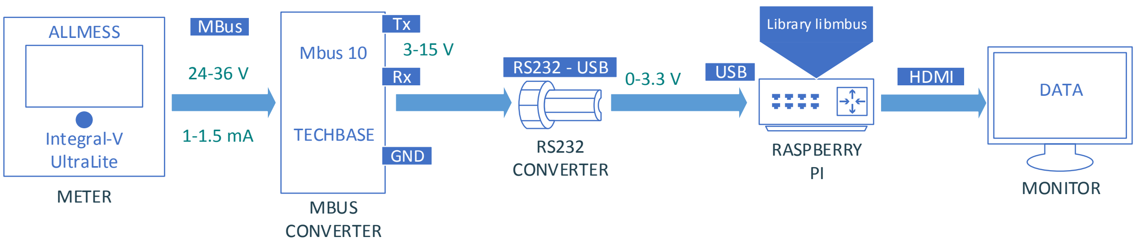


Figure 6 – Main detailed diagram of Goal 1.

In the figure 7 the diagram of the goal 2 is represented. The only main difference between this diagram and the diagram of the figure 6 is the implementation of a *Python Wrapper*, that will allow not only read the *MBus* data but also make possible to do more things with this data apart from reading it.

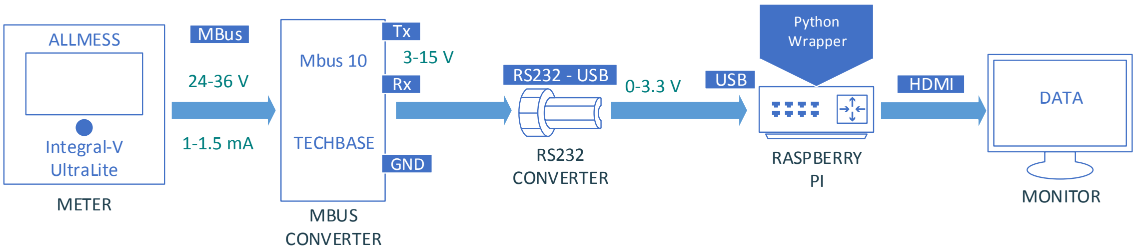


Figure 7 – Main detailed diagram of Goal 2.

And finally, the goal 3 diagram is represented in the figure 8. In this diagram the connection between the converter and the *Raspberry Pi* is completely changed. Instead of using the *USB* port of the *RPi*, the *GPIO* pins are used, and a little circuit between the converter and the *GPIO* pins is developed. This circuit allow to convert the high-level voltages that comes out from the converter to voltages levels that the *GPIO* pins of the *Raspberry Pi* allows.

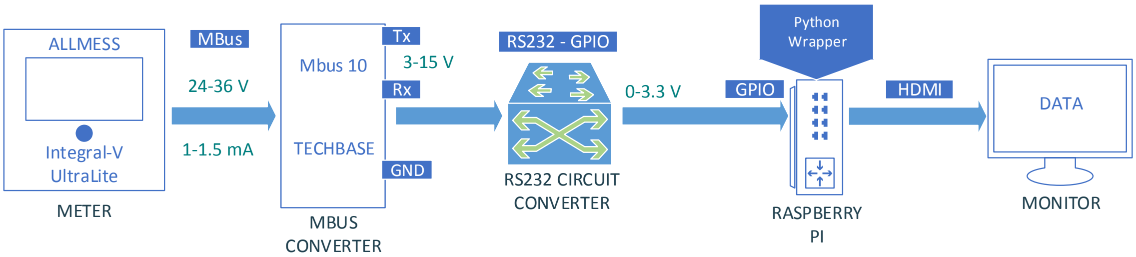


Figure 8 – Main detailed diagram of Goal 3.

Preparatory work

To configure the *Raspberry Pi,* you need to do some changes in to the software files but first of all you need the necessary components to do that. These components are shown in the figure 9.

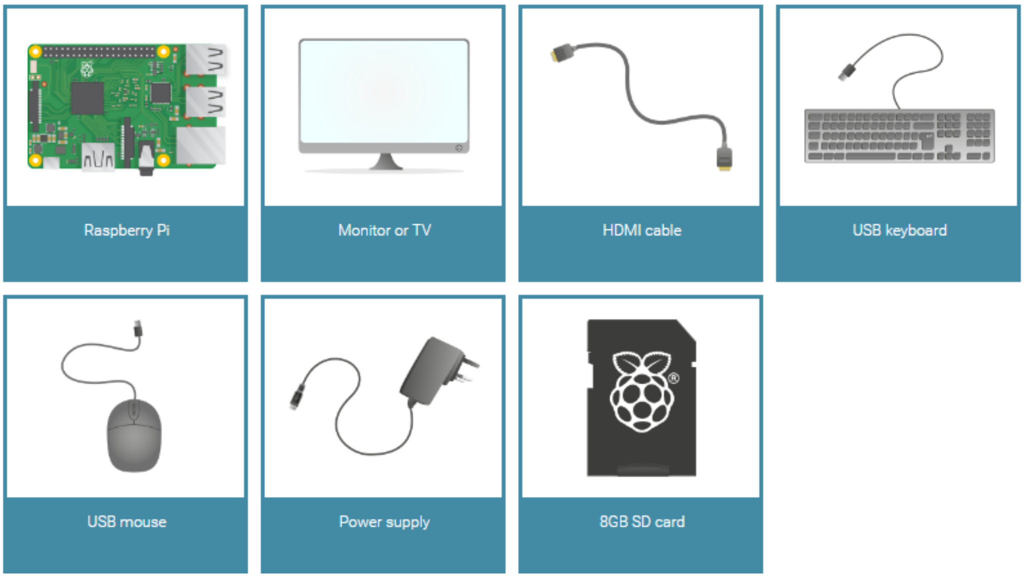


Figure 9 – Main Components to run a RPi [14]

OPERATING SYSTEM

The recommended operating system to use with the *Raspberry Pi* is called *Raspbian*, which is a version of *GNU/Linux*, designed specifically to work well with the *Raspberry Pi*.

In the figure 10 there are 4 screenshots made during the installation process of the *Raspberry Pi* used for this project. After waiting some minutes, the *Raspberry Pi* is ready to start.

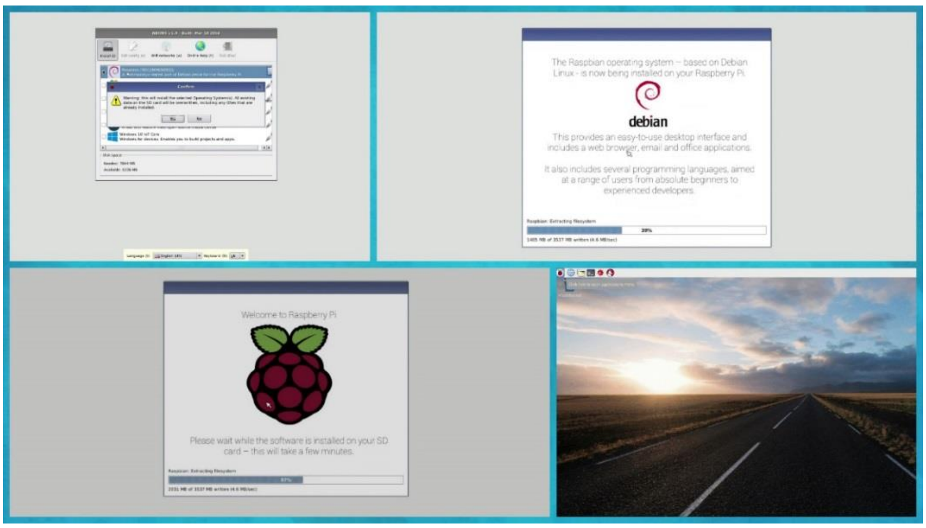


Figure 10 – Installing Raspbian

CONFIGURING INTERNET

Using an *Ethernet* cable, it is possible to connect the *Raspberry Pi* to the network of the workplace: the *CC4E*. But to do that, some steps are needed first.

Using the shell command line, or terminal, 3 files of the system must be modified. First of all, the file "*interfaces*", located in: "*/etc/network/*", using the command: "*sudo nano /etc/network/interfaces*". See figure 11.

```
pi@raspberrypi: ~
File  Edit  Tabs  Help
pi@raspberrypi:~ $ cat /etc/network/interfaces
# interfaces(5) file used by ifup(8) and ifdown(8)

# Please note that this file is written to be used with dhcpcd
# For static IP, consult /etc/dhcpcd.conf and 'man dhcpcd.conf'

# Include files from /etc/network/interfaces.d:
#source-directory /etc/network/interfaces.d

auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
address 141.22.122.233
netmask 255.255.255.0
gateway 141.22.122.1
dns-nameservers 141.22.192.100 141.22.192.101
```

Figure 11 - etc/network/interfaces

With this file, the *IP* address, the netmask, the gateway, and the *DNS* are set in order connect to the network of the building. So, this *RPi* is the only device in the whole center with that *IP* address.

Then the file "*resolv.conf*" located in the folder "*/etc*" must be modified using the same command: "*sudo nano /etc/resolv.conf*" so the file looks like the file in the figure 12.

```
pi@raspberrypi:~ $ cat /etc/resolv.conf
# Generated by resolvconf
search 141.22.192.101
nameserver 141.22.192.100
nameserver 141.22.192.101
```

Figure 12 – etc/resolv.conf

Where the nameserver 141.22.192.100 is the preferred *DNS* server, and the nameserver 141.22.192.101 is the alternate *DNS* server.

Then, a restart of the network must be done to apply correctly all the changes, using the command: "*sudo /etc/init.d/networking restart*". See figure 13.

```
pi@raspberrypi:~ $ sudo /etc/init.d/networking restart
[....] Restarting networking (via systemctl): networking.serviceWarning: Unit file of networking.servic
e changed on disk, 'systemctl daemon-reload' recommended.
. ok
```

Figure 13 – etc/init.d/networking restart

INSTALLING LIBRARIES AND UPDATING

The function of the *Raspberry Pi* is to read the information using the *Meter Bus* protocol. So, a library for this protocol is needed to read the information coming from the meter. This library has a lot of functions and files that are needed in order to extract the data correctly from the meter.

After setting up the internet connection, updating the packages must be done using the command "*sudo apt-get update*". This command downloads the package lists from the repositories and updates them to get information on the newest versions of packages and their dependencies. The screenshot of the process done during this process is shown in the figure 14.

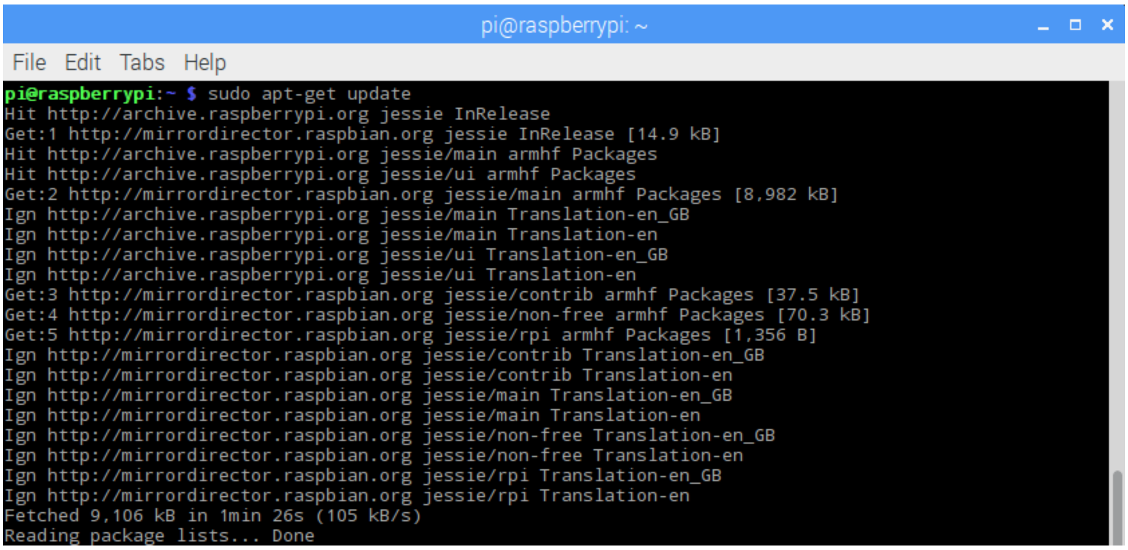


Figure 14 – sudo apt-get update

Also, an upgrade of the packages is highly recommended, using the command "*sudo apt-get upgrade*". Which this one lasts for almost one hour in the *Raspberry Pi* 1 Model B. See figure 15.

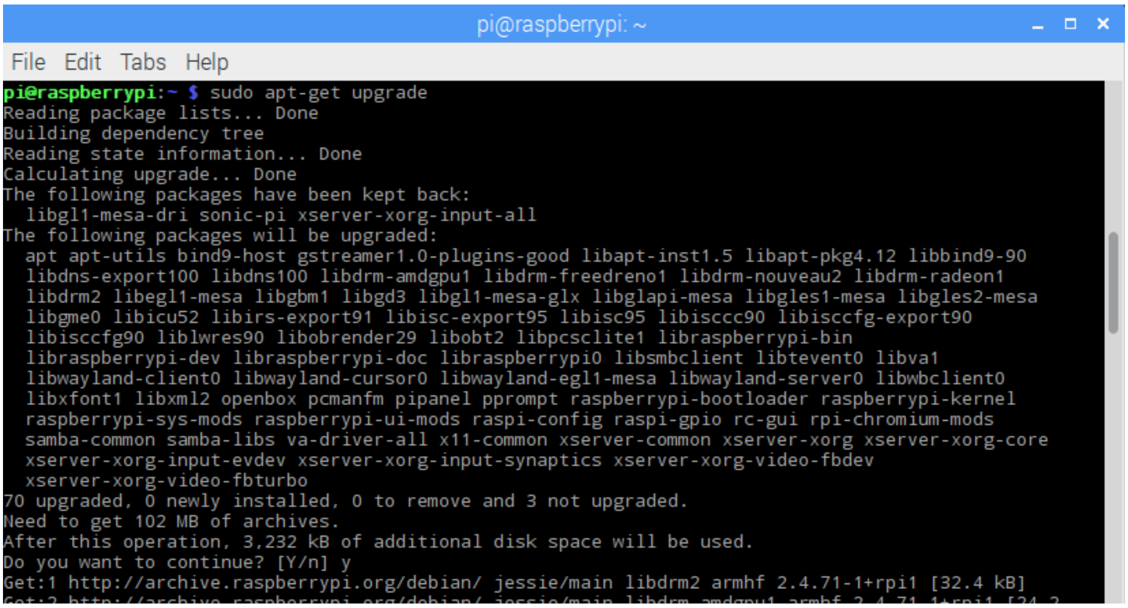


Figure 15 – sudo apt-get upgrade

Then, some steps are needed to install correctly the library. The library that is used in this project is the one called *libmbus*, developed by *rSCADA*. The description of this library and how it is installed in the RPi of this project, is explained below.

libmbus

This is an open source library. This kind of software is free distributed and developed. It is focused more on the practical benefit, the access to the source code. It's possible to modify the source of the software without license restrictions.

The main function of the *libmbus* library is to perform the communication with the *Meter Bus* slaves and to encode and decode *Meter Bus* data. The last version of the library developed by *rSCADA* dates from 2012 and allows the connection through *Meter Bus* gateways with *TCP* and *Serial* interfaces. In this project, only the *Serial* interface is used.

One of the main reasons why this library is the chosen one is the fact that it also presents the data in an easy *XML* format. This characteristic allows to simplify a lot the delivery of the *MBus* data in the *Raspberry Pi*, and it's easier to work with that kind of format.

[15]

The commands used to install correctly the *libmbus* library in to the *Raspberry Pi* of this project are the next ones:

- "*wget https://github.com/rscada/libmbus/archive/master.zip*", this command downloads the zip file containing the library in the actual folder. See figures 16 and 17.

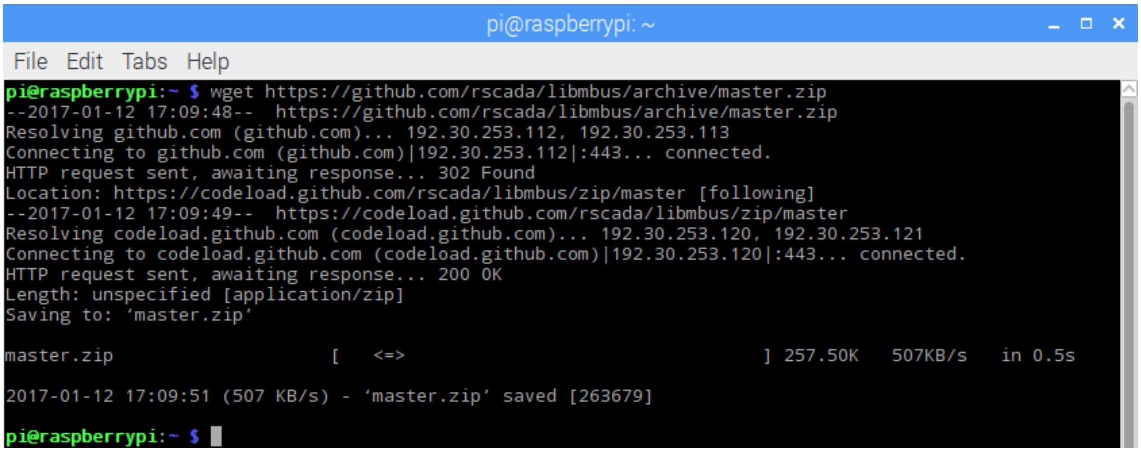


Figure 16 – wget

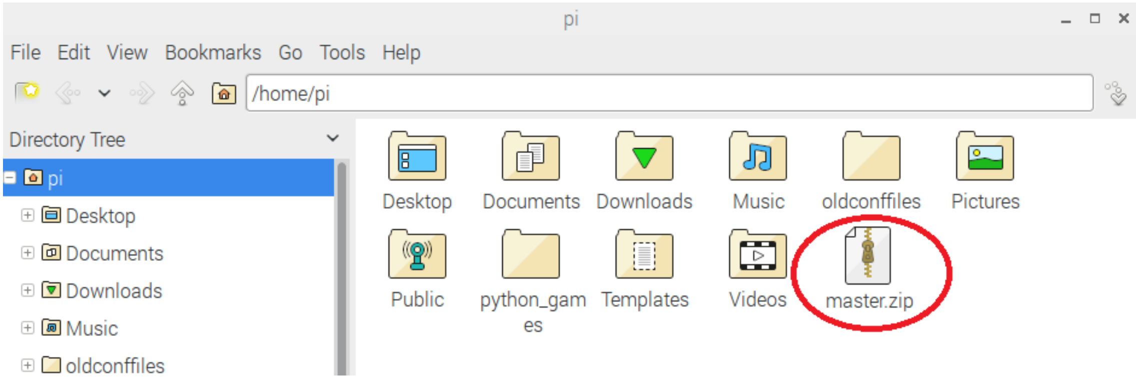


Figure 17 – wget file

- The command "*unzip master.zip*" creates a zip file in the actual folder. See figures 18 and 19.

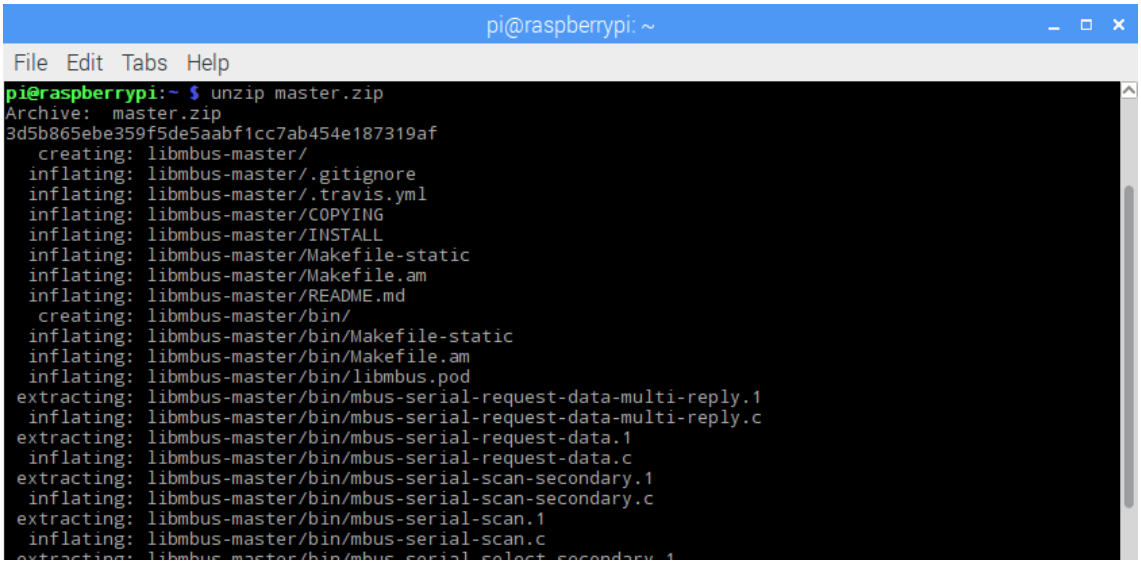


Figure 18 – unzip

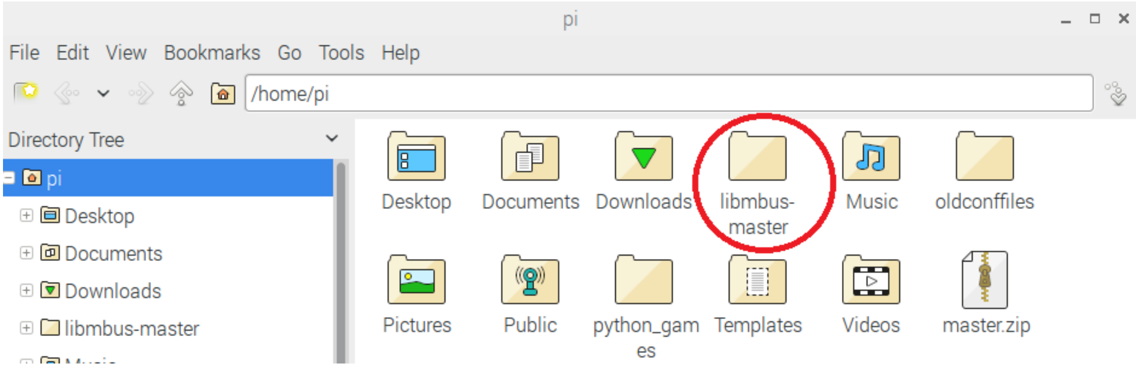


Figure 19 – unzip file

- "sudo apt-get install libtool automake", see figure 20.

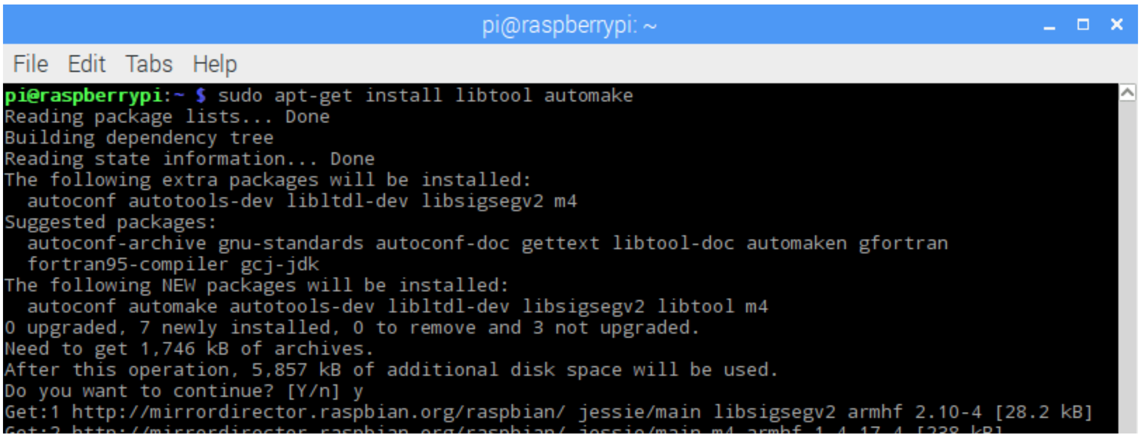


Figure 20 – sudo apt-get install libtool automake

- Then, inside the folder *libmbus-master*, using "*cd libmbus-master/*", the command "*autoheader && aclocal && libtoolize –ltdl –copy –force && auto make —add-missing –copy && autoconf*" is executed. This commands includes some commands in only one. The screenshot of the process done during this project is shown in the figure 21.

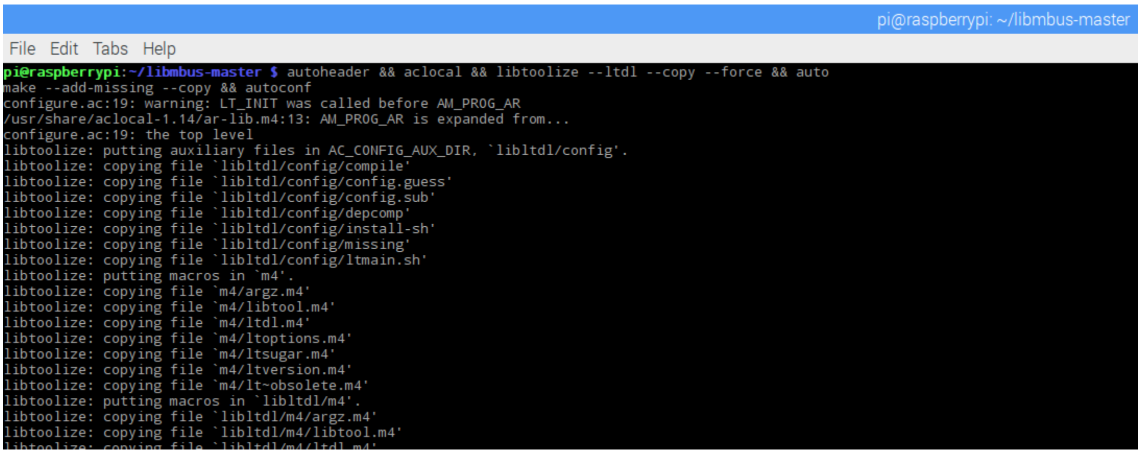


Figure 21 – autoheader && aclocal && libtoolize

- After this, the library has to be configured. To do that, and inside the folder *libmbus-master*, it is run the command "*/configure*", and some files are created inside the folder. This command use to require like half an hour, the process done during this process is shown in the figures 22, 23 and 24.

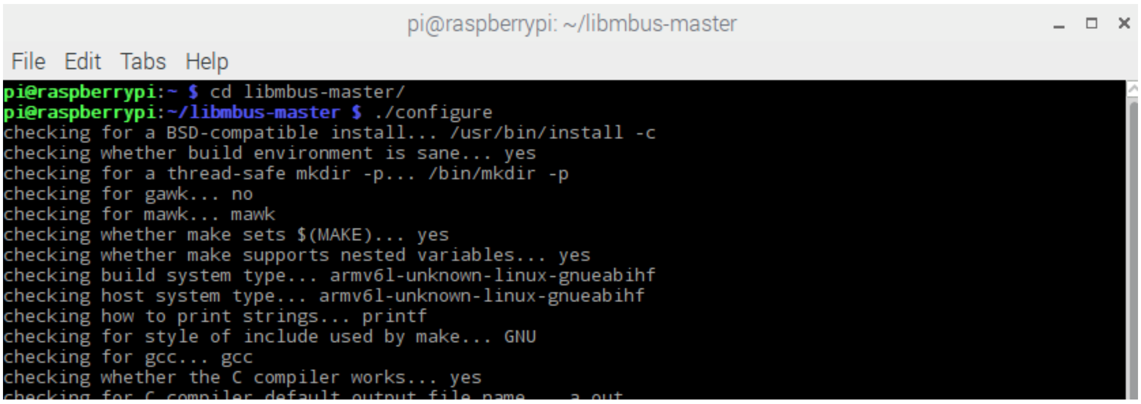


Figure 22 – configure

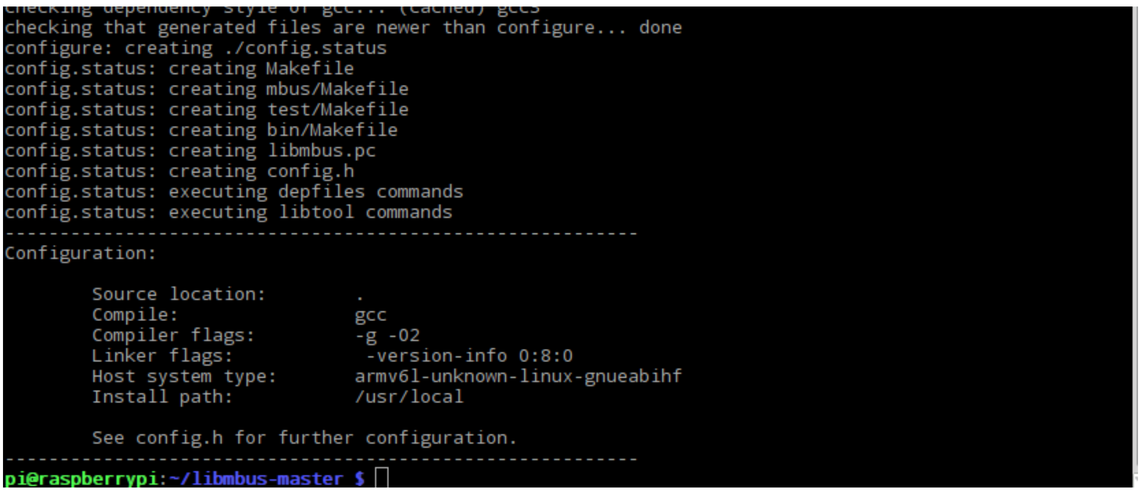


Figure 23 – configure (2)

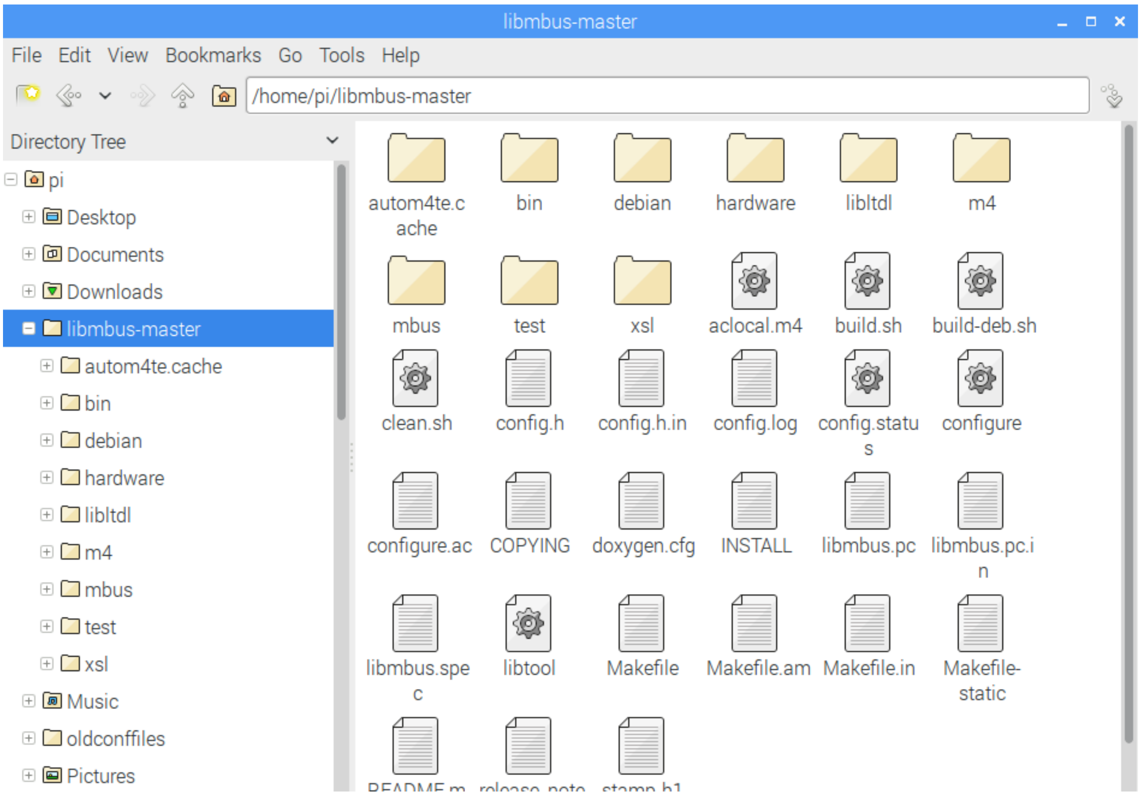


Figure 24 – configure files

- Finally, is time to install the library. The commands "*sudo make*" and "*sudo make install*" must be executed in this order, inside the folder *libmbus-master*. These commands also require a "large" amount of time. The execution process done during this project is shown in the figure 25, 26 and 27.

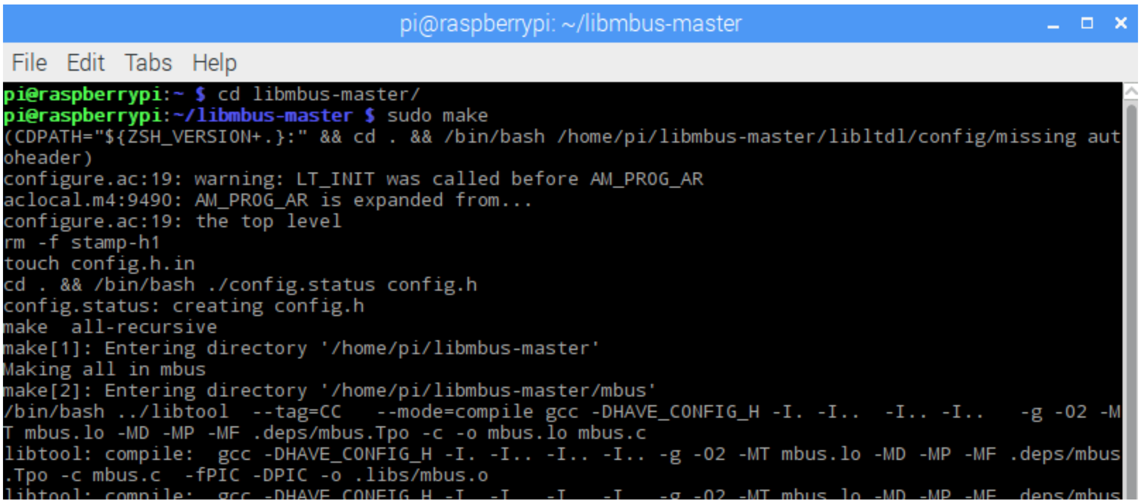


Figure 25 – sudo make

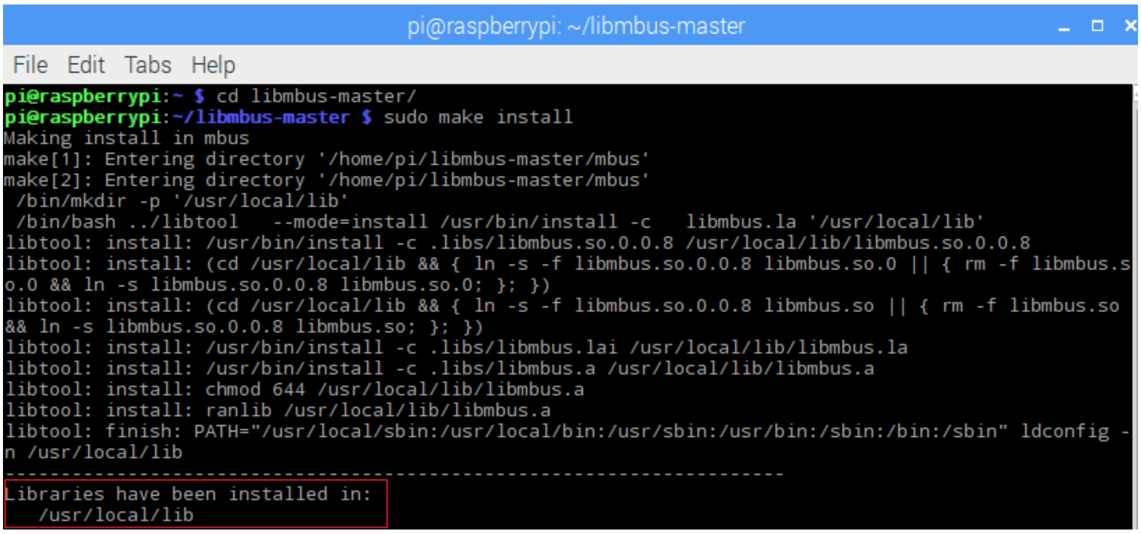


Figure 26 – sudo make install

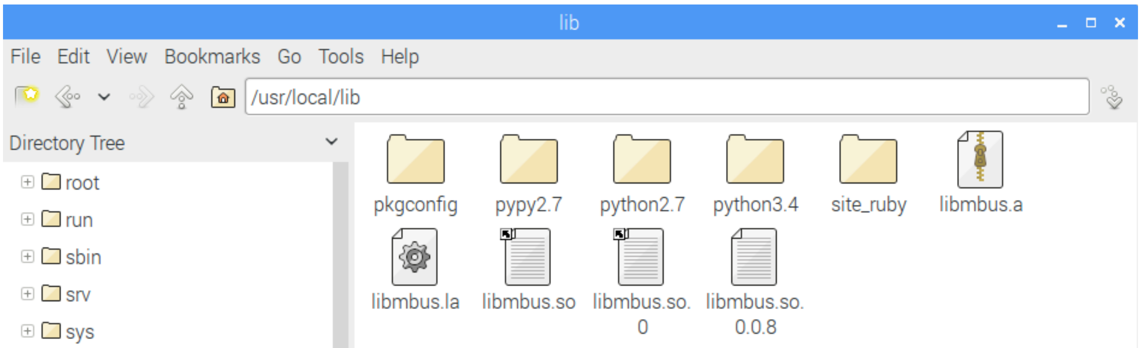


Figure 27 – Location of the installed library

But, in the case of this project one problem is faced. At the time to run the library, this one problem is found: "*Error while loading shared libraries: libmbus.so.0: cannot open shared object file: No such file or directory*". That means that the directory where the *Raspberry Pi is* trying to find the file is not correct. To solve this problem, only the command "*sudo ln -s /usr/local/lib/libmbus.so0 /usr/lib/libmbus.so.0*" is needed. See figure 28.

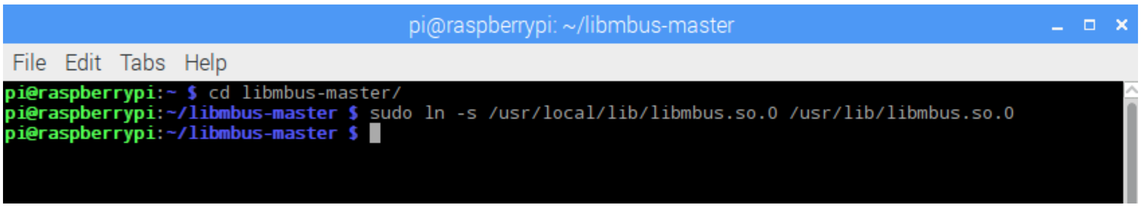


Figure 28 – sudo ln -s

This command creates a link between the actual position of the file and the position where the file is expected to be.

[16] [17] [18]

SERIAL PORT CONNECTION

“The serial port is a low-level way to send data between the Raspberry Pi and another computer.” [19]

First of all, it has to be proven that there is a connection between the *RPi* and a *Computer* using the *USB-RS232* converter. The figure 29 shows the *USB-RS232* converter diagram.

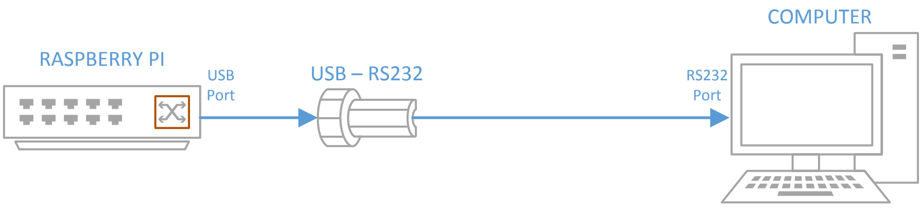


Figure 29 – USB-RS232 converter diagram.

The *USB-RS232* converter used in this project is the *Digitus USB* 1.1. Its datasheet can be found in the web page linked in the reference 3. See Figure 30.

Figure 30 – Digitus USB 1.1 serial converter. [3]

In the RPi, it is needed to be a member of the *dialout group* to access this port. To check this the command “*ls -l /dev/ttyUSB0*” is used. See figure 31.

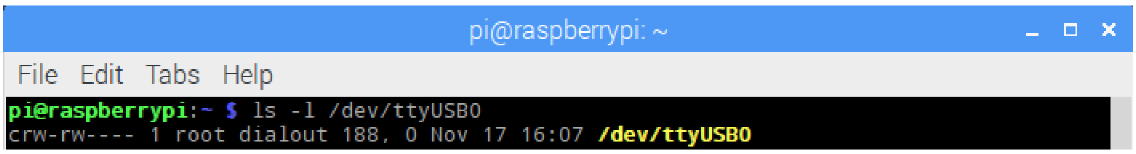


Figure 31 - ls -l /dev/ttyUSB0.

Where c means character device, the *root* can *'read,write'*, the *dialout group* can *'read,write'* and everyone else cannot access it.

In this project the *Terminal Emulation Program* called *GNU Screen* will be used, and before using it, it must be installed with the command:

```
sudo apt-get install screen
```

And then it can be executed using the command:

```
screen /dev/ttyUSB0 9600
```

With "*/dev/ttyUSB0*" the port name is indicated, and with "*9600*" the baud rate between them. At the same time, in the computer, the program called *Tera Term* is run in this project, that allows to send information between the computer and the *Raspberry Pi*. See figure 32.

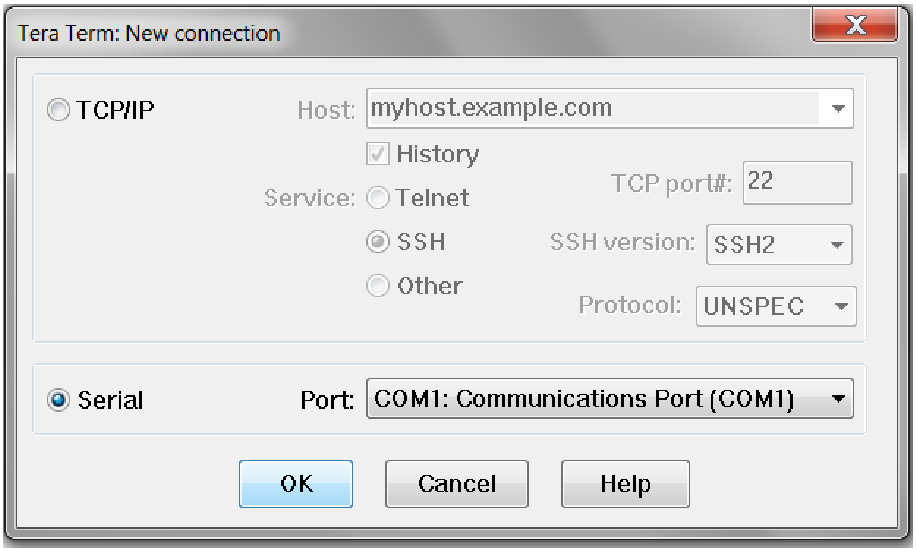


Figure 32 – New connection in Tera Term.

Then, after selecting the correct *COM* Port, the connection should be configured. In the case of the test of this project the connection is configured like it's shown in the figure 33 and 34.

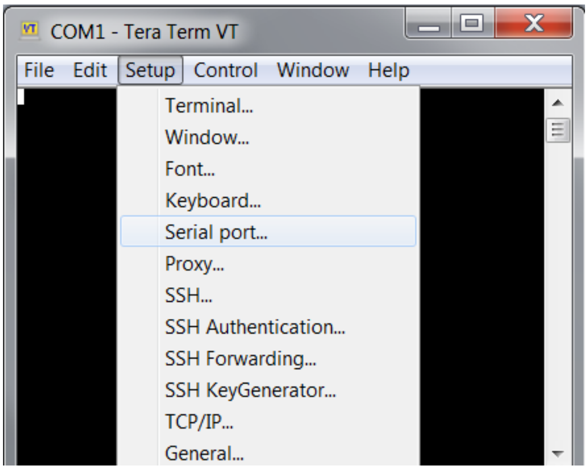


Figure 33 – Configuring connection in Tera Term.

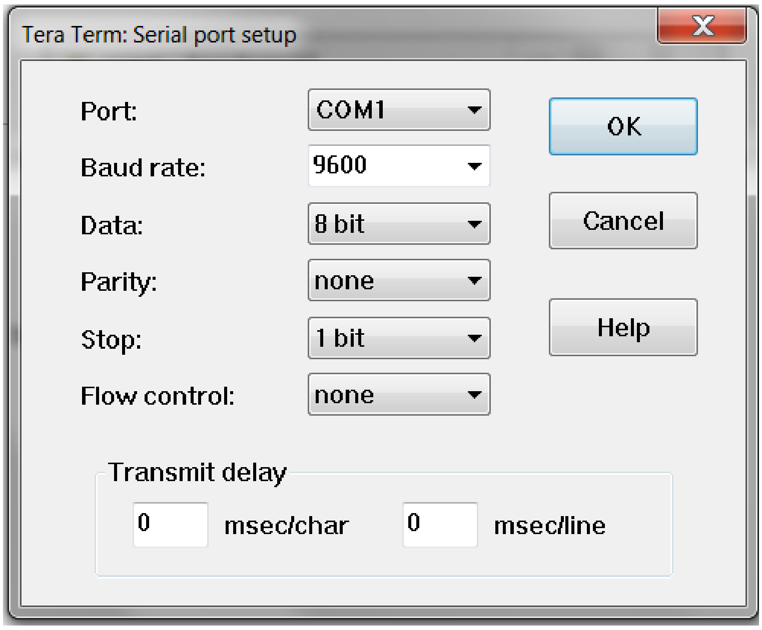


Figure 34 – Configuring the serial port connection in Tera Term.

This is the configuration that the both Raspberry Pi have, so in the Tera Term program it must be the same.

Then, when something is written in the *Tera* terminal it is shown in the *RPi* terminal, that is using the program *GNU Screen*. See figure 35.

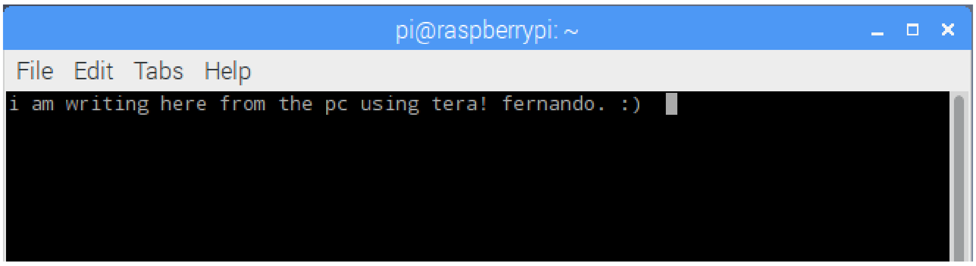


Figure 35 – Message received in RPi terminal from Tera Term.

And of course, if something is written in the *RPi* terminal, it's shown in the *Tera Term* terminal. See figure 36.

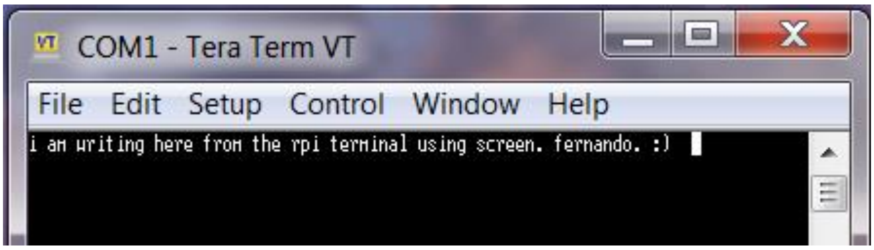


Figure 36 - Message received in Tera Term from RPi Terminal.

[20]

RECEIVING FIRST MBUS DATA

In this part of the project the first connection between all components of the goal 2 is made, as it can be shown in the figure 37. The meter is connected to the converter while this converter is fed by a power supply, and the converter is connected through a *RS232* cable to the *USB-RS232* converter. The *RPi* has connected the *USB-RS232* converter, the *Ethernet* cable, the power supply and of course the *SD* card that contains the software. The address of the meter used in this project is 0, because it is tested with an unconfigured meter. And as it was explained before, the unconfigured meters have address 0.

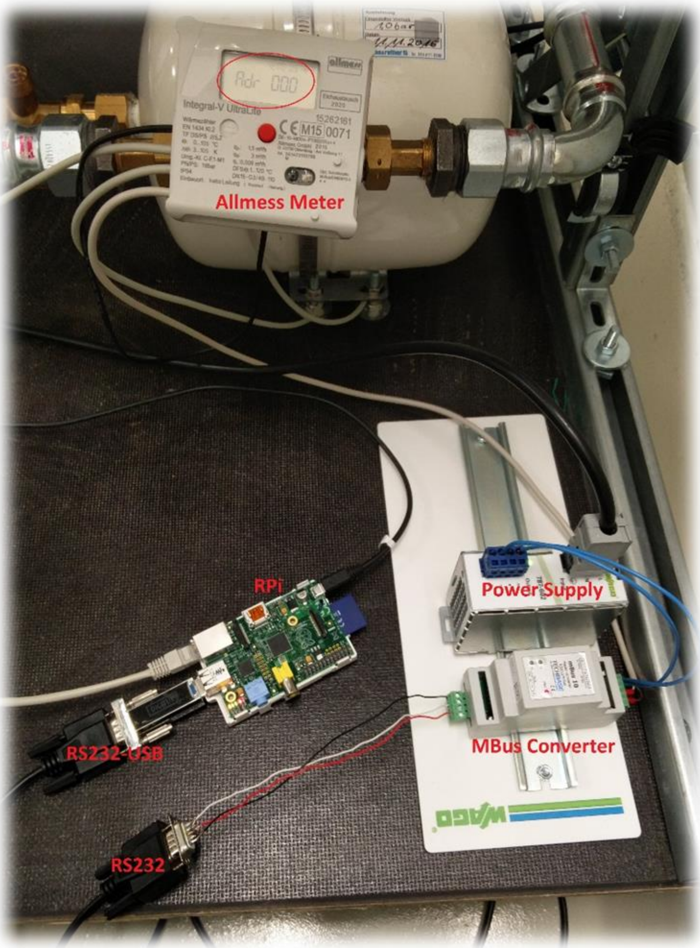


Figure 37 – First assembled system.

After connecting everything and double-checking that everything is correctly connected it can be tested. Is the time to check the results, the library that was installed before can be used. To do that the *PuTTY* program is used, a program that allow to access the shell command line of the *RPi* thanks to the *Ethernet* cable, as it's explained below.

With the command "*mbus-serial-scan*" typed on the shell command line of the *RPi* and the words "*-b 2400 /dev/ttyUSB0*" just after is it possible to get the information of how many meters are connected in the network. These last words mean that the scan is made over serial connection, the debug mode is deactivated, the baud rate is 2400 bauds and the connection is made over the serial port *ttyUSB0*.

This command activates some functions defined in the library that was installed before, and checks every address possible to find every slave connected to the network. From 0 to 250 address, a slave is found on the address 0, so all is correct because it's only connected one, and the meter used in this project is unconfigured. The execution done in this project is shown in the figure 38.

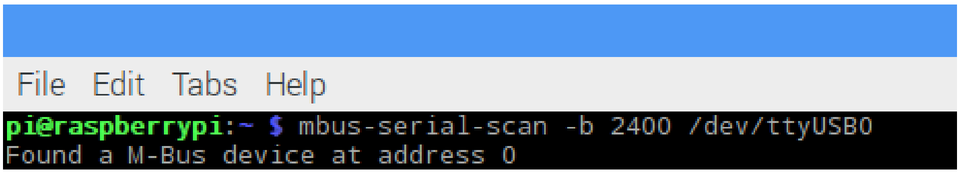


Figure 38 – mbus-serial-scan -b 2400 /dev/ttyUSB0.

Then, in order to receive the *Meter Bus* data, the command "*mbus-serial-request-data*" is used with the 2400 baud rate and the address of the *MBus* device "*-b 2400 /dev/ttyUSB0 0*", as it's shown in the figure 39.

[15]

```
File  Edit  Tabs  Help
pi@raspberrypi:~ $ mbus-serial-request-data -b 2400 /dev/ttyUSB0 0
<MBusData>

    <SlaveInformation>
        <Id>15262161</Id>
        <Manufacturer>ITR</Manufacturer>
        <Version>23</Version>
        <ProductName></ProductName>
        <Medium>Heat: Outlet</Medium>
        <AccessNumber>60</AccessNumber>
        <Status>00</Status>
        <Signature>0000</Signature>
    </SlaveInformation>

    <DataRecord id="0">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Fabrication number</Unit>
        <Value>15262161</Value>
        <Timestamp>2017-01-13T17:45:00</Timestamp>
    </DataRecord>

    <DataRecord id="1">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Energy (kWh)</Unit>
        <Value>177</Value>
        <Timestamp>2017-01-13T17:45:00</Timestamp>
    </DataRecord>

    <DataRecord id="2">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Volume (1e-2  m^3)</Unit>
        <Value>48052</Value>
        <Timestamp>2017-01-13T17:45:00</Timestamp>
    </DataRecord>

    <DataRecord id="3">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Power (100 W)</Unit>
        <Value>0</Value>
        <Timestamp>2017-01-13T17:45:00</Timestamp>
    </DataRecord>

    <DataRecord id="4">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Volume flow (m m^3/h)</Unit>
        <Value>0</Value>
        <Timestamp>2017-01-13T17:45:00</Timestamp>
    </DataRecord>

    <DataRecord id="5">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Flow temperature (1e-1 deg C)</Unit>
        <Value>210</Value>
        <Timestamp>2017-01-13T17:45:00</Timestamp>
    </DataRecord>
        <Unit>Volume flow (m m^3/h)</Unit>
        <Value>0</Value>
        <Timestamp>2017-01-13T17:45:00</Timestamp>
    </DataRecord>

    <DataRecord id="5">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Flow temperature (1e-1 deg C)</Unit>
        <Value>210</Value>
        <Timestamp>2017-01-13T17:45:00</Timestamp>
    </DataRecord>
```

Figure 39 – mbus-serial-request-data -b 2400 /dev/ttyUSB0 0.

The *PuTTY* program that is used in the project allow access to the *RPi* from another computer through the network. Through *SSH* protocol it's possible to access its shell. See figure 40.

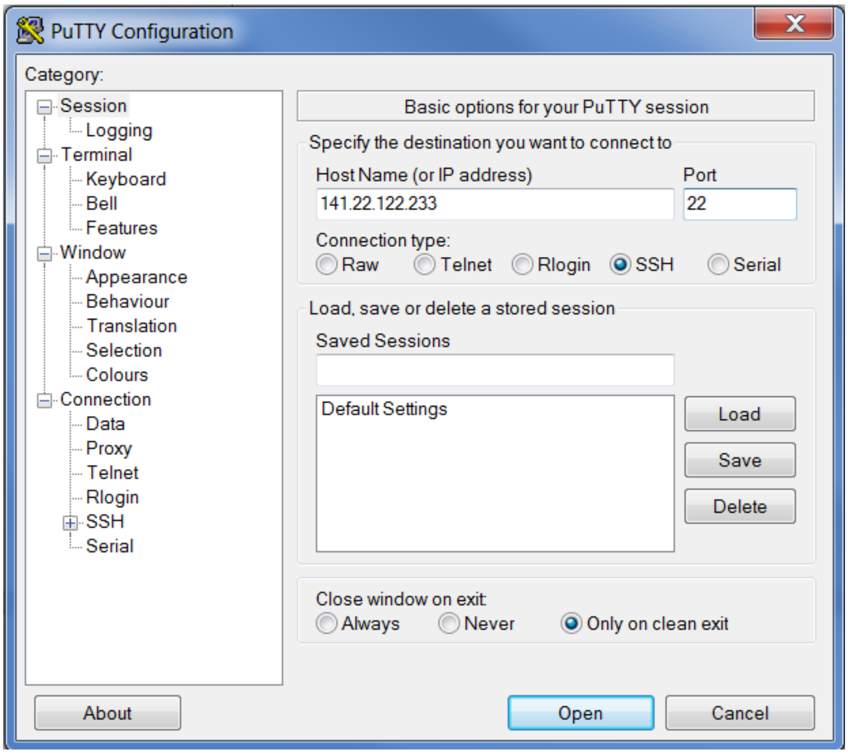
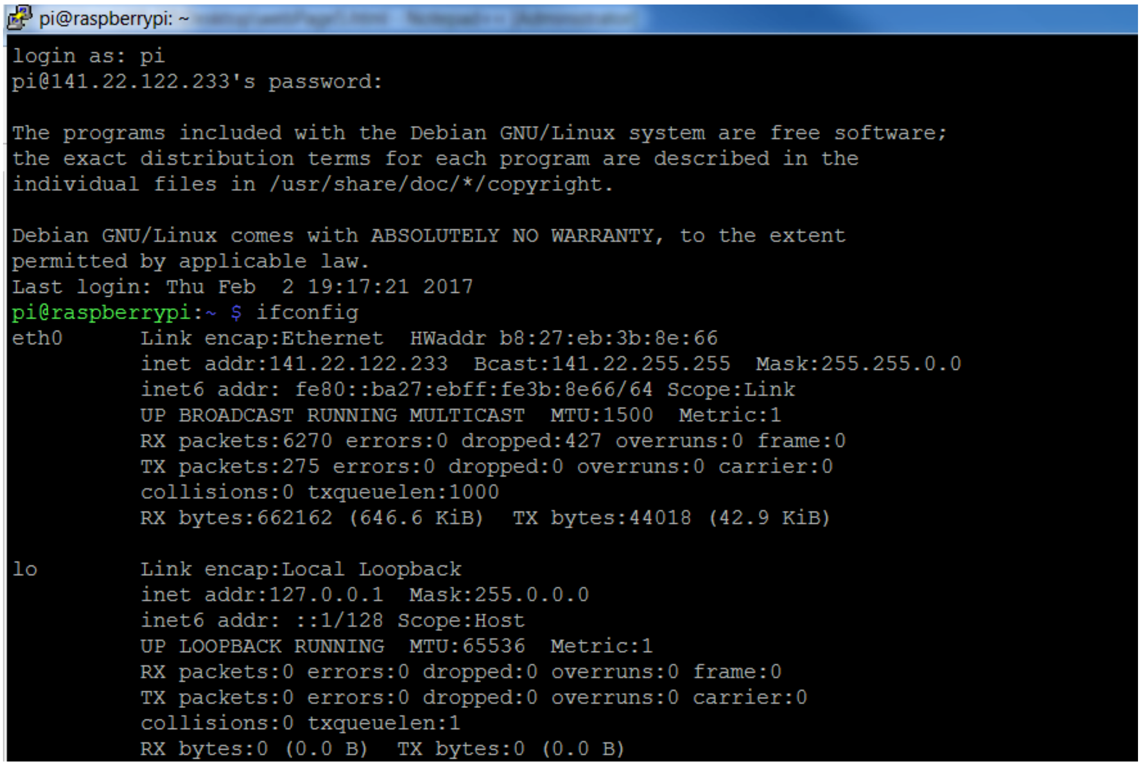


Figure 40 – PuTTY interface.

And after introducing the user and the password of the Raspberry Pi, it's possible to control everything of the device. For example, it can be asked for the information of the internet configuration using the command "*ifconfig*". See figure 41.




```
login as: pi
pi@141.22.122.233's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Feb  2 19:17:21 2017
pi@raspberrypi:~ $ ifconfig
eth0      Link encap:Ethernet  HWaddr b8:27:eb:3b:8e:66
          inet addr:141.22.122.233  Bcast:141.22.255.255  Mask:255.255.0.0
          inet6 addr: fe80::ba27:ebff:fe3b:8e66/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6270 errors:0 dropped:427 overruns:0 frame:0
          TX packets:275 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:662162 (646.6 KiB)  TX bytes:44018 (42.9 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Figure 41 – RPi login through PuTTY.

CONNECTING RPI TO PYCHARM

PyCharm software offers a lot of possibilities. One of them is creating *Python* files for the *RPi* remotely. But to do that some steps are required.

First of all, inside the *PyCharm* software, going to "*Tools -> Deployment -> Configuration*", and adding the name of the new server is needed to control and deploy the project files through SFTP. Then on "*File -> Settings*" adding a remote project interpreter must be done:

The correct interpreter is selected and the next parameters are configured in this project:

- Host: 141.22.122.233
- Port: 22
- User: pi
- Password: raspberry
- Python interpreter: "/usr/bin/python/python3.4"

The screenshot made during the configuration of the *PyCharm* software in this project is shown in the figure 42.

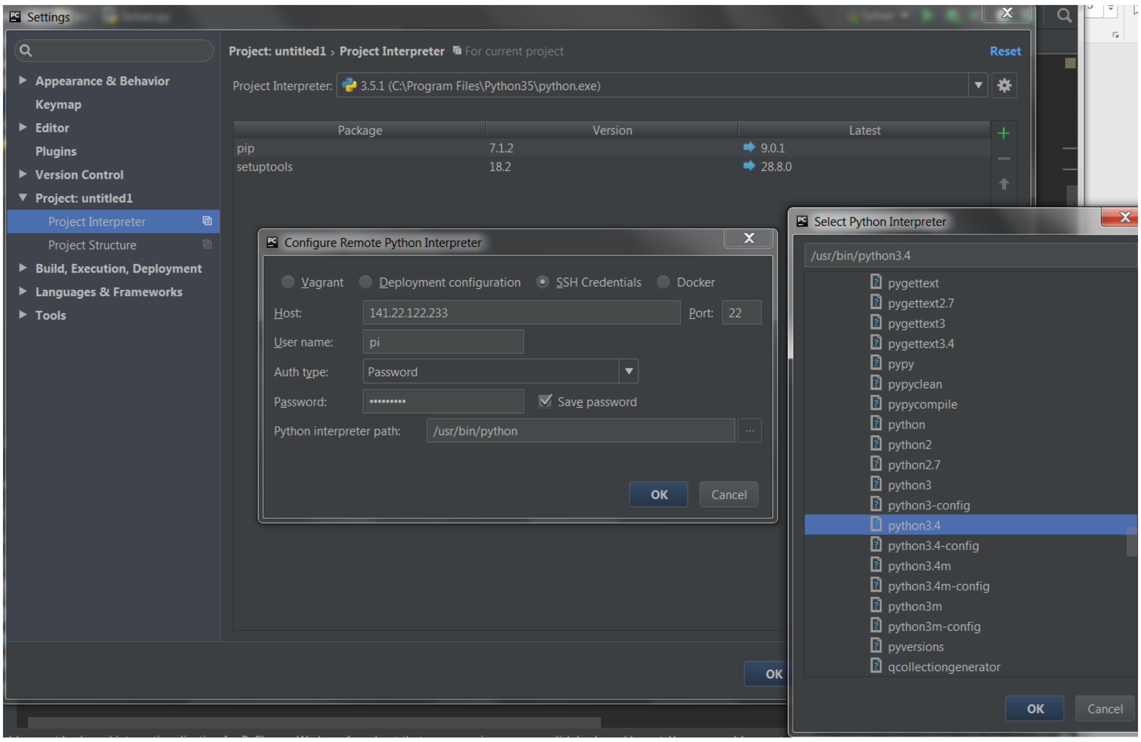


Figure 42 - Connecting RPi to PyCharm.

Then, if it is wanted to test that everything works fine, a little Python code can be created in the PyCharm program. See figure 43.

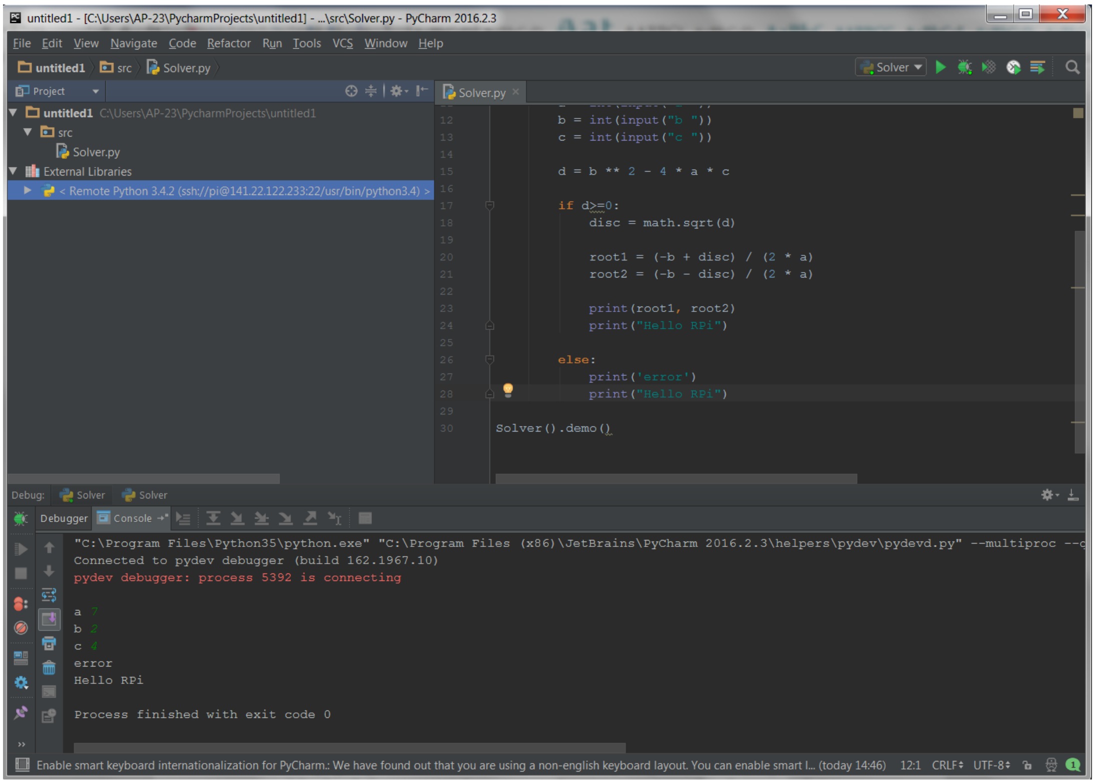


Figure 43 - Connecting RPi to PyCharm (2).

Going to "*Tools -> Deployment -> Configuration*" and entering the next parameters is needed, in order to create a server where the code can be run. See figure 44.

- Host: 141.22.122.233
- Port: 22
- User: pi
- Password: raspberry

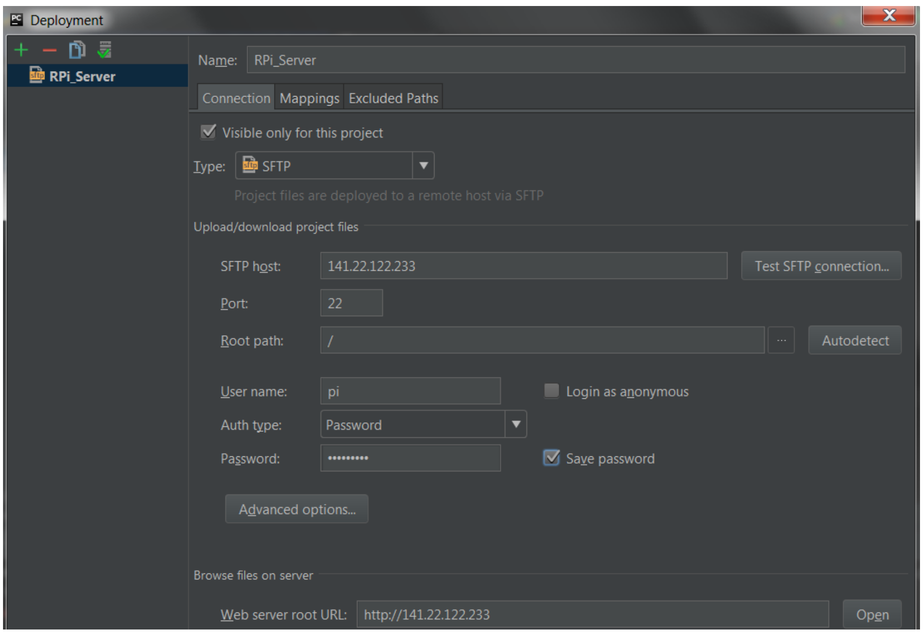


Figure 44 - Connecting RPi to PyCharm (3).

But first, a new folder in the *Raspberry Pi* must be created to be able to do the next step. In this project, it's found in the directory "*/home/pi/*" called "*pythonProjects*". See figure 45.

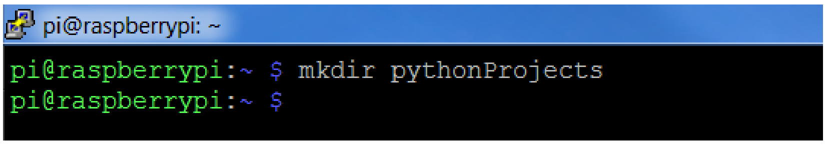


Figure 45 – mkdir pythonProjects.

With that, the *Mappings* configuration can be done, as is shown in the figure 46.

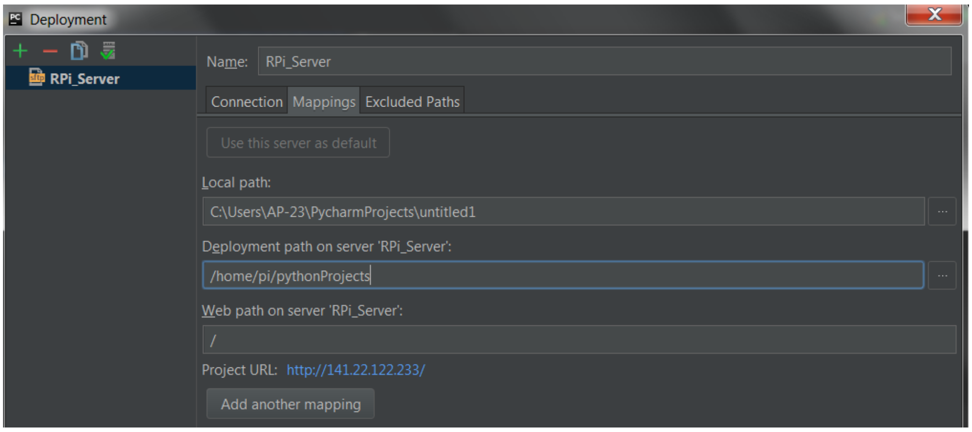


Figure 46 - Connecting RPi to PyCharm (4).

Then, uploading the file to the RPi is the last step, in "*Tools->Deployment->Upload to RPi_Server*". And finally, the *Python* file can be found in the *RPi* just in the folder was created before. See figure 47.

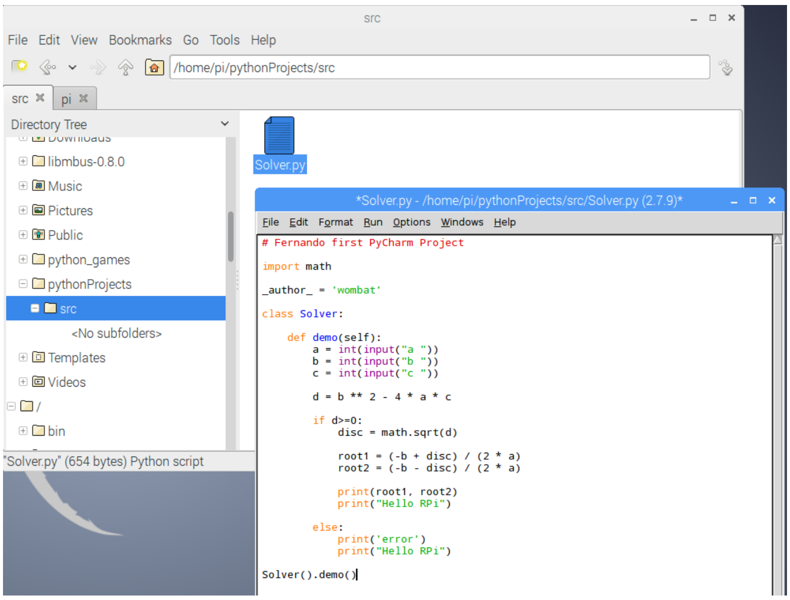


Figure 47 - Connecting RPi to PyCharm (5).

After all these steps, all the work place is configured, working and ready to start creating a program for the *C4DSI*.

Software concepts

In order to make the *Meter Bus* data obtained from the meter accessible from all the computers of the *CC4E* it is needed to create program that is able to not only read but extract, modify and send the data obtained by the master.

As explained before, the master receives the data from the slave, where in this project the master is the *Raspberry Pi* and the slave is the meter. But this data is represented as a text in the shell command of the *RPi* and it shows the data in one exactly moment, not through the time.

Creating an own *Wrapper* is one of the most difficult solutions, but it allows to learn and control a lot. With this solution, up to 5 different programming languages are used and it is required to understand the performance of the *libmbus* library that was created by *rSCADA*, tot all the library but most of his files and folders.

LIBRARIES

Despite having previously commented on the library used in this project, it's necessary to understand a little better what a library is for the development of this project.

A library is a collection of resources used by programs, usually to develop software. These resources can include configuration data, code, classes, values, or type specifications, among others. If it is wanted to write a top-level program, a library can be used to make system calls instead of implementing those system calls repeatedly.

The program calls the library through a language mechanism. What distinguishes the call to a library instead of being to another function in the same program, is the way the code is organized in the system.

The library code is distributed in such a way that it can be used by several programs, while the code that is part of a program is written to be used only within that program.

Most compiled languages have a standard library, although programmers can also create their own custom libraries.

[21]

CTypes

It's also important to explain what *CTypes*, because it is used several times in the programming code. *CTypes* is a module capable of calling in a language to routines or make use of services written in another type of language.

It allows loading dynamic libraries and calling *C* functions. In the case of this project it's used to interact with external *C* code, which is the code in which the *libmbus* library of *rSCADA* is written

This library is loaded using the "*ctypes.CDLL*" function. After loading the library, functions inside the library can already be used as regular *Python* calls.

[22] [23]

Wrapper

This concept has been mentioned some few times in this document. In programming, *Wrapper* is a program or script that makes possible the running of another program. "*Wrapper* libraries consist of a thin layer of code which translates a library's existing interface into a compatible interface. Library *Wrappers* translates the interface of the library into a compatible interface." [24]

[25]

SOFTWARE IMPLEMENTATION

In this project 3 different main are built to implement the server in the *Raspberry Pi*. Three files that interacts between them using up to 5 different programming languages. In the next diagram, shown in the Figure 48, it's possible to see how they are structured and named.

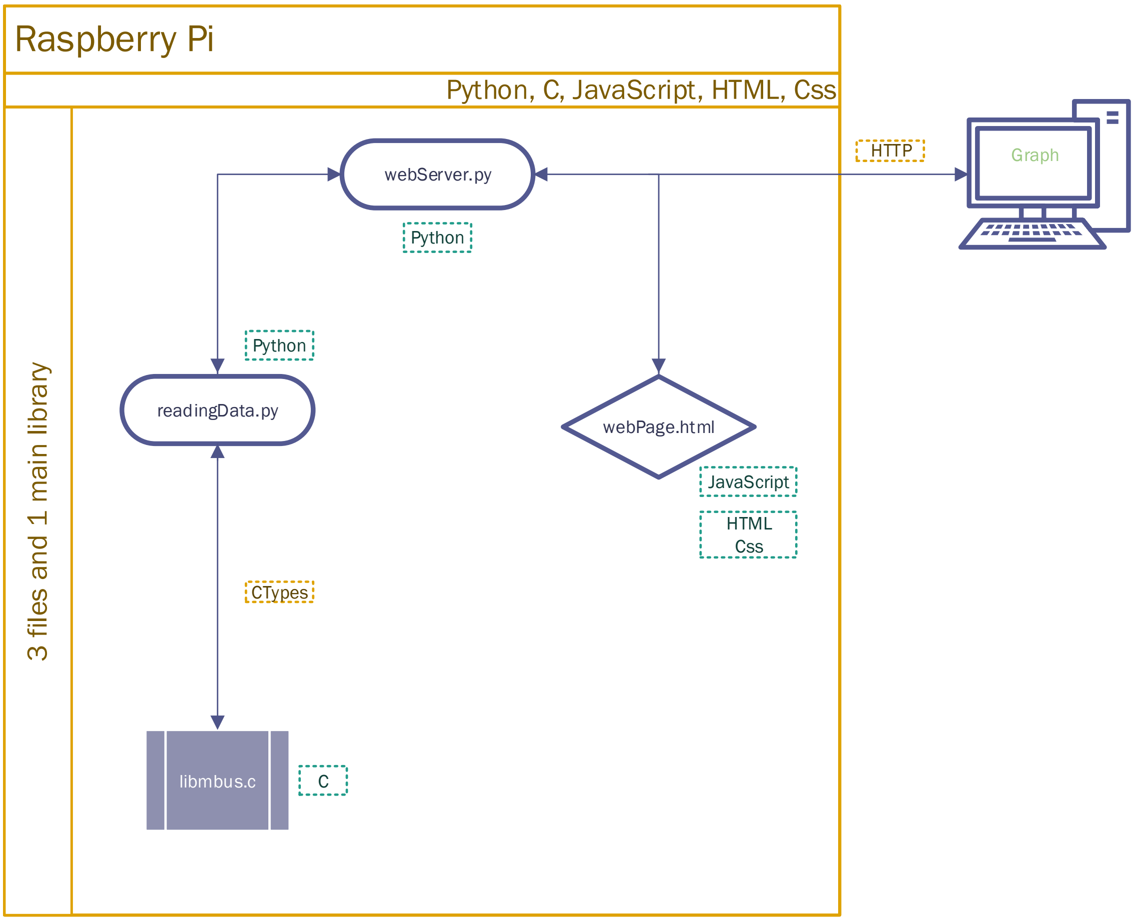


Figure 48 – Main detailed diagram of the code.

All the complete code can be found on the attached data of the thesis.

readingData.py

Is important to start explaining this file first because it's the file used to extract the data from the meter using the *libmbus* library. This file contains 363 lines, 5 imports, 15 class definitions and 5 functions. All the imports and class definitions are needed to use the functions of the *libmbus* library.

This file is the *Wrapper* created for this project, that interacts with the most important and needed functions of the *libmbus* library, using *Python* code and interacting with the *C* code used in the library using *CTypes*. In the figure 49 it's possible to see an overview of the 5 functions.

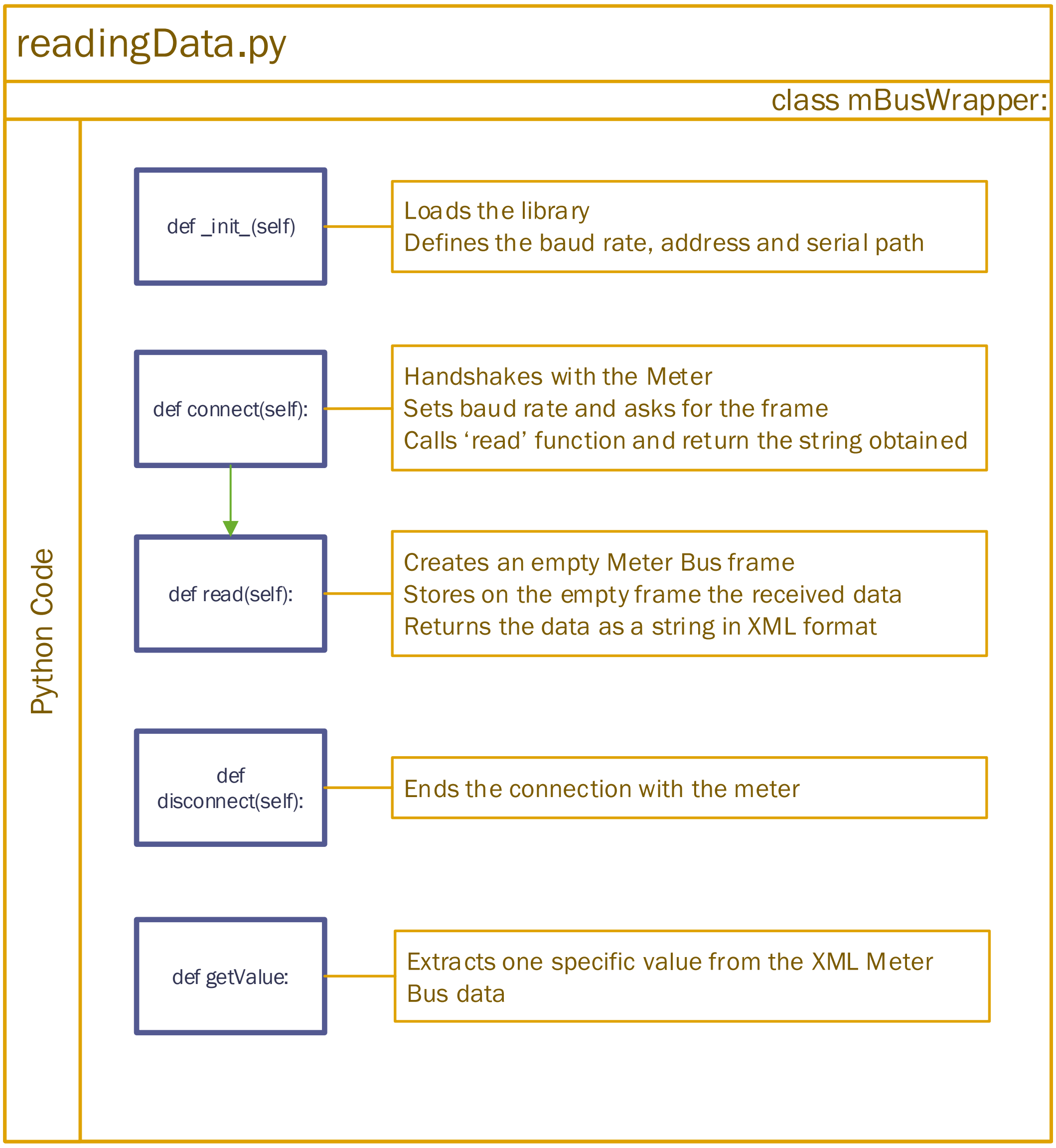


Figure 49 – Main detailed diagram of the readingData.py file.

The functions used in the *Wrapper*, as well as in the other two files, are needed to make possible the goal 2 exposed at the start of this document. The explanation of the code is made trying to comment only the most important code lines.

- Def _init_(self):

Inside the *mBusWrapper* class, this one is the first function that is executed when the class is loaded. It defines the variables that makes possible the communication with the meter of this project and loads the C library *libmbus* using *CTypes*. The serial path needs to be encoded to be used correctly by the C library.

```
class mBusWrapper:
    def __init__(self):
        self.myLib = ctypes.cdll.LoadLibrary("/usr/local/lib/libmbus.so")
        self.baudRate = 2400
        self.address = 0
        self.serialPath = "/dev/ttyUSB0"
        self.utfSerialPath = self.serialPath.encode('utf-8')
```

- Def connect(self):

Also inside the *mBusWrapper* class, the handshake between the *Raspberry Pi* and the meter is done by this function. This function is executed only when is called by the *webServer.py* file.

First, the type of data that the C function "*mbus_connect_serial*" needs and returns, must be set. This is very important to interact correctly with the meter. With the next lines, it is said that the given data to the function is a *pointer* of type char and the result is a *pointer* of type '*mbus_handle*'.

```
    def connect(self):
        self.myLib.mbus_connect_serial.argtype = ctypes.POINTER(ctypes.c_char)
        self.myLib.mbus_connect_serial.restype = ctypes.POINTER(mbus_handle)
```

It's also important to know what a *pointer* is, because there are a lot of *pointers* used in the *Wrapper*. "In computer science, a *pointer* is a programming language object, whose value refers to, or "points to", another value stored elsewhere in the computer memory using its memory address." [26]

The type of data '*mbus_handle*' and all the other types of data used by the *libmbus* library, are defined at the start of the file as a class. "A class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods)." [27]

The result pointer of type '*mbus_handle*' that the handshake between the *RPi* and the meter has given by the *C* function "*mbus_connect_serial*" is stored in the variable '*handleValue*'.

```
        self.handleValue = self.myLib.mbus_connect_serial(self.utfSerialPath)
```

From this data, it is needed to extract only the *Serial* value, because the library is made for 2 ways of connection, *Serial* and *TCP*. In the case of this project only the *Serial* value is needed. This is done using the *pointer* of type '*mbus_handle*' from before.

```
        handleSerialValue = self.handleValue.contents.m_serial_handle
```

So, it's kept only the handshake serial value given by the previous *C* function. In this case, the '*self*' before the value is not needed because this variable won't be used outside the connect function.

Once the hand shake is made, baud rate must be set following the same rules explained before. All inside the same function, using the *C* function "*mbus_serial_set_baudrate*" defined in the *C* library and the previous handle serial value extracted.

```
        self.myLib.mbus_serial_set_baudrate.argtypes =
[ctypes.POINTER(mbus_serial_handle),ctypes.c_int]
        self.myLib.mbus_serial_set_baudrate.restype = ctypes.c_int

        intSerialSetBaudRate =
self.myLib.mbus_serial_set_baudrate(handleSerialValue, self.baudRate)
```

And now the *Raspberry Pi* can ask for the *MBus* frame, the main goal of this file.

```
        self.myLib.mbus_send_request_frame.argtypes =
[ctypes.POINTER(mbus_handle), ctypes.c_int]
        self.myLib.mbus_send_request_frame.restype = ctypes.c_int

        intSendRequestFrame =
self.myLib.mbus_send_request_frame(self.handleValue, self.address)
```

The *libmbus C* library made by *rSCADA* contains a lot of functions for the protocol *Meter Bus*, but in this project only a few of them are needed. These ones are the needed for the purposes of this project. This functions has been chosen after studying all the files of the library and understanding how the library code works.

With the study of the *Meter Bus* protocol and the *C* library has been possible to make this *Wrapper* in *Python*, also studying how *Python* and *CTypes* works.

With the '*int*' variable returned by the previous function it is only known if the connection is successfully, but the meter knows that the data is wanted, so the conversation has been made.

Now the next function is called inside the class *mBusWrapper*, that obtains the information from the meter.

```
        vals = self.read()
```

And this function returns a string with the *Meter Bus* data in *XML* format.

```
        return vals
```

- Def read(self):

This function is called by the *connect* function explained before, and its main goal is to extract the data from the meter. To do that first 2 empty variables of type '*mbus_frame*' and '*mbus_frame_data*' are created.

```
    def read(self):
        mBusFrameExample = mbus_frame()
        mBusFrameDataExample = mbus_frame_data()
```

First, the "*mbus_serial_set_baudrate" C* function needs to be called. This function returns an '*int*' type that if its 0 means that everything worked correctly, after applying some functions of the *C* library.

This *C* function needs the handle value from the previous hand shake connection, and a pointer to the empty *Meter Bus* frame created before.

```
        self.myLib.mbus_recv_frame.argtype = [ctypes.POINTER(mbus_handle),
ctypes.POINTER(mbus_frame)]
        self.myLib.mbus_recv_frame.restype = ctypes.c_int

        emptyFramePointer = ctypes.addressof(mBusFrameExample)

        intReceivedFrame = self.myLib.mbus_recv_frame(self.handleValue,
emptyFramePointer)
```

This next *C* function needs the same address (*pointer*) of the empty *Meter Bus* frame and the *pointer* of the empty *MBus* frame example, so it can store the *Meter Bus* frame inside. If everything works fine it will return a 0, and the *MBus* data will be stored in the address given.

```
        self.myLib.mbus_frame_data_parse.argtype =
[ctypes.POINTER(mbus_frame), ctypes.POINTER(mbus_frame_data)]
        self.myLib.mbus_frame_data_parse.restype = ctypes.c_int

        emptyFrameDataPointer = ctypes.addressof(mBusFrameDataExample)

        intMBusDataParse = self.myLib.mbus_frame_data_parse(emptyFramePointer,
emptyFrameDataPointer)
```

Now this data is required in *XML* format so it can read, understood and extracted the exact data that it's wanted from all the *MBus* data. To do that the *C* function from the *libmbus* library it's called using *CTypes*, the same method used before, giving now to the function only the address (*pointer*) of the frame *MBus* data. So, it can know where the data is stored.

```
        self.myLib.mbus_frame_data_xml.argtype =
ctypes.POINTER(mbus_frame_data)
        self.myLib.mbus_frame_data_xml.restype = ctypes.c_char_p
```

```
        charPointerMBusDataXML =
self.myLib.mbus_frame_data_xml(emptyFrameDataPointer)
```

It is important to say that the result type of the function must be '*c_char_p'*, because this is used for pointing to a null terminated string, if not the function doesn't work. The result of the previous *C* function will be the address of all the *XML Meter Bus* data.

Now the data must be decoded in order to be used correctly as a string by the *Wrapper*.

```
        charMBusDataXML = charPointerMBusDataXML.decode('utf-8')

        return charMBusDataXML
```

- Def getValue(self, xml, id):

Function used by the *webServer.py* file, that extract the exact data that it's wanted for the project, from the whole *XML* string. This function uses *objectify*, imported at the start of the file, which allows you to extract data from *XML*.

```
    def getValue (self, xml, id):
        mBusData = objectify.fromstring(xml)
        value = mBusData.DataRecord[id].Value
        return value
```

The number or the word extracted from the whole *XML* data is stored in the *value* variable.

- Def disconnect(self):

Used for ending the communication between the *Raspberry Pi* and the meter, using the previous handle value, so it knows which connection ends:

```
    def disconnect(self):
        self.myLib.mbus_disconnect.argtype = ctypes.POINTER(mbus_handle)
        self.myLib.mbus_disconnect.restype = ctypes.c_int

    intMBusDisconnect = self.myLib.mbus_disconnect(self.handleValue)
```

webServer.py

This one is the main file. This is the entry point and the one who works with all the files. It's starting an endless server that is constantly expecting for requests. This file contains 138 lines, 3 imports, 1 class and 2 functions. In the figure 50 it's possible to see an overview of the file.

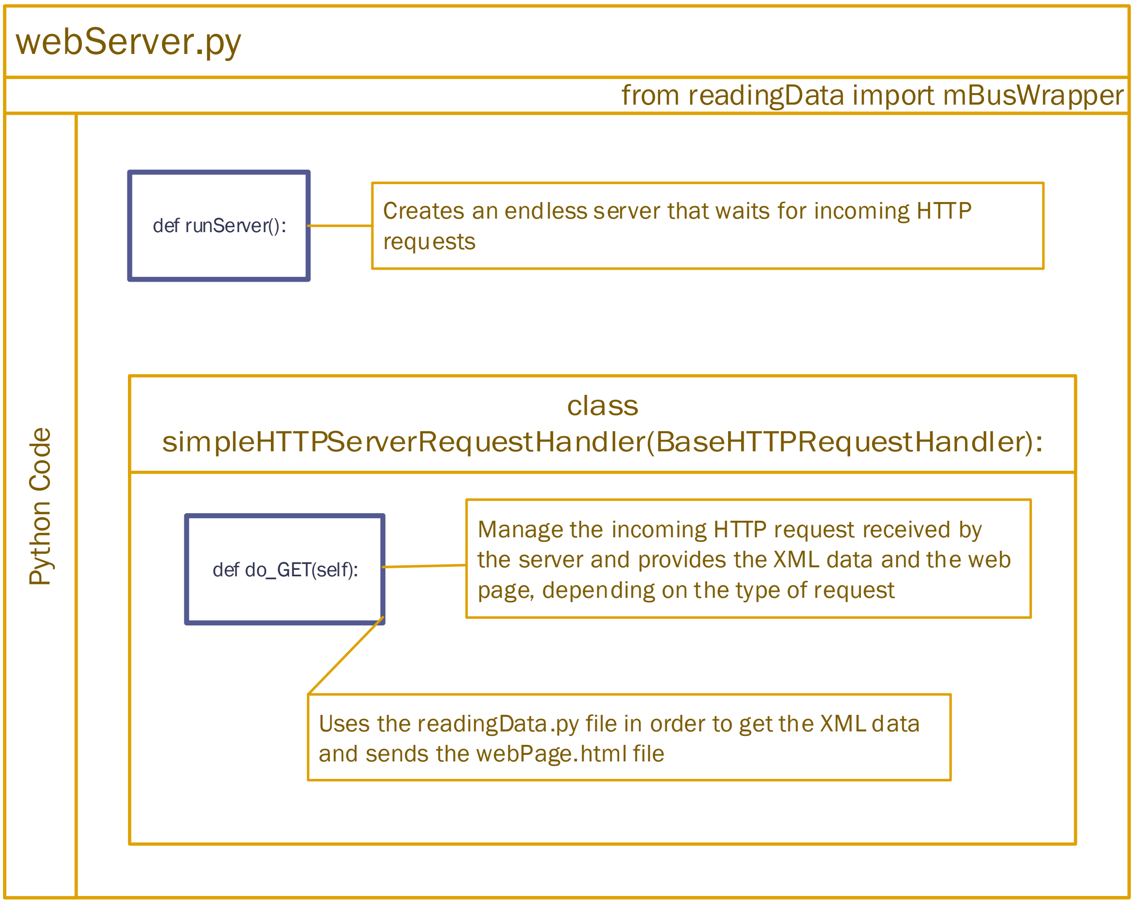


Figure 50 – Main detailed diagram of the webServer.py file.

As it can be seen in the diagram of the previous figure 50, this file is importing the previous explained *class mBusWrapper* from the file *readingData.py*, and uses it in the code.

```
from readingData import mBusWrapper

mbus = mBusWrapper()
```

It has only one execution line of code, which executes the main function of this file.

```
runServer()
```

- Def runServer():

The main function that runs the *HTTP* server forever, until it's stopped manually.

```
def runServer():
    server_address = ('', PORT)
    server = HTTPServer(server_address, simpleHTTPServerRequestHandler)
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print('\n^C received, shutting down the web server')
        server.socket.close()
```

To create the server, the library '*http.server*' is used. It is given to 2 parameters, the server address and a class created before in the code, as it can be seen in the previous code.

```
from http.server import BaseHTTPRequestHandler, HTTPServer
```

This server is created for listening uninterruptedly for incoming *HTTP* requests from other *PC's* of the *CC4E* building, so with this method the *PC's* can ask for the data obtained by the *Raspberry Pi* using the *HTTP* protocol. The *port* is defined before which is not the default port for *HTTP*. Usually is 80 but on the *Raspberry Pi* ports under 1024 needs a root access, so the 8080 port is used.

The class "*simpleHTTPServerRequestHandler*" is inheriting the class defined by the library and overwriting the '*do_GET*' method in order to use it correctly for the purposes of this project, as it can be seen in the code below.

- Def do_GET(self):

This function is allocated in the class that it is mentioned before. Its main function is to handle the incoming *HTTP* requests from the *PC's* that wants to access to the data given by the *Raspberry Pi*.

```
class simpleHTTPServerRequestHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path == "/":
            self.send_response(200)
            self.send_header('Content-type', 'text/html')
            self.end_headers()

            f = open(curdir + sep + "webPage.html", 'rb')
            self.wfile.write(f.read())
            f.close()

        elif self.path == "/all":
            self.send_response(200)
            self.send_header('Content-type', 'text/xml')
```

```
        self.end_headers()

        values = mbus.connect()
        values = str(values)
        mbus.disconnect()

        values = values.encode('utf-8')
        self.wfile.write(values)

    elif self.path == "/volume-flow":
        self.send_response(200)
        self.send_header('Content-type', 'text')
        self.end_headers()

        VOLFLOW = 4
        xml = mbus.connect()
        value = str(float(mbus.getValue(xml, VOLFLOW))/10.0)
        mbus.disconnect()

        value = value.encode('utf-8')
        self.wfile.write(value)
```

Depending of the *HTTP* request received, this function is doing different things. The first one "/" if someone writes the *IP* address of the *Raspberry Pi* and the 8080-port, sends an *HTML* file to the navigator that requested that, and the navigator applies all the functions that the *HTML* file contains. This file is explained later in this document.

With the second case "*/all*" if someone writes the *IP* address of the *Raspberry Pi* and the 8080-port followed by the word '*all*', the server sends to that *PC* all the *XML* file extracted from the meter but without the *HTML* file. Only the *XML* text file.

In the third case an exact value of the XML is extracted and sent as a text, also without sending the *HTML* file, only the *XML* text value. In the previous code the *ID4* of the *XML* file is shown, but there are also the *IDs* 5 and 6 used in this project.

webPage.html

This file contains 3 different languages: *HTML*, *CSS* and *JavaScript*. The first one is defining the structure and the contents inside the web page while the *CSS* defines the visual style of the page. The *JavaScript* contains the programming code of the web.

On the *HTML*, there are created 3 empty labels, which ones will contain the 3 graphs that are created below.

In the figure 51 it's possible to see an overview of the file.

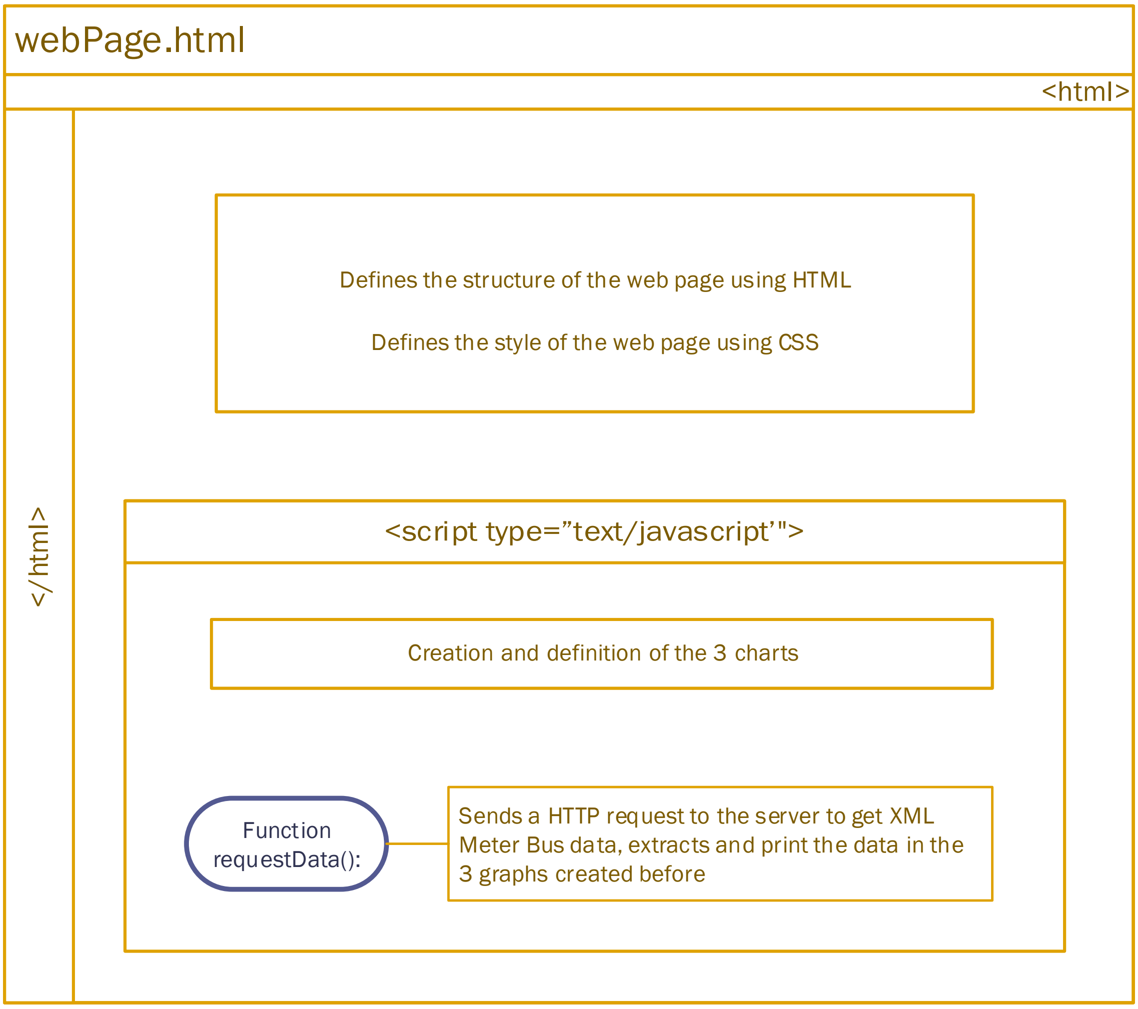


Figure 51 – Main detailed diagram of the webServer.py file.

With the *JavaScript* code, 3 objects are defined that contains 3 charts that are created thanks to the *HighChart* import made before in the *HTML* code.

These ones contain things like the title, axis, or series. Also, contains a call to a function that will control later that all the 3 graphs are loaded before adding values to the graphs. This function is only called when the constructor (the library) finishes building the object.

Before the definition of the charts, a *jQuery* function is used in order to make sure that all the *HTML* code is ready in the navigator. This *jQuery* function is represented by the $ symbol and is loaded before in the *HTML* code, as well as the *HighChart* import.

- function requestData() {

This is the main function of this file, is written in *JavaScript* and is the one that prints the graphs with the data taken from the server.

This function prints extracts and print the data into the 3 graphs defined before in the code.

```
function requestData() {
    $.ajax({
        url: '/all',
        success: function(xml) {

            var volumeFlow = parseFloat(xml.querySelectorAll("DataRecord
Value")[4].firstChild.textContent);
         var flowTemperature = parseFloat(xml.querySelectorAll("DataRecord
Value")[5].firstChild.textContent)/10.0;
            var returnTemperature =
parseFloat(xml.querySelectorAll("DataRecord
Value")[6].firstChild.textContent)/10.0;

            var now = new Date().getTime();
            volFlowChart.series[0].addPoint([now, volumeFlow], true,
volFlowChart.series[0].data.length > 20);
            flowTempChart.series[0].addPoint([now, flowTemperature], true,
flowTempChart.series[0].data.length > 20);
            retTempChart.series[0].addPoint([now, returnTemperature], true,
retTempChart.series[0].data.length > 20);

            // call it again after one second
            setTimeout(requestData, 1000);
        }
    });
}
```

Request data function uses the *jQuery* library and *Ajax*, the technique that allows to make possible that webpages actualize themselves without having to download all the page again, automatically.

Inside this function, an "*/all*" *HTTP* request is made to the server. And when the data has been received correctly, the next function "*function(xml)*" is called.

Them, the needed values are extracted from the *XML* data and sets into the *Y* axis of each corresponding graph. Also, a time is set for the *X* axis, the same for the 3 graphs, and the printing of the values inside the graphs is made.

This function is also important because it calls itself every one second, actualizing the graphs.

The last line makes all the process explained start again. It makes the whole process start since the start. All the whole process.

Endless until it the user closes the navigator.

[28] [29] [30]

SOFTWARE VALIDATION

Some screenshots were taken during the several tests of the developing of this thesis. The next screenshots shown in this part of the project, shows the final result when all the code is working correctly.

These screenshots are taken on windows, in a computer of the *CC4E* building and using the *PyCharm* software. The software is run while in the heat meter station the *Raspberry Pi* the whole system explained in the goal 2 is connected and working. See figure 52.

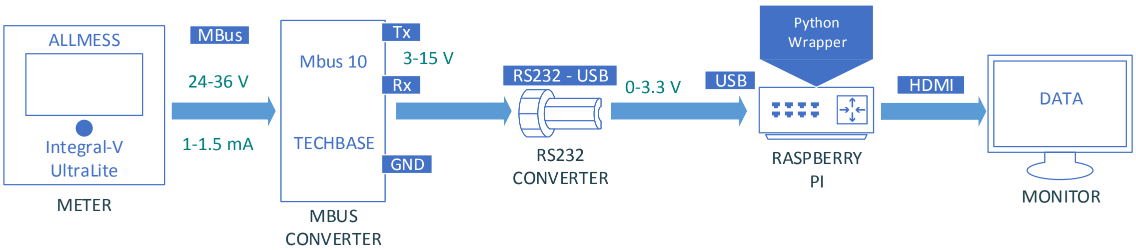


Figure 52 – Main detailed diagram of Goal 2 (2).

Also, some videos were taken, in order to see more detailed and in first person how the whole programming server works. This videos are attached in the *CD* of this project.

In the next figure, it can be seen the screenshot validations of the working *Wrapper*, defined in the *readingData.py* file. The screenshot is taken when only the *readingData.py* has been written. See figure 53.

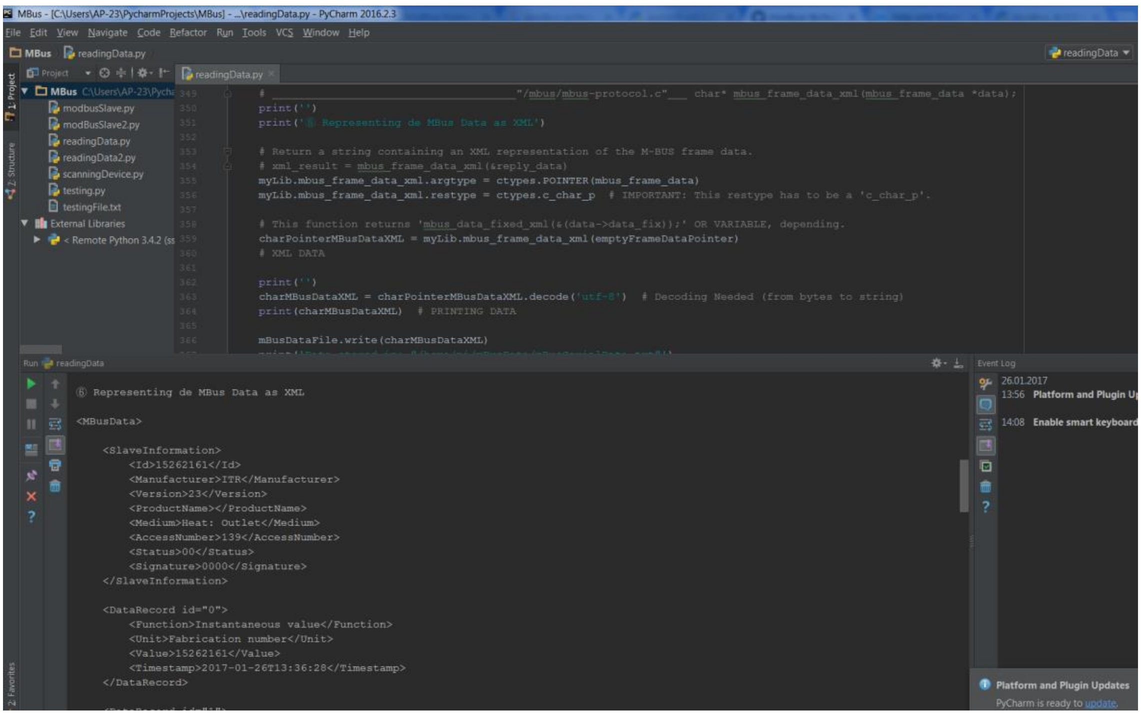


Figure 53 – readingData.py PyCharm running

In the next screenshot, it can be seen how the final web page looks like after all the 3 files are written. This is how the web page looks like running. See figure 54.

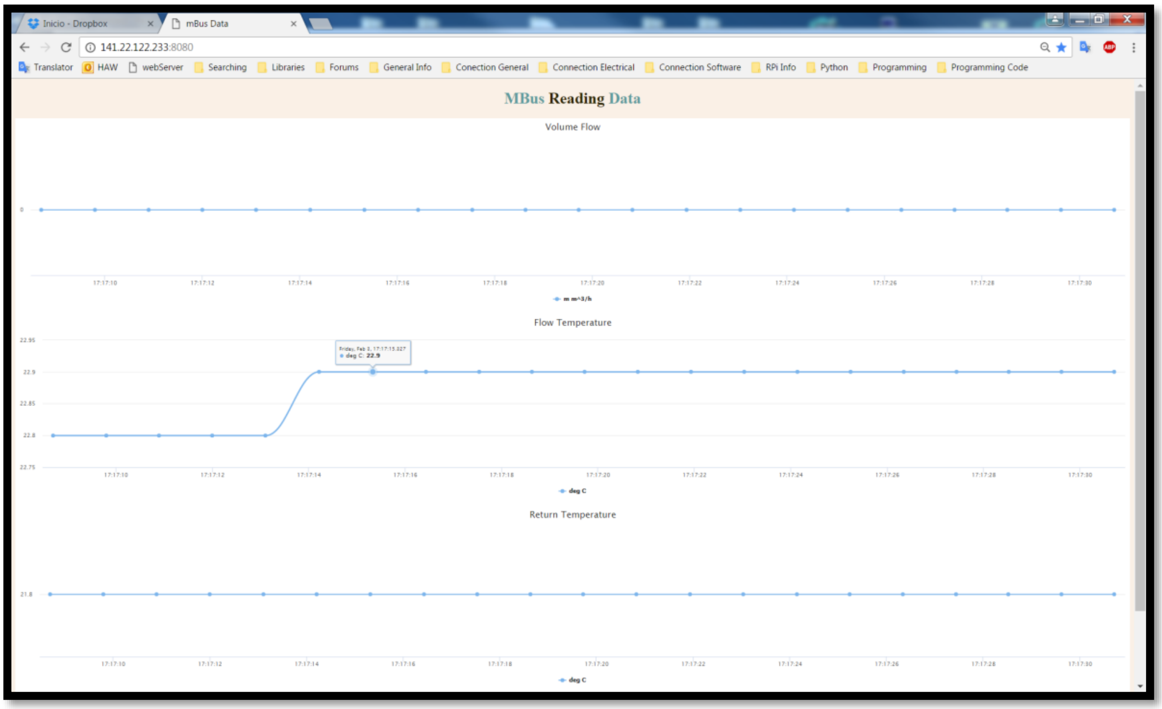


Figure 54 – Web Page Running.

Hardware concepts

The last goal of this project is to replace successfully the previously explained *RS232-USB* converter made by *Digitus*, in order to make the system as cheaper as possible.

To replace this converter the *GPIO* pins from the *Raspberry Pi* are used, so it's not need to use the *USB* port anymore. The exactly pins used are the *UART* pins 14 and 15 for transmitting and receiving data, respectively.

"A universal asynchronous receiver/transmitter, *UART*, is a computer hardware device for asynchronous serial communication in which the data format and transmission speeds are configurable. The electric signaling levels and methods (such as differential signaling, etc.) are handled by a driver circuit external to the *UART*." [31]

In the case of this project the external circuit that the definition of *UART* is talking about is the circuit built for this project and it is explained below. In the next figure 55 it can be seen a diagram of this circuit.

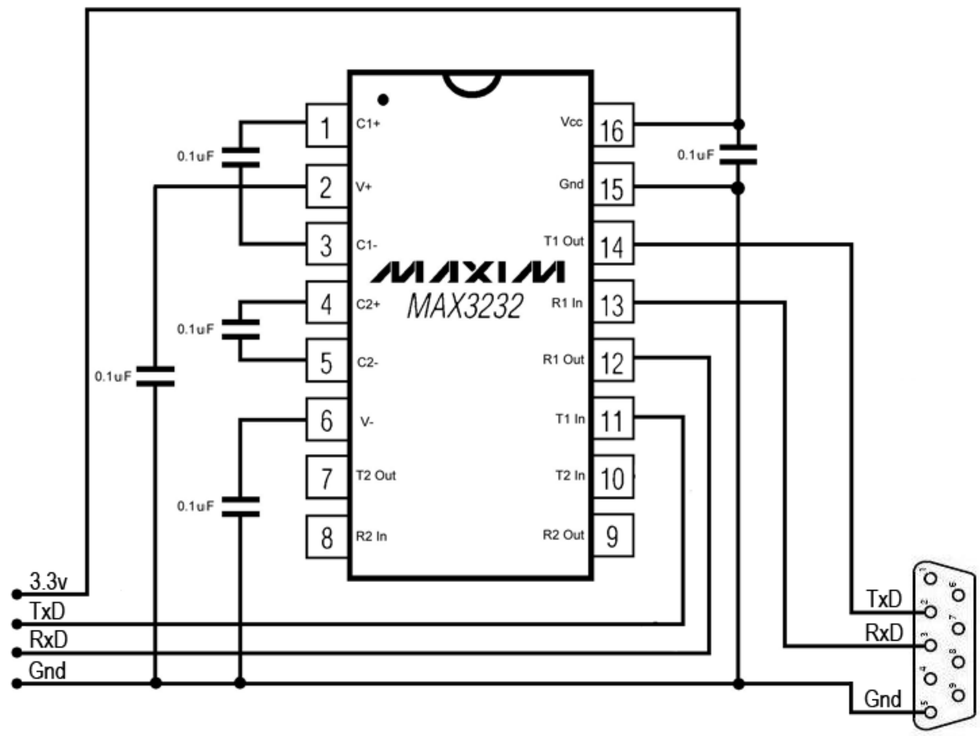


Figure 55 – Converter 1 diagram circuit. [32]

Also, as an extra goal, in this project it's built another circuit trying to replace the most expensive part of the system, the *Meter Bus* converter made by *TechBase*. The circuit design is extracted from the official documents, in this case the *UNE-EN 1434-3* Spanish rule, edited and printed by *AENOR*, the Spanish association of normalization and certification. In the next figure 56 it can be seen a diagram of the circuit, extracted from the official *PDF* named before.

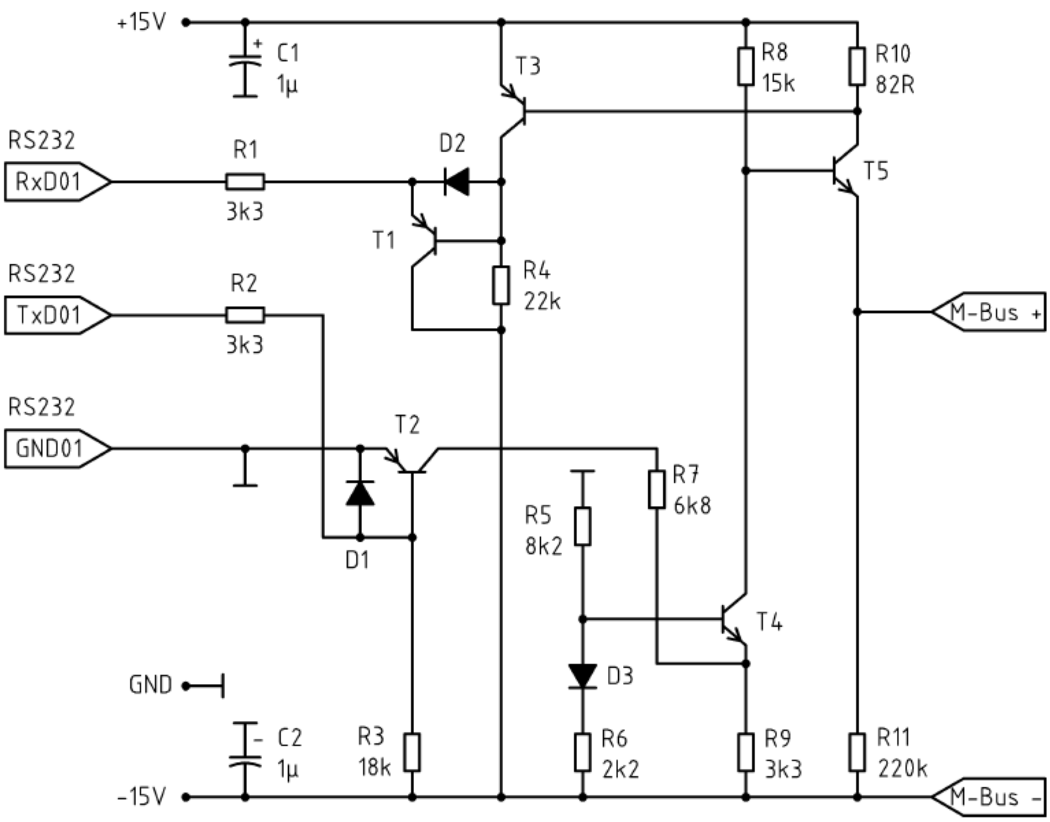


Figure 56 – Converter 2 diagram circuit.

The implementation and validation of the hardware concepts are made at the same time. It must be made before connecting everything in order to not damage the devices or the components of the circuit.

The software validation with the hardware implemented is also done, but after all the implementation and hardware validations.

HARDWARE IMPLEMENTATION AND VALIDATION

To do the goal 3, a level converter is built in a *protoboard* using some components, as it is shown in the previous figure 55. The info of how to build a level converter from serial to *TTL/CMOS* can be extracted from the references [33] and [34], or from the official datasheet [35]. This is the circuit that the *UART* definition is mentioning, as said before.

The circuit is composed by 5 capacitors of 0.1*µF* each one and one *MAX3232 CPE* transceiver. In the figure 57 it's possible to see the name of every pin of the transceiver and its internal circuit, figure extracted from the official datasheet that can be also found in the reference [35].

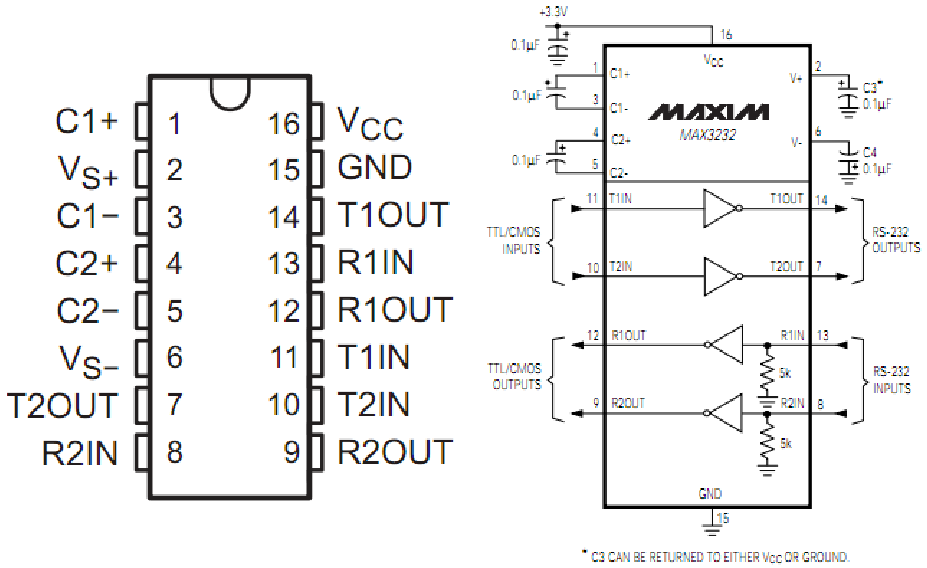


Figure 57 – MAX3232 diagram. [35]

With all this information, the circuit is built in a protoboard as it can be seen in the figure 58 and after double checking everything it is prepared to be tested by an oscilloscope to check its behavior. From now on this circuit will be called circuit 1.

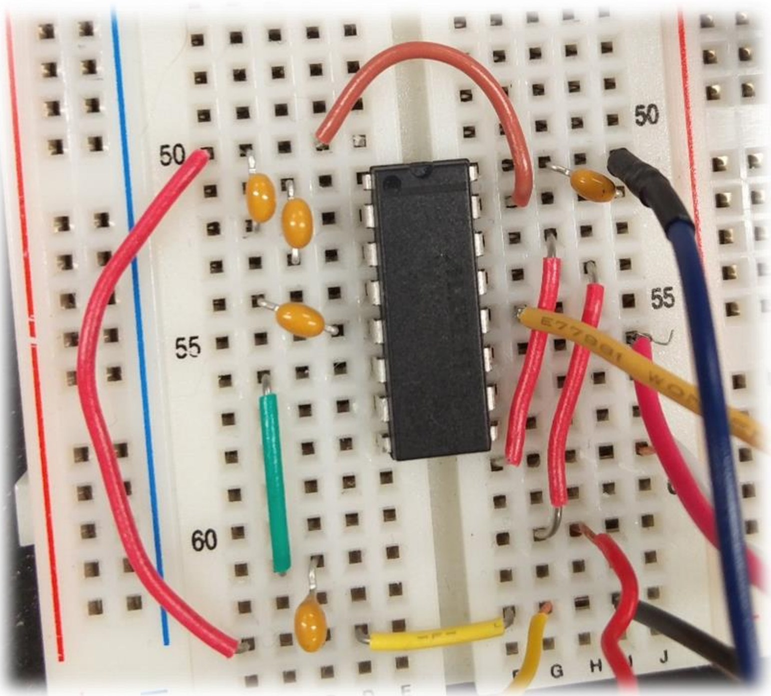


Figure 58 – Circuit 1 built.

First Checking

The power supply it's providing +3.3 volts to the pins 16 and 11 of the *MAX3232*, *Vcc* and *T1IN* respectively. The transceiver needs to be fed with that voltage level. The ground is connected to the same ground of the circuit, pin 15. See figure 59.

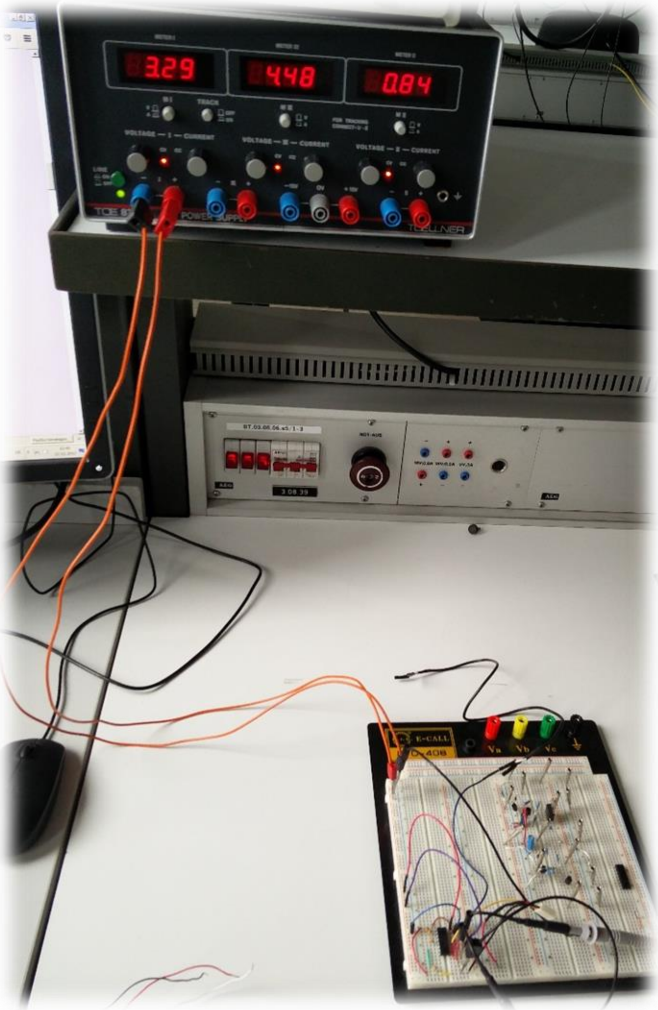


Figure 59 – Circuit 1 ready to be checked.

The channel 1 of the oscilloscope, in the case of this project with a strong blue color, is connected to the same ground of the power supply and the circuit, and at the same time to the pin 11, *T1IN*. The channel 2, soft blue color, is connected to the pin 14 of the *MAX3232*, which is the *T1OUT*.

With this connection, it is tried to see how the circuit is acting when the data is coming from the *Raspberry Pi* to the *RS232* serial cable that goes to the *Meter Bus* converter. The result provide by the oscilloscope is shown in the figure 60.

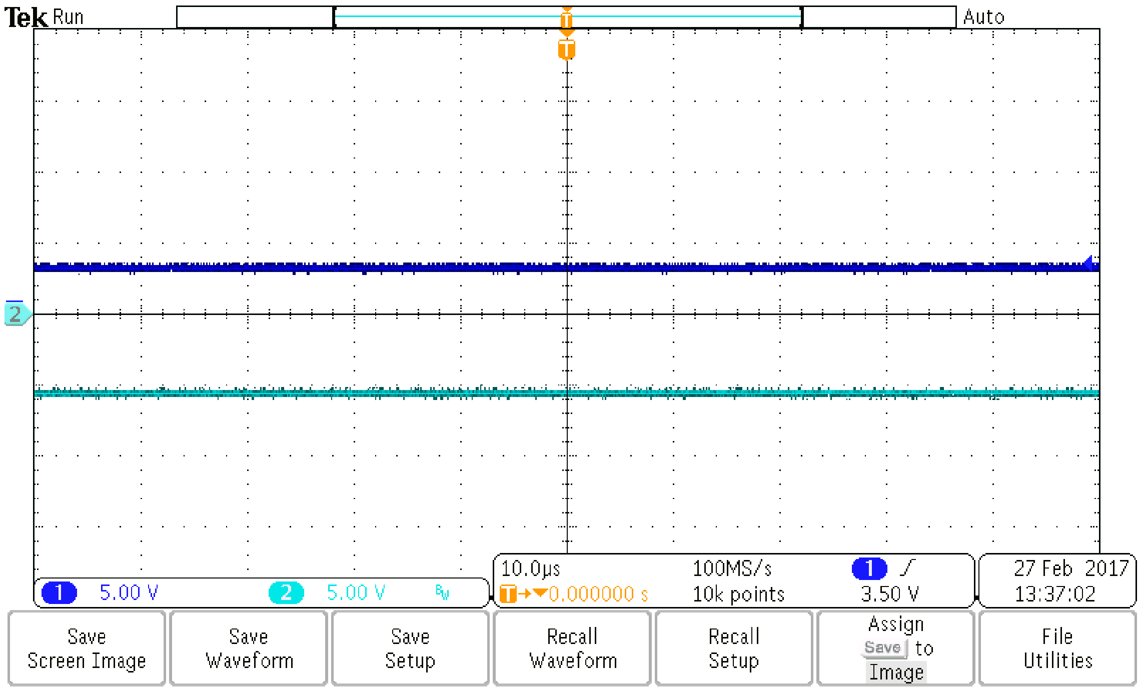


Figure 60 – Oscilloscope screenshot 1.

In this previous figure 60 it can be seen how the input is 3.3v, the strong blue line seen in the figure. The same that our power supply is giving us, a constant value along the time. That's exactly what is going inside *T1IN*, as explained before, and this represents the data sent from the *Raspberry Pi*.

In the line marked with a soft blue color, it can be seen the result that represents the signal after going through the transceiver. The level voltage has been increased a little bit more than 5 volts. This voltage is negative because in the internal circuit of the transceiver it goes through an inverter, as it can be seen in the previous *MAX3232* diagram.

"In digital logic, an inverter or NOT gate is a logic gate which implements logical negation." [36] When the input bit is a '0', the inverter returns a '1'; and when the input bit is a '1' it returns a '0'.

This circuit is in the middle of a digital communication between the *Raspberry Pi* and the meter. The bit '0' and the bit '1' used sent by the *RPi* are interpreted different by the *RS232* protocol. This transceiver *MAX3232* is used because in *RS232* it's represented as a bit '0' the signals with voltage levels between +3V and +15V; and as a bit '1' for the voltage levels between -3V and -15V. See figure 61.

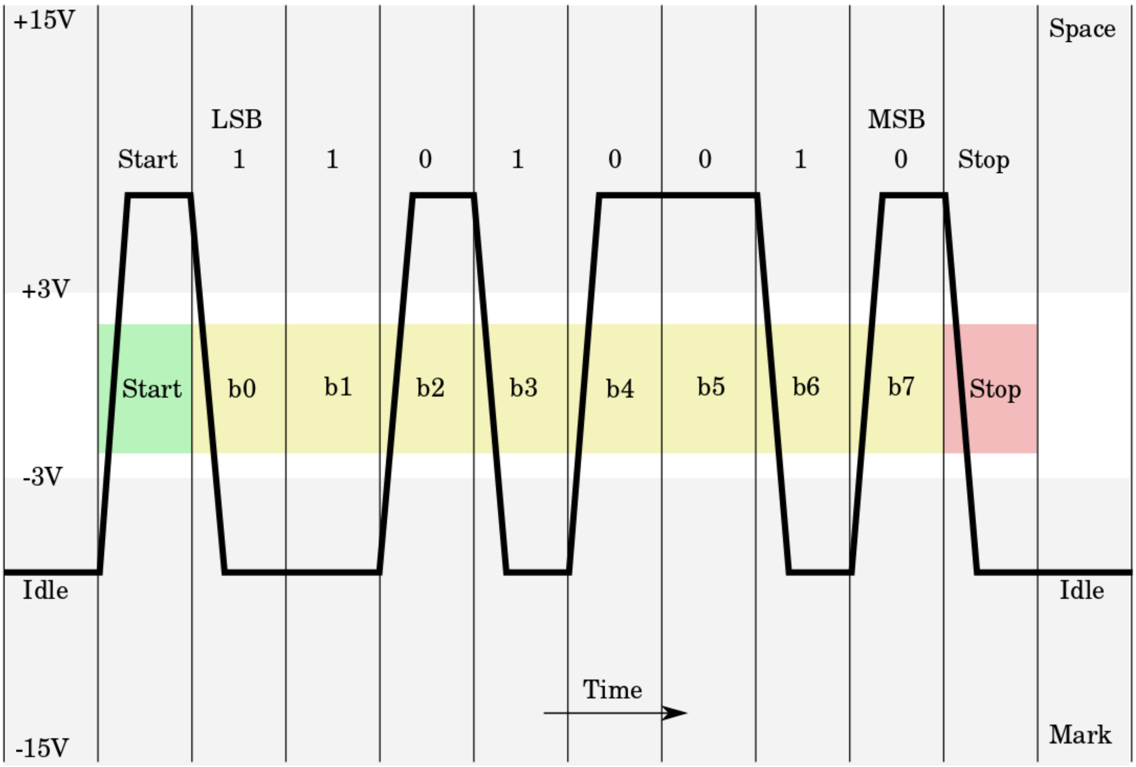


Figure 61 – Voltage levels RS232. [8]

It's also necessary explain the *TTL/CMOS* voltage levels. Every input voltage signal between 2V and 5V (or 3.3V in the case of the *RPi*) is considered as a bit '1'; while every input voltage signal between 0V and 0.8V is considered as a bit '0'. See figure 62.

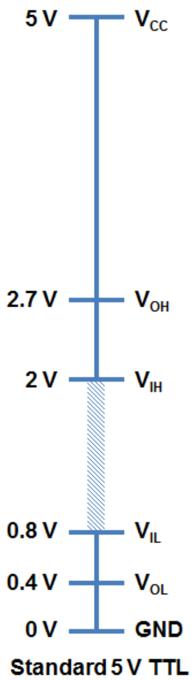


Figure 62 – Voltage levels TTL/CMOS. [37]

Also, the *Raspberry Pi* always sends an output signal level of at least 2.7V, not less, for the bit '1'; and it sends outputs signals not higher than 0.4V for the bit '0'.

As it is said previously, the output voltage seen in the oscilloscope is approximately -5v. That means that to an incoming signal of +3.3V the transceiver MAX3232 is sending a signal with a voltage level of -5V. That also means that in the *RS232* the reading of this signal it is a bit '1'. Correct according that for the *Raspberry Pi*, a signal of 3.3V means a '1' as well.

Second Checking

After this first checking is time to check the same but with a waveform generator. The cable that was providing voltage to the *T1IN* is removed, and instead of that a new cable is connected in the same position, that is also connected to the waveform generator. This new signal has +3.3V of amplitude and +1.25V of offset. This offset is added to try to have more than +2V in the high part of the signal. If the offset is not set the signal will be between -1.65V and +1.65V.

The new circuit setup can be checked in the figure 63 and the result can be seen in the figure 64.

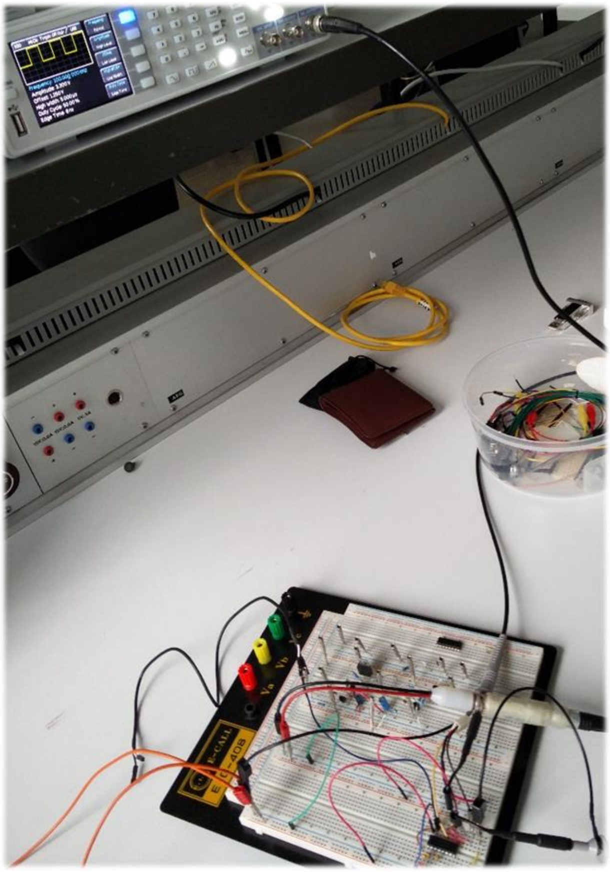


Figure 63 – Circuit 1 ready to be checked 2.

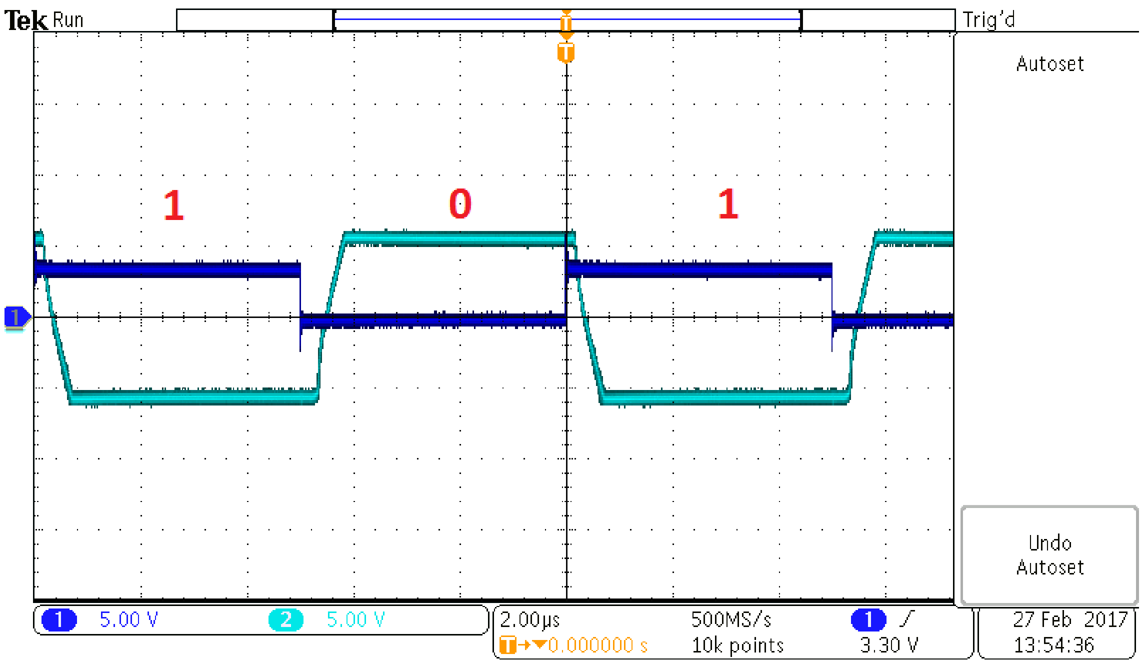


Figure 64 – Oscilloscope screenshot 2.

The output signal has the same voltage level as before, -5V approximately, when the incoming signal is higher than +2V.

And it has also +5V approximately when the incoming signal level is lower than +0.4V. The figure is modified a little bit to see more clear when a bit '1' or a bit '0' are sent.

Also, is interesting to comment the little delay in terms of time in the output signal when the voltage changes. This delay is called *Slew Rate*, and can be found on the specifications of the MAX3232 transceiver ([35]). "The *Slew Rate* is defined as the change of voltage per unit of time." [38]

Third Checking

After this second checking, it's also possible to see with more detail the *Slew Rate* and add into the graph of the oscilloscope a third signal.

In this third checking a connection between the pins 13 and 14 is made, so the same output signal of before is now going into the transceiver into the *R1IN* pin, the pin for the incoming *RS232* signals.

The third channel of the oscilloscope is connected into the same ground of the circuit as the other two channels, and into the pin 12, the *R1OUT*. The result of the oscilloscope can be checked in the figure 65.

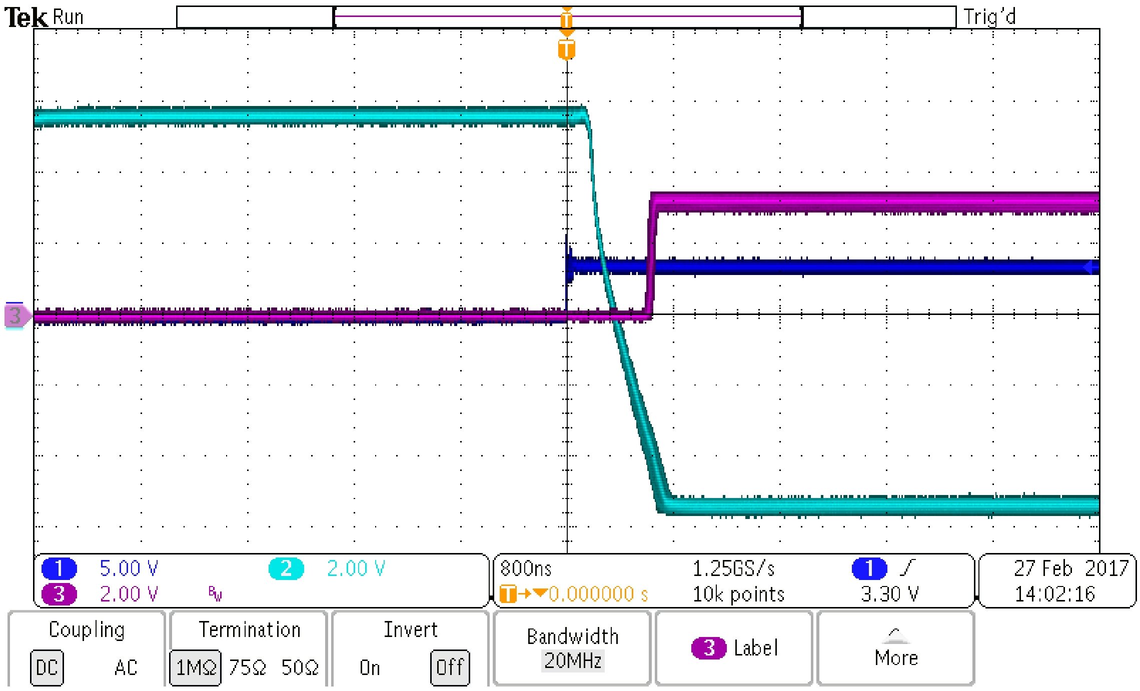


Figure 65 – Oscilloscope screenshot 3.

It must be remarked that the volts per division of the graph in the second and third signals has been reduced, so in comparison with the first signal they are bigger than they are supposed to be. This is made because in this case these two last signals are the ones to be analyzed.

This new third signal in pink color, is the result of introducing the previous output signal (soft blue color) into the *R1IN* pin of the transceiver (*R1IN* is the pin for incoming *RS232* signals) and checking with the oscilloscope what is happening in the output, the *R1OUT* pin.

It can be seen how the voltage changes the same as the second check made before, but this time in the inverse process, from *RS232* to *TLL/CMOS*.

For an incoming signal of approximately -5V, the output signal has a +3V level approximately. In terms of digital communication that represents a bit '1', as explained before.

SOFTWARE VALIDATION WITH THE HARDWARE

Before checking the previous circuit with all the system and with the all programming code created for this project, some few steps must be followed to configure correctly the *Raspberry Pi*.

The *Raspberry Pi GPIO* 14 and 15 pins are used by default only for access into the shell command of the *RPi*. It is needed to change that in order to use them to interact with the meter.

First, in the command shell of the *RPi*, is it advisable running the command '*ls /dev*' to check if the port '*/dev/ttyAMA0*', that corresponds to the pins used in this project to interact with the meter (*UART* pins), is on the list. Also, a good checking is running the command '*ls -la /dev/ttyAMA0*', because the result shown by this must show that is it possible to write and read using this port.

To change all of this in this project, the command "*sudo nano /boot/config.txt*" is run, and the last line is changed. Instead of '*enable_uart=0*', the 0 is changed for a 1 ('*enable_uart=1*').

In the next cutout screenshot shown in the figure 66 it can be seen all the commands results needed to make sure that the *GPIO* pins can be used for the purposes of this project.

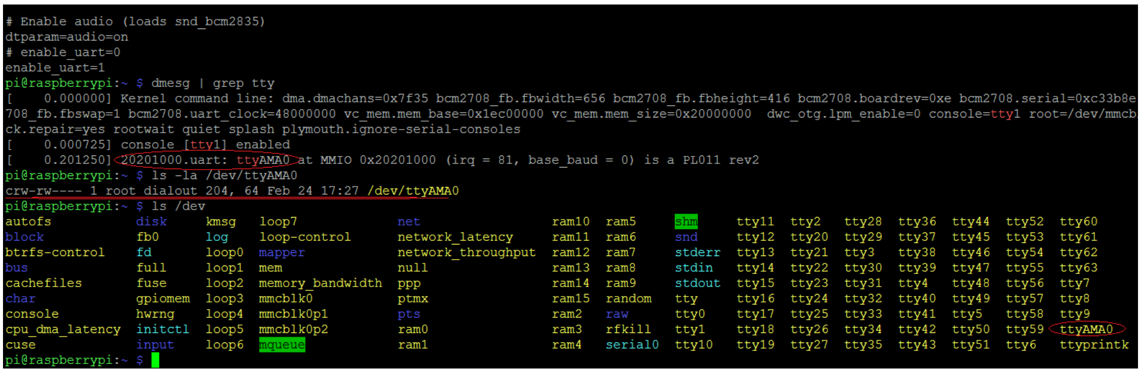


Figure 66 – Command line ttyAMA0 UART enabled.

As explained in the previous software validation, this screenshots are taken on windows in a computer of the *CC4E* building using the *PyCharm* software. The software is run while in the heat meter station the *Raspberry Pi* and the whole system explained in the goal 3 is connected and working. See figure 67 and 68.

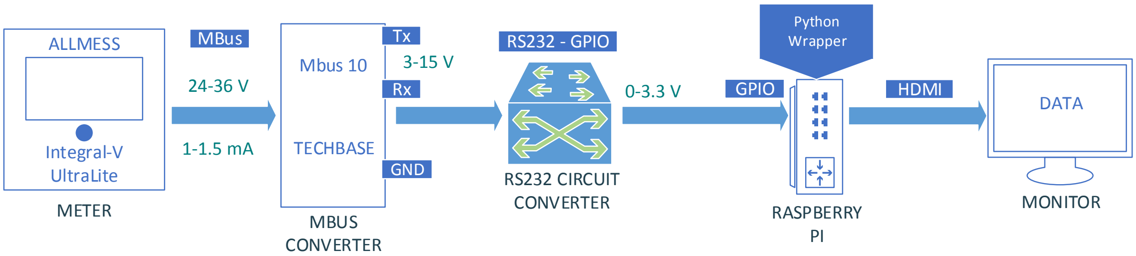


Figure 67 – Main detailed diagram of Goal 3 (2).

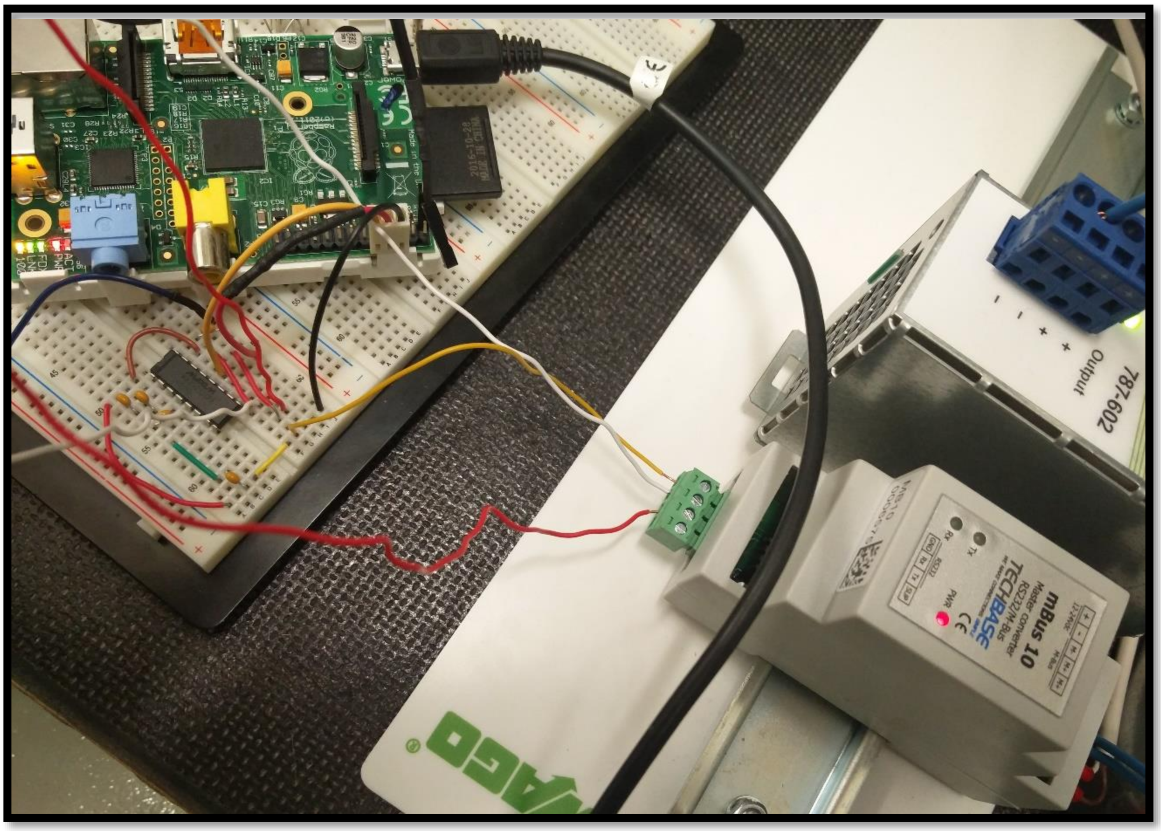


Figure 68 – Implementation of the Goal 3.

The *GPIO* pins used are the 14 and 15, the UART. Also, the 3.3v pin is used to feed the circuit. The *Tx* cable of the *MBus 10* converter is connected to the Rx of the *Raspberry Pi,* and the Rx of the converter to the Tx of the *RPi.*

In the next figures, it can be seen the screenshots validations of the working complete project, with all the software and hardware working. The screenshot of the web page running is not added because the result is the same as before, but the web page with the hardware implementation and with the graphs actualizing itself every one second can be watch also in the videos attached in the CD.

See figures 69 and 70.

```
pi@raspberrypi:~ $ sudo mbus-serial-scan -b 2400 /dev/ttyAMA0
Found a M-Bus device at address 0
^Cpi@raspberrypi:~ sudo mbus-serial-request-data -b 2400 /dev/ttyAMA0 0
<MBusData>

    <SlaveInformation>
        <Id>15262161</Id>
        <Manufacturer>ITR</Manufacturer>
        <Version>23</Version>
        <ProductName></ProductName>
        <Medium>Heat: Outlet</Medium>
        <AccessNumber>80</AccessNumber>
        <Status>00</Status>
        <Signature>0000</Signature>
    </SlaveInformation>

    <DataRecord id="0">
        <Function>Instantaneous value</Function>
        <Unit>Fabrication number</Unit>
        <Value>15262161</Value>
        <Timestamp>2017-03-17T19:13:36</Timestamp>
    </DataRecord>

    <DataRecord id="1">
        <Function>Instantaneous value</Function>
        <Unit>Energy (kWh)</Unit>
        <Value>191</Value>
        <Timestamp>2017-03-17T19:13:36</Timestamp>
    </DataRecord>

    <DataRecord id="2">
        <Function>Instantaneous value</Function>
        <Unit>Volume (1e-2  m^3)</Unit>
        <Value>50092</Value>
        <Timestamp>2017-03-17T19:13:36</Timestamp>
    </DataRecord>
```

Figure 69 – Final receiving data.

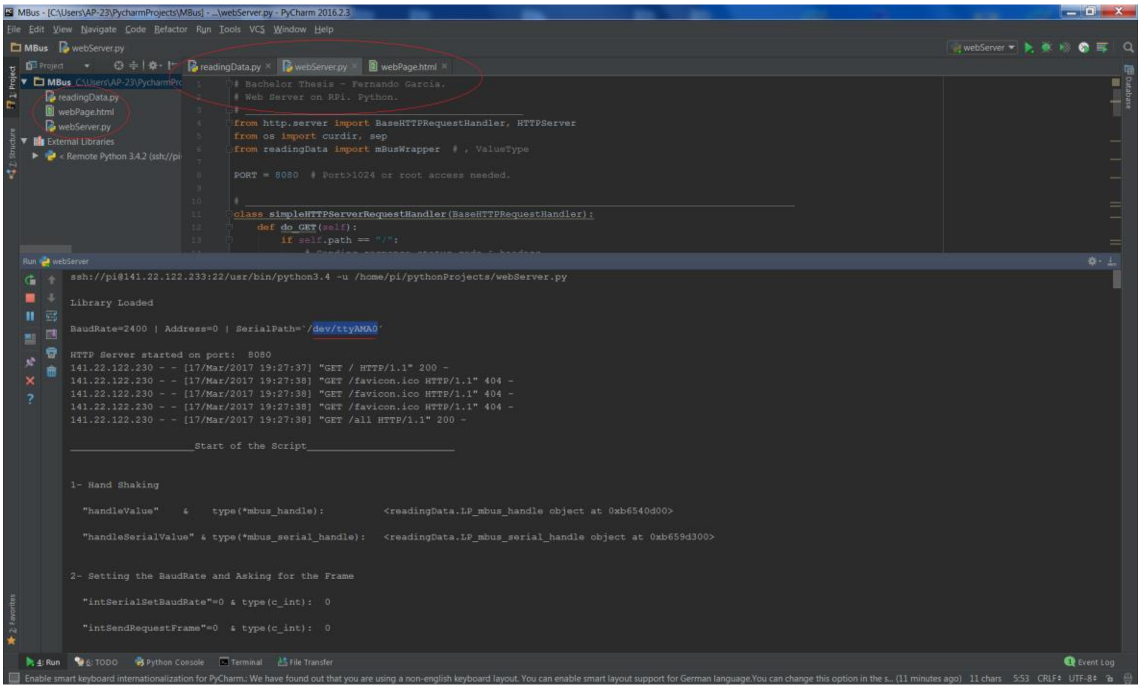


Figure 70 – Final receiving data (2).

ADDITIONAL HARDWARE IMPLEMENTATION

It this project, it is also attempt to replace the *Meter Bus* converter made by *TechBase*, as explained before following the *UNE-EN 1434-3 Spanish rule*, an official document.

The diagram circuit can be checked in the previous figure 56. The circuit is double checked, making sure that everything is correctly connected. After connecting it into the main circuit of our project, the *Raspberry Pi* is detecting something connected in the '*/ttyAMA0*' port, but the data is not received correctly.

This circuit is called circuit 2 in this project. In the next figures 71 and 72 it is shown how the circuit is built and the test in the stand of the *CC4E* building.

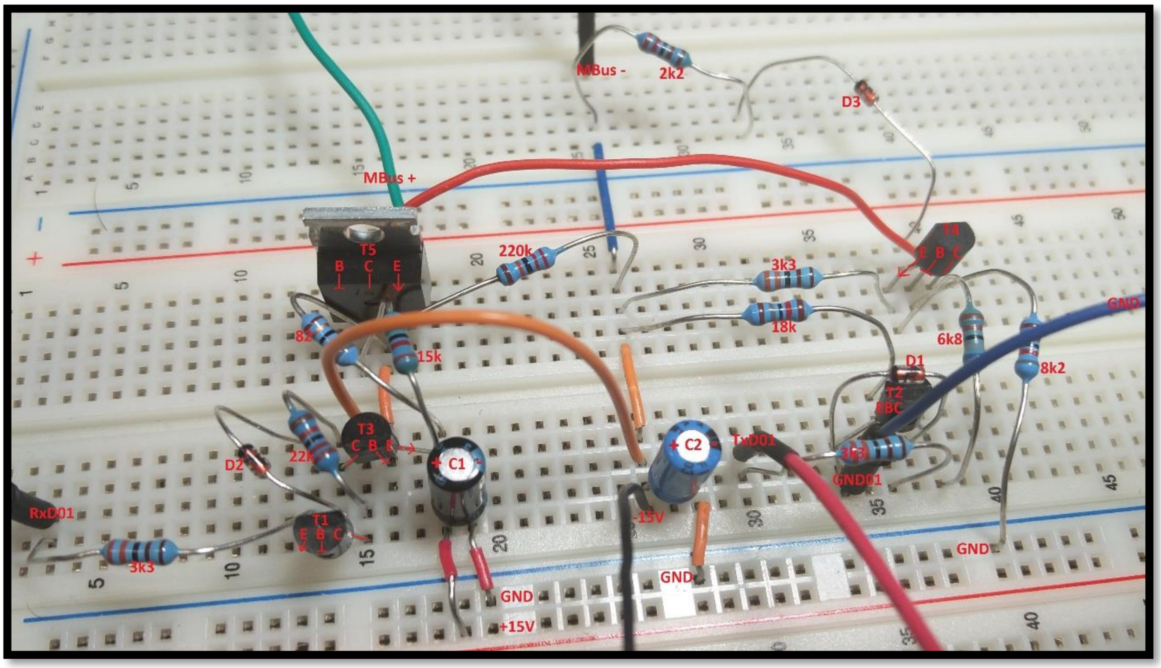


Figure 71 – Circuit 2 built.

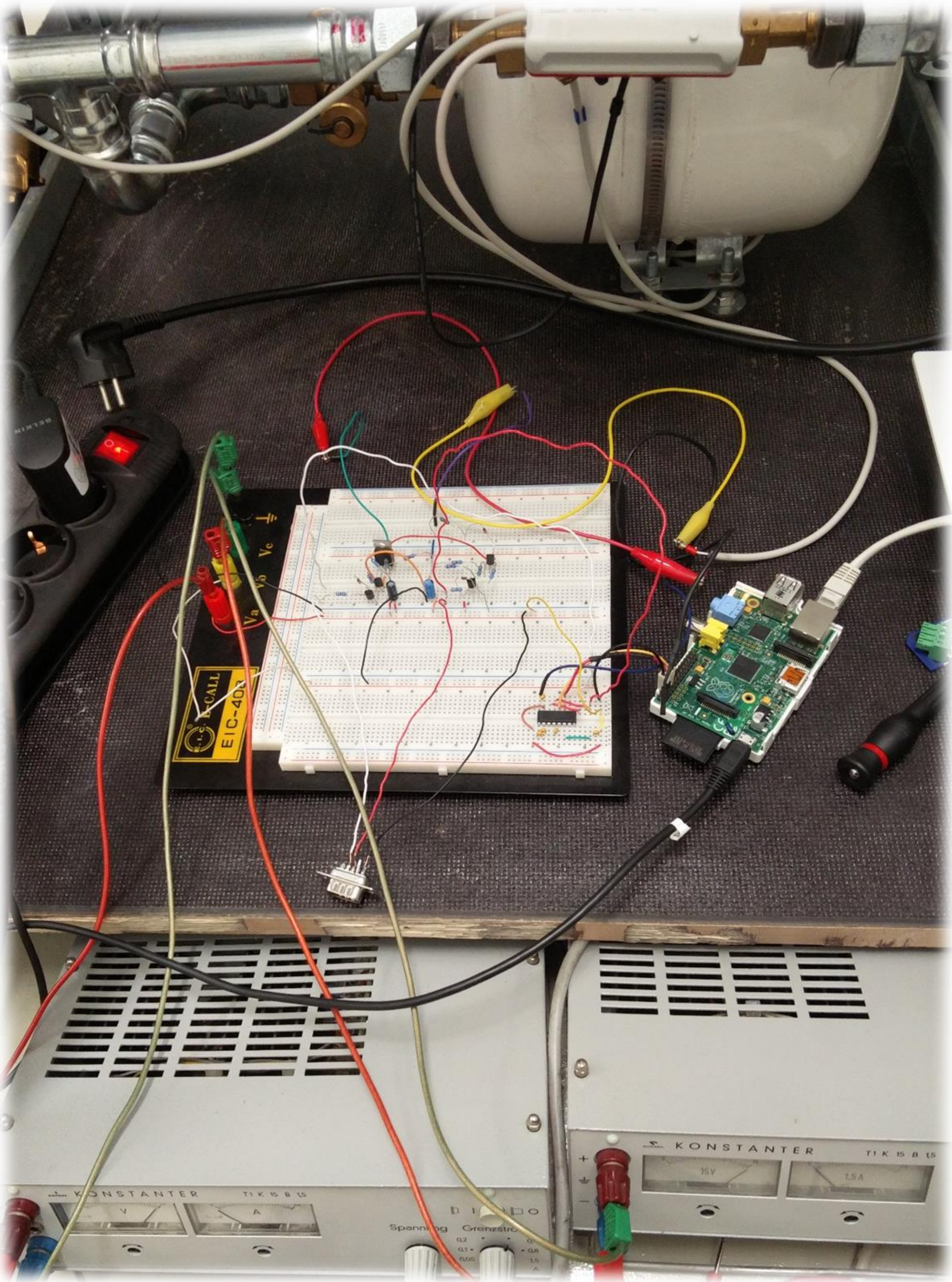


Figure 72 – Converter circuit 2 built.

The circuit is fed with +15v and -15v as it can be seen in the bottom of the previous figure. The outputs of the *Meter Bus* are connected to the inputs of the circuit, and the output of the circuit is connected to the output of our circuit 1 built before.

With the *RPi* correctly working and the circuit 1 correctly working as well, the *RPi* detects something connected but is not able to detect any meter connected.

Evaluation

The main goals proposed at the beginning of this project have been achieved, not without some difficulty. It has been possible to create a program in *Python* code that allows reading the data, and that data can be obtained from any part of the building where the project is made: the *CC4E*. This data is also readable in a very clear and dynamic way, showing not only an instant data, but also all the data over time, uninterruptedly.

In addition, the previous *PLC* has been successfully replaced by a *Raspberry Pi*, which offers many more possibilities for the future and it is also way cheaper, and the small circuit between the same and the meter has tested successfully.

What has not been done despite trying, is replacing the expensive *Meter Bus* converter, with a much more complex circuit than the previous one, but that would have saved a lot of money at the station. The circuit seems to work properly and both the meter and the *Raspberry Pi* are not affected by it, but more investigation is needed in order to see if it is just a Software problem.

OUTLOOK

The project can continue in several ways. The RPi offers a world of possibilities with which to investigate more about how to make the station a much more dynamic and intelligent.

In addition, the circuit that has not been successfully replaced can be tried again, starting from the bases and the initial investigation carried out in this project. With all the research and verification made in this document it is easy for a next person to continue from this work.

Another possible would be to replace the small circuit implemented in the protoboard by a printed circuit, more robust, with which it can be fixed in a place of the building.

Also, the meter in the *CC4E* is unconfigured, so the program can be modified and used with a configured meter, that provides real data. With this change the code can improve a lot. Another possible way to improve the code is adding some features to the web page, or asking to the client for the parameters of the meter that it is request, so is the user the ones who asks for it.

Conclusion

This thesis improves a previous heat transfer station allocated in the CC4E building, in the city of Hamburg, in order to make this station a smarter station.

To achieve that, the station has been made accessible from any part of the building using low cost components, and it uses the same programming language that used by the engineers of the center.

The components used in order to save as much money possible has been a Raspberry Pi and one own designed circuit. Also, the code inside the RPi is a program that allows the flexibility and manageability, and allows the possibility to improve the program a lot.

This thesis is also made as the first step of a further development, so another's future researchers of the C4DSI can continue improving the smart heat transfer station.

It is a great beginning of a project that can become very beautiful and exciting. With the bases that are established on this thesis, great progress can be made.

References

[1] "DataSheet Meter," 16 03 2017. [Online]. Available: http://www.allmess.de/fileadmin/multimedia/alle_Dateien/DB_P0014_Integral-V_UltraLite_TS1015.pdf.

[2] "DataSheet MBus Converter," 16 03 2017. [Online]. Available: http://energycare.dk/wp-content/uploads/2014/05/ENERGYCARE-CONVERTER-TECHBASE-MBus10_v2_eng.pdf.

[3] "Digitus Converter," 16 03 2017. [Online]. Available: http://www.digitus-professional.com/it/products/computer-accessories-and-components/computer-accessories/serial-and-parallel-adapter/da-70155-1/.

[4] "Raspberry Pi WebPage," 16 03 2017. [Online]. Available: https://www.raspberrypi.org/blog/raspberry-pi-2-on-sale/.

[5] "Monitor Photo," 16 03 2017. [Online]. Available: https://pixabay.com/en/photos/monitor/.

[6] "Meter Bus Info," 16 03 2017. [Online]. Available: https://en.wikipedia.org/wiki/Meter-Bus.

[7] "MBus User Guide," 16 03 2017. [Online]. Available: https://www.openmuc.org/m-bus/user-guide/.

[8] "RS232 Wikipedia," 16 03 2017. [Online]. Available: https://en.wikipedia.org/wiki/RS-232.

[9] "RS232 Info," 16 03 2017. [Online]. Available: https://www.commfront.com/pages/3-easy-steps-to-understand-and-control-your-rs232-devices.

[10] "Raspberry Pi modelB Web Page," 16 03 2017. [Online]. Available: https://www.raspberrypi.org/products/model-b/.

[11] "Raspberry Pi Wikipedia," 16 03 2017. [Online]. Available: https://en.wikipedia.org/wiki/Raspberry_Pi.

[12] "GPIO Wikipedia," 16 03 2017. [Online]. Available: https://en.wikipedia.org/wiki/General-purpose_input/output.

[13] "GPIO UART Pins," 16 03 2017. [Online]. Available: http://www.sustainablenetworks.org/CIS508/?page_id=1585.

[14] "Raspberry Pi Guide," 16 03 2017. [Online]. Available: https://www.raspberrypi.org/learning/software-guide/.

[15] "libmbus rSCADA Web Page," 16 03 2017. [Online]. Available: http://www.rscada.se/libmbus/.

[16] "Library Install RPi," 16 03 2017. [Online]. Available: http://www.raspberrypi-spy.co.uk/2012/05/install-rpi-gpio-python-library/.

[17] "Install libmbus Library Web Page," 16 03 2017. [Online]. Available: http://bends.se/?page=anteckningar/automation/m-bus/libmbus.

[18] "MBus Domotiga Project," 16 03 2017. [Online]. Available: https://www.domotiga.nl/projects/domotiga/wiki/M-Bus.

[19] "RPI Serial info," 16 03 2017. [Online]. Available: http://elinux.org/RPi_Serial_Connection.

[20] "RPI Serial Connection," 16 03 2017. [Online]. Available: http://elinux.org/RPi_Serial_Connection#Console_serial_parameters.

[21] "Library Wikipedia," 16 03 2017. [Online]. Available: https://en.wikipedia.org/wiki/Library_(computing).

[22] "Ctype WikiBooks," 16 03 2017. [Online]. Available: https://en.wikibooks.org/wiki/Python_Programming/Extending_with_ctypes.

[23] "Ctypes Wikipedia," 16 03 2017. [Online]. Available: https://en.wikipedia.org/wiki/Dynamic-link_library.

[24] "Wrapper Wikipedia," 16 03 2017. [Online]. Available: https://en.wikipedia.org/wiki/Wrapper_library.

[25] "Wrapper Definition," 16 03 2017. [Online]. Available: http://searchmicroservices.techtarget.com/definition/wrapper.

[26] "Pointer Definition," 16 03 2017. [Online]. Available: https://en.wikipedia.org/wiki/Pointer_(computer_programming).

[27] "Class Definition," 16 03 2017. [Online]. Available: https://en.wikipedia.org/wiki/Class_(computer_programming).

[28] "Ajax Definition," 16 03 2017. [Online]. Available: https://es.wikipedia.org/wiki/AJAX.

[29] "HighCharts Web Page," 16 03 2017. [Online]. Available: http://www.highcharts.com/.

[30] "JQuery Web Page," 16 03 2017. [Online]. Available: https://jquery.com/.

[31] "UART Wikipedia," 16 03 2017. [Online]. Available: https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter.

[32] "MAX3232 Diagram," 16 03 2017. [Online]. Available: http://s288.photobucket.com/user/ninexunix/media/Max3232.png.html.

[33] "GPIO Circuit Info," 16 03 2017. [Online]. Available: http://elinux.org/RPi_Serial_Connection#Console_serial_parameters.

[34] "Circuit 1 Info," 16 03 2017. [Online]. Available: http://codeandlife.com/2012/07/01/raspberry-pi-serial-console-with-max3232cpe/.

[35] "MAX3232 DataSheet," 16 03 2017. [Online]. Available: http://www.alldatasheet.com/datasheet-pdf/pdf/73152/MAXIM/MAX3232CPE.html.

[36] "Inverter Gate Info," 19 03 2017. [Online]. Available: https://en.wikipedia.org/wiki/Inverter_(logic_gate).

[37] "TTL Logic Levels," 16 03 2017. [Online]. Available: https://learn.sparkfun.com/tutorials/logic-levels/ttl-logic-levels.

[38] "Slew Rate Definition," 16 03 2017. [Online]. Available: https://es.wikipedia.org/wiki/Slew_rate.

Appendix

This Bachelor Thesis contains an appendix of program listings, hardware descriptions etc. on a CD (disk or supplementary booklet). This Appendix is deposited with Prof. Dr. Eng. Franz Schubert.

Declaration

I declare within the meaning declare within the meaning of part 16(5) of the General Examination and Study Regulations for Bachelor and Master Study Degree Programmes at the Faculty of Engineering and Computer Science and the Examination and Study Regulations of the International Degree Course Information Engineering that: this Bachelor Thesis has been completed by myself/ourselves independently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources

Hamburg, 20. March 2017

Signature ________________________