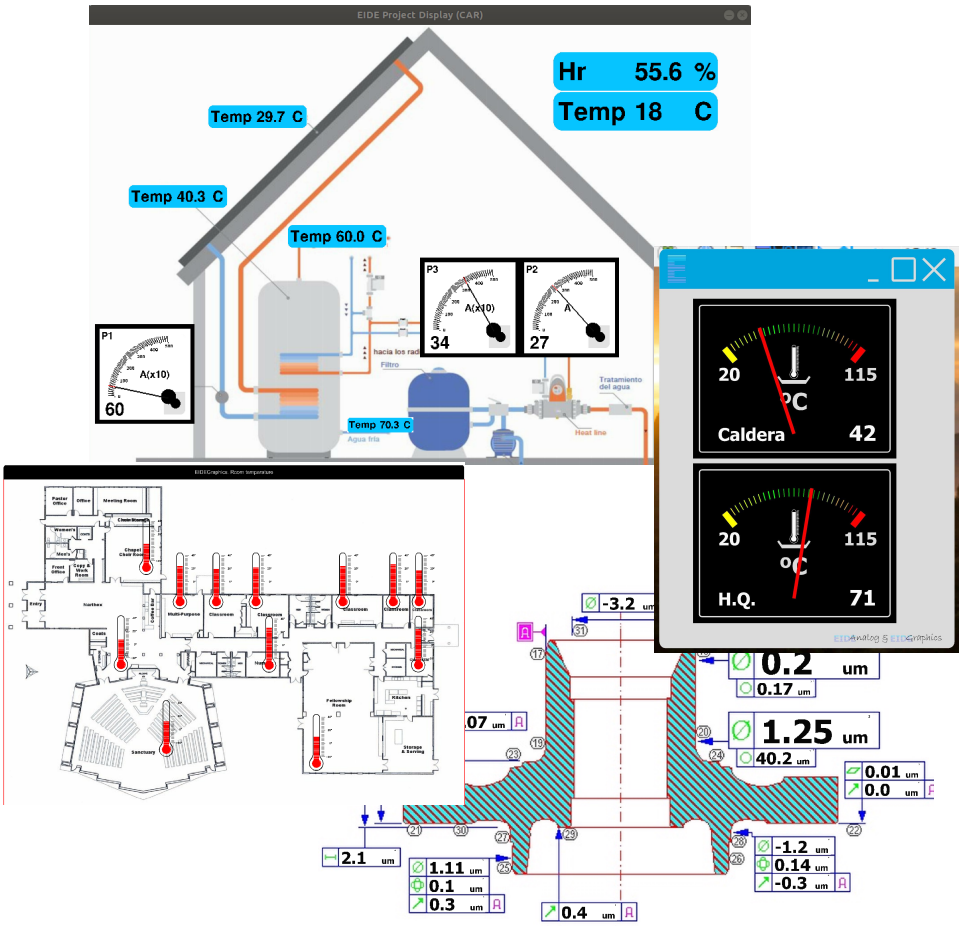
Data acquisition using Python and Raspberry Pi



EIDEAnalog library
(Making of)

Revisions

N°	Date	Paragraph	Description
0	May the 7th, 2020	-	First chapter issued.
1	May the 10th, 2020	-	Second chapter issued.
2	May the 18th, 2020	-	Third chapter issued. Misspelling and untranslated text fixed.
3	June the 8th, 2020	-	Fourth chapter issued.
4	June the 30th, 2020	-	Fifth chapter issued.

Table of Contents

1.- Brief introduction.

We will explain through this 'blog' how a *library* to help capturing physical parameters (analog-digital conversion) has been conceived and developed using Python as a programming language and OOP (Object Oriented Programming) as a technique. The hardware to be used is A) a Raspberry Pi card without added hardware for the oneWire bus, B) the same Raspberry plus an ADS1115 type analog to digital converter and C) the Raspberry connected to an Arduino ProMini; for the first option probes used are of the DS18B20 type, for the other two *combos* half a dozen conventional sensors are used - temperature, voltage - which will be opportunely depicted. As will be exhaustively explained in the blog, much emphasis is placed on the decision of the pattern of (the classes that make it up) the library and its development, issues that are usually the most complex in this type of programs.

The blog may be of interest:

- As a guide to programming practices for intermediate level courses. It may be even possible that in certain university degrees it will serve for the same purpose.

- For fans of (or professionals in) programming who wish to improve or test their knowledge of OOP.

- Engineers in the ‘data acquisition’ field. Although the hardware used has not an industrial grade (not, at least, the one that has been specifically used to prepare this blog), it is not ruled out that it may be of their interest.

Descriptions along the blog are intended to be as concise as the subject permits, which, as will be pointed several times, is not a simple issue.

(Should you just want to familiarize yourself with the use of the library and not with its design, please go directly to point 8, "Use of the library").

1.1.- Blog declared goal.

The stated objective of the blog is, therefore, to develop a -small- library written in Python to ease the use of various types of AD adapters, the popular ADS1115, the "oneWire" bus and an Arduino Promini. The first and the third are hardware additions to the Raspberry, while the oneWire bus is a Raspberry *native* option: loaded the driver, you can enable the pins of the GPIO connector that you deem appropriate to implement it.

Configuring the Raspberry: Depending on the version of the Raspberry you have and the Raspbian *distribution*, you will have your Raspberry configured for the i2c bus (necessary for the ADS1115 to work) and/or oneWire (necessary for the bus of the same name) and/or serial communication (necessary for the Arduino). See Annex II, “Configure your Raspbian/ Raspberry” for details.

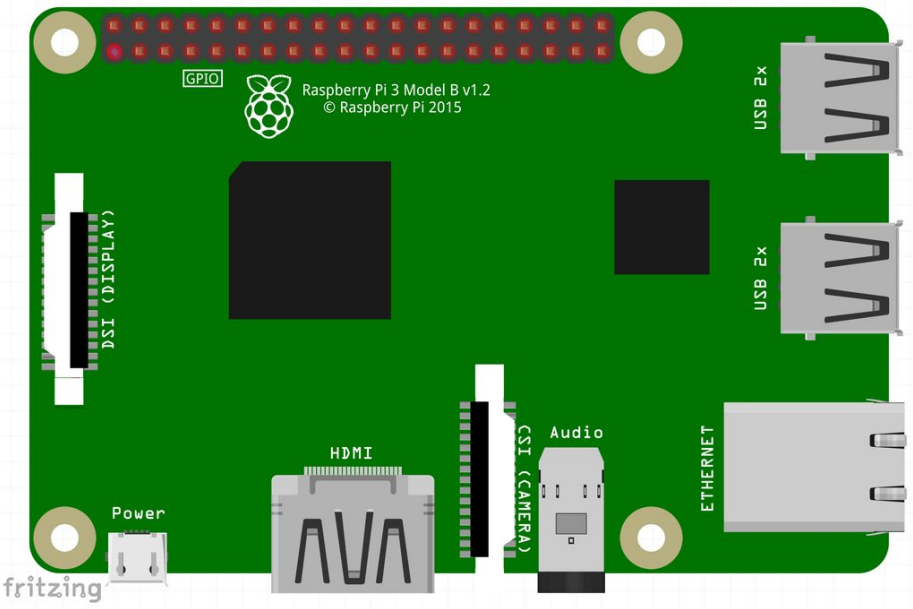


Figure 1. Raspberry 3B.

Yet, the blog is not limited to this: although the library is actually developed -as it is written- as outlined in the previous paragraph, there is another objective as important as the stated one, at least for who wants to learn some analysis and OOP programming. Let’s hope that the trip, encompassing the needs, analysis, problem decomposition into classes and the discussion of the final pattern of the library, is as useful as its -trip’s one- destination, which is the mere use of the library.

1.1.1.- OOP design pattern.

To focus the question: when addressing an OOP development analysis, one of the main issues is the *architecture* of the system, its pattern; i.e.: How do I find the classes that will be instantiated to give rise to the objects that make up my program?. How much should I crumble -granularity- the problem?. How many classes?. What are going to be the relationships among them? And the objects among themselves? How is it documented?

The specific matter that is addressed in this blog -the library itself- is quite simple: you can solve it with hardly a dozen classes (as we will see later). Its best feature is that it is fully developed (that the library code is available https://github.com/Clave-EIDEAnalog/EIDEAnalog -and that it works!). The reader, whose knees will probably tremble sometimes, is at least sure to come to an end.

In other words, the best virtue of this blog is that it is a fully developed example that does not limit to describe the basics of the OOP analysis, a complex issue that requires much more than this blog. If you want to follow it -the blog- please keep this in mind: it is a developed (programmed, working) example. Nothing more, nothing less.

1.1.2.- Data acquisition systems. A/D conversion.

If it is a matter of doing OOP pedagogy, one could have chosen a subject other than A/D (analog to digital) conversion . What does this have -that others do not- to have been chosen as the common thread of the blog?

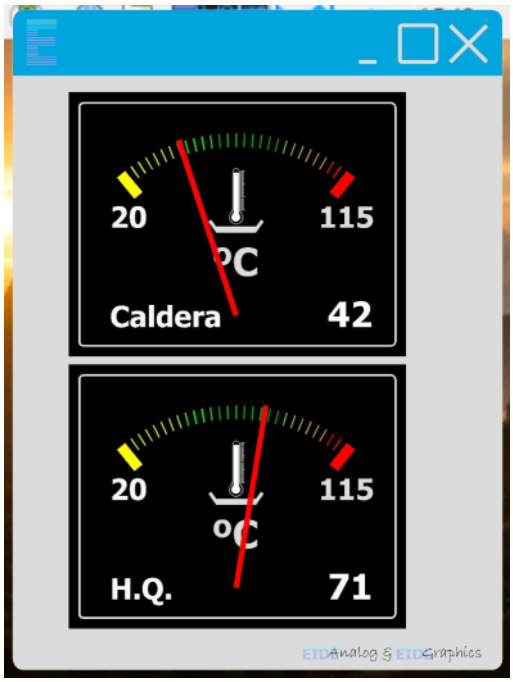


Figure 2. Raspberry data acquisition system.

That the blog authors know it well (and are perfectly aware that choosing a library to "acquire physical parameters", as stated in the third line of the first point of this blog, will

take away a good part of potential followers who, a different topic chosen, perhaps would have been encouraged to follow it).

On the other hand, may be that the matter also is attractive, as the result is connecting sensors to a computer (an issue that, in many ways, can be considered eccentric). This is an issue of interest to a good number of activities, both in the field of industry and training, and in this - that of training – to all the engineering and degrees in physical or chemical sciences. At the end of the day, and except for the subtleties of the A/D conversion device itself -which are specific to this discipline- the remaining is not much more than algebra and binary logic; once you are satisfied with the purpose of what you are doing -the library-, the way of doing it -analysis- is valid for similar problems, or of a similar approaches.

1.1.3.- Blog development. Pedagogics.

We have been careful in the writing of the blog, using a pedagogical style. It is not just a matter of writing some classes so that they can be incorporated into a project by a “copy-paste” (yet this can be done without a problem - and it works). The main goal of the blog is, as mentioned, to help anyone who wants to analyze a case on how to approach the analysis and development of a library using OOP techniques.

1.1.4.- Conclusions. Goals.

By the end of the blog you should have acquired certain skills regarding OOP (see 2.5.3, “OOP. Conclusion."). In addition you will learn:

- Computer data acquisition (basic concepts). ADC (Analog to digital converters).
- User level Raspberry management. (Very) basic concepts of Linux.
- (Rudiments of) operation of oneWire, i2c buses and serial communication (UART).
- Functioning of some commercial (and popular) sensors.
- Thermistors. NTC. Tabulated sensors.

1.2.- Basic knowledge to follow the blog. Prerequisites.

Blog authors have experienced more than once manuals that begin -enthusiastically- explaining that to use the computer mouse you have to operate it with your hand, move it across the pad and press the left button to select or the right one for options (and that can be configured the opposite for left-handed); inevitably the manual -or whatever- abandons this level of detail on the second paragraph to, much worse, give indecipherable instructions just three or four paragraphs later (each of them -of the indecipherable instructions-, deserving for themselves a manual).

We will try not to repeat this error; It would be delusional to think that in order to follow this blog you do not need any prior knowledge of the subjects being addressed. Whoever that wants to follow it must have a *user knowledge* (whatever this means) of computing, personal computers. As we use a Raspberry as hardware you should familiarize with it (of course, whenever the Raspberry is referred to, we are also including the Linux operating system).

Figure 3. ADS1115 tied to a Raspberry.

Nor you should get disappointed for getting your hands a little dirty: whatever the option you choose to implement the blog content you will have to connect something to the Raspberry -see Figure 3 above- and tighten some screws (terminals). We encourage you to connect things to each other: the Raspberry with the A/D converter, the probes to the converter, the Raspberry's power supply (it's like the one of a mobile phone), ... nothing too complicated.

Let’s go step by step.

1.2.1.- Electronics. A/D converters. Binary numbers.

You should know how to use a polymeter; it is worth knowing how to measure DC voltages. A review on Ohm's law won't hurt (although it is not strictly needed for the blog).

Figure 4. Polymeter.

The ADC (analog to digital conversion) issue is a bit trickier.

When it comes to acquiring values *quickly*, the A/D conversion issue is truly a challenging one. *Quickly* means thousands of times per second -or more; or much more- which is the case, for example, on an audio studio to capture music without losing quality (44,100 times per second); similar speeds are required in many problems in the industry of all kinds, in research, telecommunications, image (much more, this case).

This is not this blog case; it is enough understanding that it is all about converting real world parameters to numbers much more calmly. Although we will have to delve into how the A/D converter is controlled, this is a more *administrative* than an engineering problem, as we will see when it comes to it.

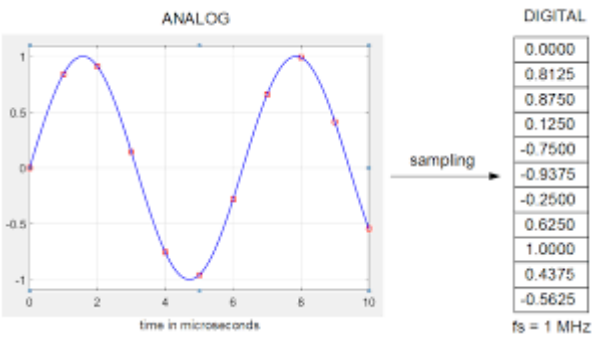


Figure 5. A/D conversion.

(As some WEB page explains, you sitting next to a mercury column thermometer taking notes every, say, 10 seconds, becomes an analog-to-digital converter: you are transforming -and recording- analog values to numbers).

The point is that the computer works with the base 2 numbering system (the -in-famous 1010001010 ...), which in this blog becomes the main scientific problem (at least different from the basic one: the analysis in OOP). If you already master this matter, congratulations, please go to the next paragraph; should you not, you have to make some investigation: entries of wikipedia for "*binary*" and "*bitwise*" are a good start. You also have to know what the hell the "two's complement" format is.

A basic knowledge of the terms *byte* and *word* are necessary too.

Incidentally, the hexadecimal numbering system is also mentioned; take a look at it on wikipedia.

1.2.2.- Raspberry.

Everything you need to know about the Raspberry hardware, at least to start with, is 1) how to connect it and 2) how to configure the µSD (microSD). There are literally thousands of WEBs that explain how to connect it; even the accompanying leaflet includes it.

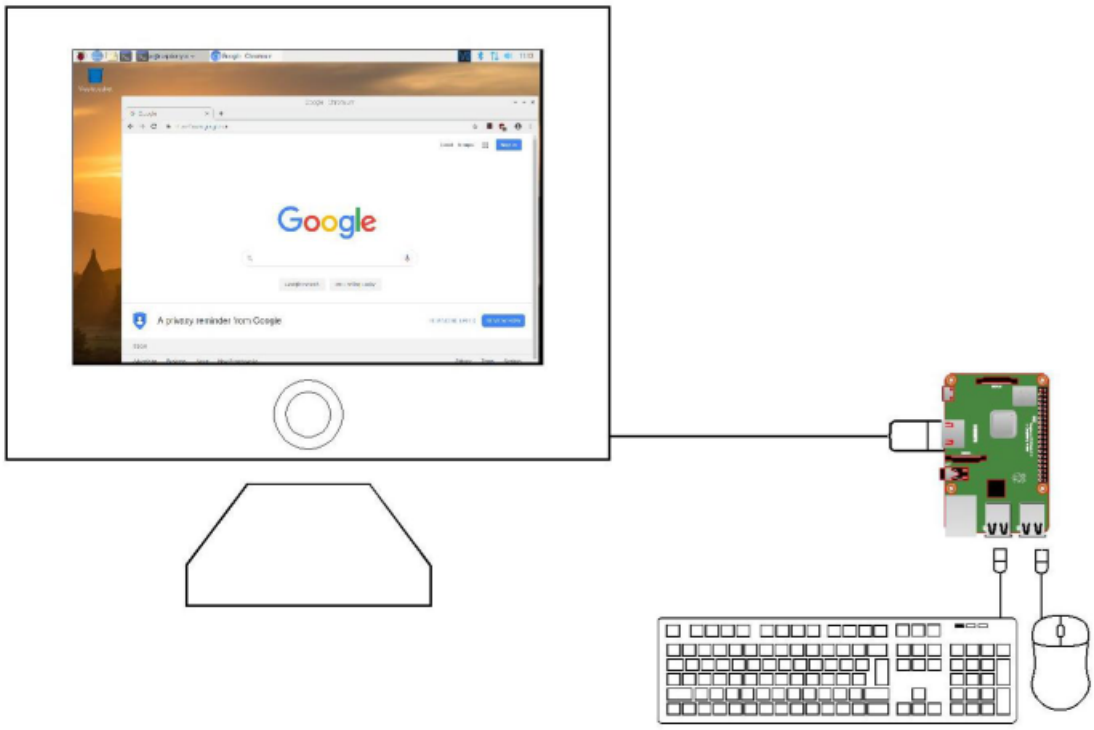


Figure 6. A Raspberry acting as a laptop.

(The easy way to get started is to connect an HDMI monitor to the video port, a keyboard and a USB mouse to two of the 4 USB ports it has, insert a properly configured µSD memory card and power it all.).

The µSD thing is a bit harder: see Annex II, “Raspbian Configuration” for more information.

Although as we present the different subjects they will be explained in detail, it does not hurt finding out what a oneWire, an i2c and a serial communication buses are. Take a look at the wikipedia entries for “oneWire”, “i2c” and “Serial Communication”.

1.2.3.- Linux.

The Raspberry has the enormous advantage of working with Linux ... should you not know Linux you miss it. But do not panic, it is not necessary to become an expert to use it, and using the graphical interface is quite simple to start with.

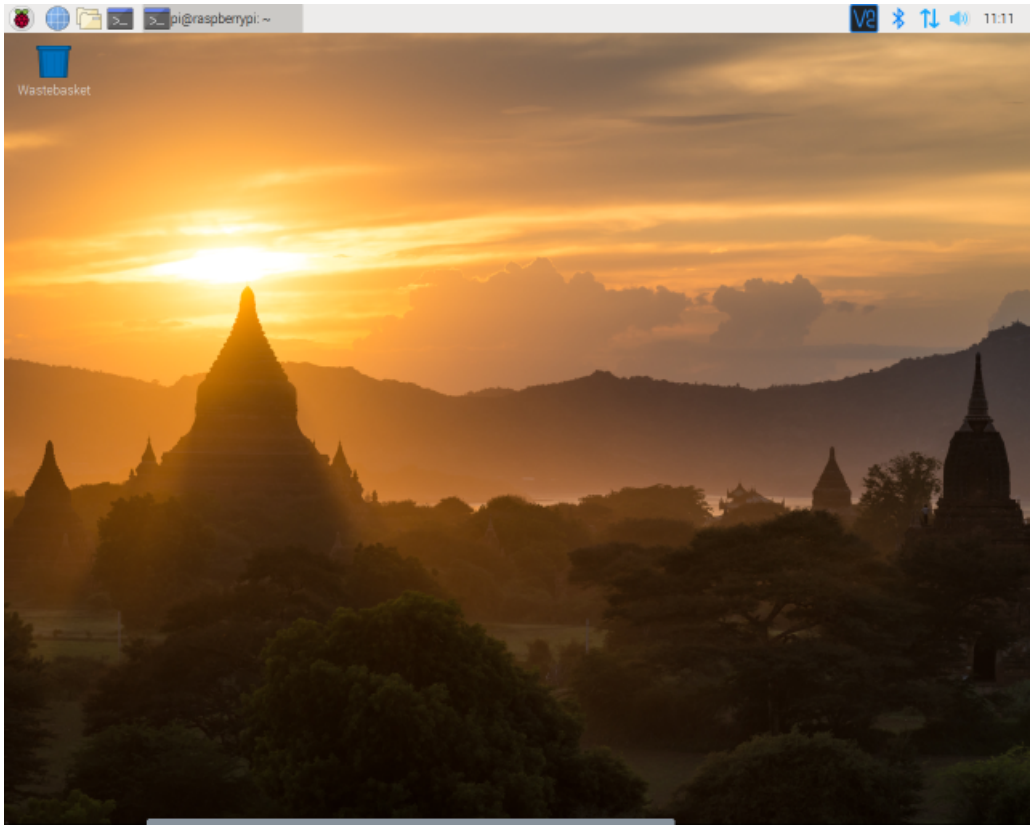


Figure 7. Linux interface (Raspbian).

To following the blog, you don't have to know much about Linux: should you have a well configured *distribution*, a user knowledge should be enough to start (see annex II, “Raspbian configuration”).

As in the precedent paragraph, if you manage with Linux, congratulations, please go to the next paragraph.

Else, we recommend you to learn a bit:

- It is in your best interest to read something to at least familiarize yourself with the terms ("Debian", "Jessie", "Raspbian", "Noobs", "Ubuntu", ... don't be alarmed, they are very much the same). Read the article from wikipedia for “Raspbian” (be patient: you won't understand many things at first, but you have to get a bit comfortable with them); Raspberry's website (“www.Raspberry.org”) is not the best in the world, but having a look at it is a good idea either.

- “Terminal": the term refers to the use of the operating system (Linux, in this case, although it can be applied to any O.S.) by means of a "command line". The elders in the city would remember the black screen of MS-DOS prior to Windows: that is the

"terminal " (and, by the way, a few MS-DOS commands are copied from Linux; they are the same).

- It is quite convenient that you know some commands to configure Raspberry things (again the reader is referred to Annex II, “Raspbian Configuration”). See
  - https://maker.pro/linux/tutorial/basic-linux-commands-for-beginners
  - https://www.digitalocean.com/community/tutorials/an-introduction-to-linux-basics
  - https://www.freecodecamp.org/news/the-best-linux-tutorials/
  - https://www.tutorialspoint.com/unix/
  - The book “The Linux Command Line: A Complete Introduction. William E. Shotts Jr ” is excellent.

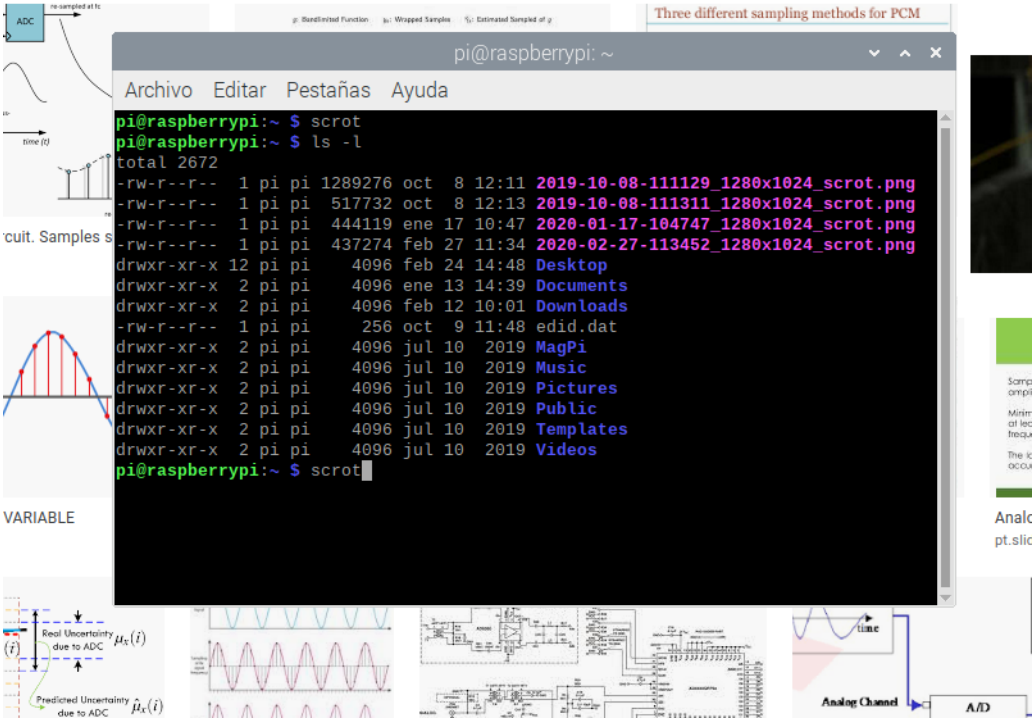


Figure 7. Linux terminal session (Raspbian).

Backup copies: One of the main advantages of Linux is the ability of recovering from the disaster of the loss of the S.O. itself. Using *another* operating system, the hard drive failure leaves you out in the open. In the case of the Raspberry, the equivalent would be the failure of the µSD (or the Raspberry itself, the worst disaster); The good -excellent- new is that it is possible to have a copy of the µSD simply by using another µSD. If you are going to mess around with the Raspberry, which we recommend, it is convenient that you have an updated backup by making periodic copies from one µSD onto the other: a few dollars in exchange of a priceless tranquility.

1.2.4.- Python. OOP.

This blog is not a Python tutorial (nor an OOP one). To follow it without too many shocks -without having to return frequently to more basic issues than those required to understand what is being developed- it is necessary to have a reasonable knowledge of Python. And you have to have written half a dozen simple Python programs.

(The term "reasonable" in the previous paragraph is used with some bitterness because, who is able to say what a *reasonable* knowledge of Python is?).

Python is one of the easiest (the easiest?) programming languages to learn, and it is extremely powerful. And it is not just that its syntax allows coding in a way that summarizes in a single line of, say, 30 or 40 characters what using other languages requires many lines: Python has implemented as native all the instructions and architecture of any other modern language, and you have to take into account what is added by the libraries included in the normal download. It also has structures -dictionaries, for instance- and/or novelties -class attributes- that at the time, and still today, makes it different from the closer environment.

Python is, in summary, a world to explore. So let's see who is the brave woman or man who defines what a *reasonable* knowledge of Python is.

So we will proceed by extension: let’s take the Python WEB reference tutorial (“https://docs.Python.org/3/tutorial”) as a guide. You need to know:

- How to create and manage numeric variables, their types -int(eger), float, ..- and formats -decimal, hexadecimal, binary, ...-. All the arithmetic and logical operations. You need to understand what “None” is.

- How to create and manage strings (the basics: what a string is).

- "Flow control". The above mentioned Python tutorial is not among the best of the foundation (“Python Software Foundation”): the first mention it makes on this concern -flow control of a program- in point 3.2 is undoubtedly unfortunate: one cannot understand why the –very first- example is implemented with a *while* without any previous anesthesia. To follow the blog smoothly go to point “4”: you have to understand all the contents in chapters 4.1 to 4.6, all.

- Lists, tuples and dictionaries: in depth. You should be familiar with all the content of tutorial points 5.1.1, 5.1.2. and 5.5. Along the library they are used a lot to iterate: please study in depth the content of point 5.6 and review the point 4.2.

- Modules: you need to understand what the "import" command is for and what is meant by "standard modules" (6.2).

- **Errors and exceptions**. They deserve their own comment: the implementation of a library that is worth of that name should not be made available to potential users without it having a minimum of *protection* against the mistakes that will surely be made when using it (and the ones that, undoubtedly, will be utterly hidden in the code of the library itself). The counterpart is that the use of error handling clauses

introduces a new complication in the ‘aspect’ of the code, which, at least at first glance, causes the reject of rookies.

A necessary compromise has been reached: the first version of the library (the one that is discussed in depth into this blog and can be downloaded from https://github.com/Clave-EIDEAnalog/EIDEAnalog), has no *error handling* code. A ‘Pro’ version that will have it (open source too) is in preparation. Have a look on chapter 8 of the tutorial, though.

- Classes. Objects: This is the moment of the truth. The terms *class* and *object* should have not secrets for you; and we are off to a bad start, however, if you frown as soon as the first *self* appears or you panic when you see a *__init__* coming floating in the code. Along the blog we neither take everything for granted (related to the concepts of class, object, etc.) nor can we make pedagogy about it with the basic concepts of the subject (which, on the other hand, is done -pedagogy- when the things get complicated: abstraction, encapsulation, inheritance, polymorphism, patterns). On the other hand, the didactic standard of the chapter 9 of the above mentioned tutorial (“https://docs.Python.org/3/tutorial”) leaves, again, something to be desired. And this is not an easy matter (neither the basic question of the OOP technique nor how to start with it). Please, find below a couple of WEB sites that you should study if you want to follow the blog contents without shocks.
    - https://www.programiz.com/Python-programming
    - https://realpython.com

Python download, install and configuration: If you are not familiar with Linux, it is possible that starting up Python on Raspberry involves some difficulty. Although the authors of the blog are unrepentant supporters of Linux -and, therefore, self-conscious detractors of the "other" operating system-, it would be an excess to suggest to the blog follower to study it thoroughly -Linux- as a previous step to start with the blog . The Raspberry startup and use is not complicated at all if you have a good *distribution* (the jargon to refer to the version of the S.O.). There are many vendors on the Internet who can supply you with a properly configured µSD. Linux has a graphical interface similar to that of Windows. See Annex II, “Raspbian Configuration” for complete information.

1.2.5.- Other prerequisites.

- File types: You have to be able to identify and modify a text file (windows *notepad*; Linux *leafpad*).
- Needless to say that you have to know what the ASCII code is.
- Have a look into the wikipedia entrances for NTC and thermistor.

2.- Project: ADC library. Specs & tools..

If after such a lengthy advancements, so many reading recommendations, so many warnings and even threats, if, after all this stuff, you have come this far, congratulations: it is time, finally, to get into the matter.

Let's see, now in detail, what we are going to do and what tools we have to accomplish it.

2.1.- Specs.

We face this project::

1. The goal is developing a library in Python 3 to use a Raspberry as a data acquisition system (conversion of analog to digital variables). The library must allow to select the ADC from three *native* alternatives (alternatively or simultaneously): 1/ the Raspberry Pi native oneWire bus, 2/ A “chinese” board -there is no explicit manufacturer- that mounts an ADS1115, and 3/ an arduino ProMini. The measurements taken by the ADC will be converted to a readable format.

2. The library has to solve connecting several sensor alternatives: those designed for the oneWire bus (see 2.3.1, “oneWire bus”) and *generic* sensors for temperature, pressure, distance, sound, etc. -those ones that convert the measured parameter to volts- in the case of the other two options (ADS1115 and arduino). While to add a new ADC to the library involves writing a new class, new sensor types would be added without modifying the library code. It should even be possible to do so for tabulated (non-linear) sensors.

3. Library should allow including new types of ADC converters easily. Classes that implement the control of *native* ADCs will be designed in such a way -interface- that it can be replicated for any ADC whose handling is standard. This way, the client application -the one of the user of the library- will not need to be modified even if a new type of ADC is added.

2.2.- Raspberry.

The Raspberry is a computer (microcomputer, nanocomputer, pocket computer, as you like) credit card sized and fully operational (applications compatible with -Microsoft- Office, Internet browsing, multimedia content playback, etc.). And last but not least, the latest model -perfectly suited for the purposes of this blog- costs less than 40€. Add a power supply (the official one costs 12€, there exist great ones for half the price), the µSD card and you have a full-fledged computer. A bargain for the kids and for yourself. Please visit the WEB (“https://www.raspberrypi.org/”) for complete information.

Models from the 3B upwards have bluetooth and wifi too. All of them have, what is of great importance for the purposes of this blog, a 40-pin connector to which you can connect things that you cannot connect to a desktop computer or a conventional laptop.

Raspberry was conceived for students, proposing Python as a programming language, which is already pre-installed in Raspbian.

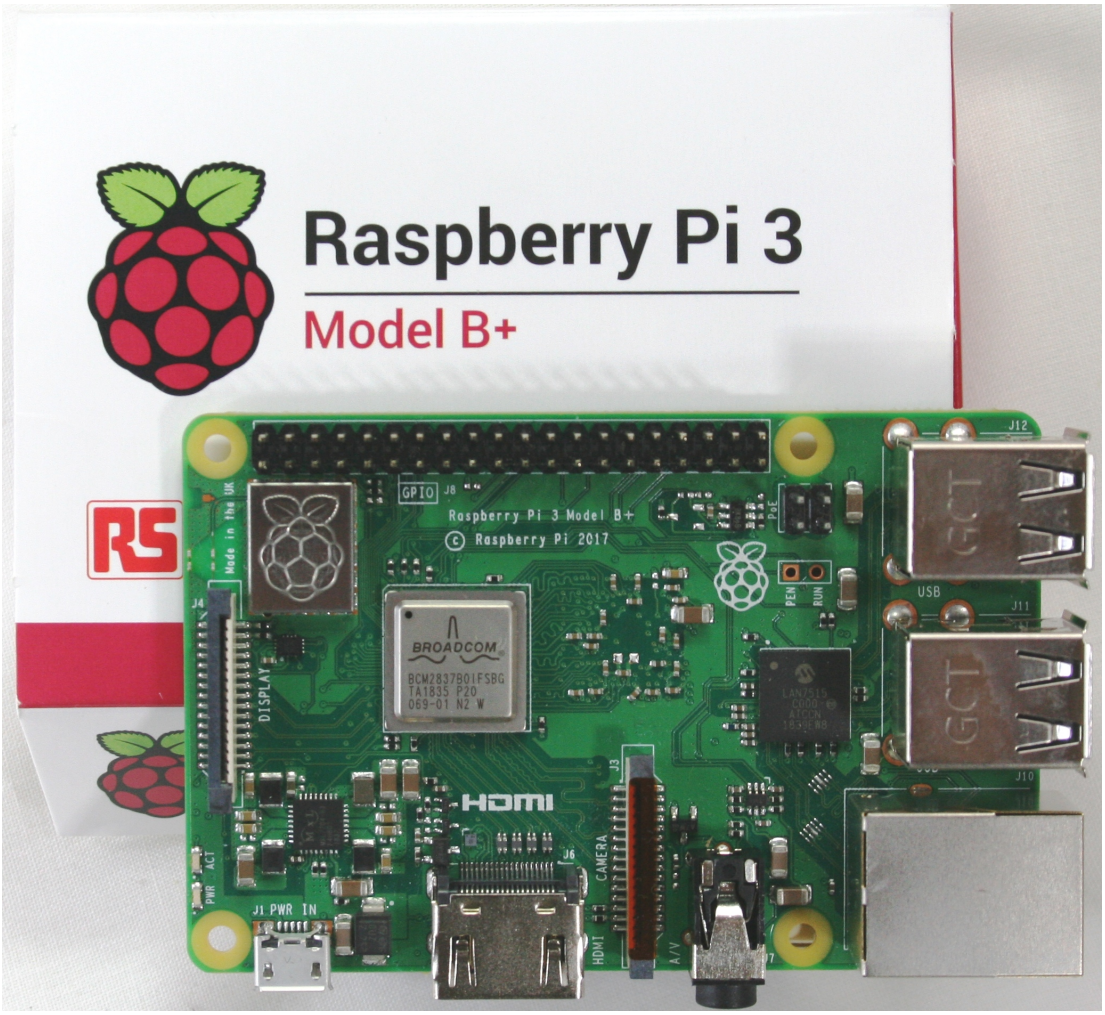


Figure 8. Raspberry Pi 3B +

Praises are over; you can find hundreds in the Internet, along with thousands and thousands of WEBs in which the product is described and all the other stuff that can be done using it. Finally, to state that any model from the "2" -Raspberry Pi 2- is more than enough for this project.

2.3.- ADC. Analog to digital converters.

An ADC converter converts -input- volts into a number you can operate with.

About the terms **"ADC" and "bus"**: These two terms are neither interchangeable nor about the same thing: while the first refers to a "chip" (or electronic board, or module), the second refers to the way two or more chips, either in a small system like the one in this project or in much more complex equipment, communicate among them. In this project -EIDEAnalog- is the case for both the *conventional* ADC chips -the ADS1115- and the small ADC converters that are integrated into the individual devices that are connected, in this case, to the Raspberry by means of a oneWire "bus" (the DS18B20 thermometers). This bus -the oneWire- thus solves the two problems simultaneously: first the sensors give the conversion already done and the bus -this is its mission- transmits it to the Raspberry; as you can see, the whole is a mixture of sensors, digital analog converter (s) and a communication bus that, from the point of view of the library we are conceiving, will be controlled by a single class - "oneWire" -.

The ADS1115 connects to the Raspberry through the i2c "bus" and the Arduino by means of a conventional "serial" bus -RS232 (buses that, on the other hand, allow you to connect many kinds of things to the Raspberry: displays, video cameras, keyboards, printers, ...).

Being the object of this project the design of classes that address all of the aforementioned possibilities in an homogeneous way, (and -being- the access to the bus and the capture of sensor data intertwined from a logical point of view), when referring to the classes that control the bus and ADC chips we will use both terms almost interchangeably (and even the -nonexistent- word "*ADCBus*", which is not rigorous at all).

2.3.1.- oneWire bus.

The oneWire bus was invented by Dallas Semiconductor Corp. and is a protocol (a set of specs for both hardware and software) that eases chips connection (with each other). It has the advantage of just needing one wire for data transmission (two, in fact, the other one is "earth" or "0 volt", as you wish) for all connected devices (in number almost unlimited; of course there are restrictions due to the power supply, lengths, interference, physical space, ...). The transmission speed is not very high 16 kbits / s (about 1600 characters per second) although much more than enough for our project.

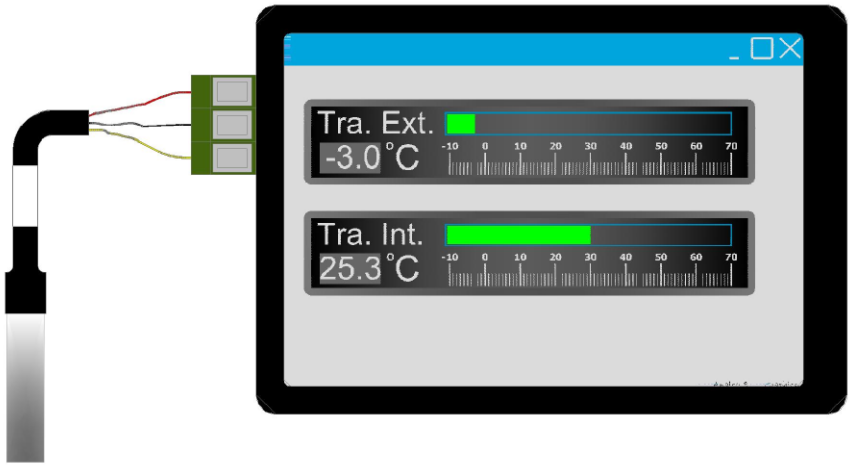


Figure 9. Commercial kit for temperature (Raspberry + oneWire)

The Raspberry acts as a oneWire bus “master” (the name says it all) without the need of any other hardware; It is, therefore, free (see Annex II, “Raspbian Configuration”). Several kinds of devices can be connected to the bus, temperature sensors among them. Along the blog we will use the DS18B20 for the examples; It is a temperature sensor with a microchip that includes an ADC converter and the necessary electronics to communicate -through the bus-, everything conveniently encapsulated. Each of the thermometers (and any other "slave" devices connected to the bus) have a single identification, that consists in a 12-character hexadecimal code. DS18B20 thermometers cost is around two dollars per unit.

Figure 10. Temperature probe DS18B20. Water proof finishing.

The DS18B20 arrangement needs a third cable and a small resistance to supply the sensors (See figure 11).

The standard Raspberry driver -software- to manage the oneWire bus at its lower level works in a quite peculiar way: when starting the Raspberry the driver makes a polling of the bus detecting every device that is connected to it; it creates a folder for each one it finds (See 6.1, “oneWire Class”). Then, to read the sensor, what you have to do is to look inside the text file "w1_slave" on the correspondent folder, this one identified by the sensor hexadecimal code.

This behavior is convenient, since without the need for any specific software you may know the temperature by just opening the file. The problem is that this is quite a slow process, at least in terms of what is usual for ADC devices, even the simple ones: as average, the time it takes to get the reading is about 0.9 seconds per sensor. Despite this, it is enough for our project, since it is about discussing different solutions; this is one of them.

(Actually the probe, including the accompanying electronics, takes a fraction of those 0.9 s. It is the subsequent reading of the file that makes the complete cycle so slow).

The oneWire bus is implemented by using three of the Raspberry Pi GPIO header.

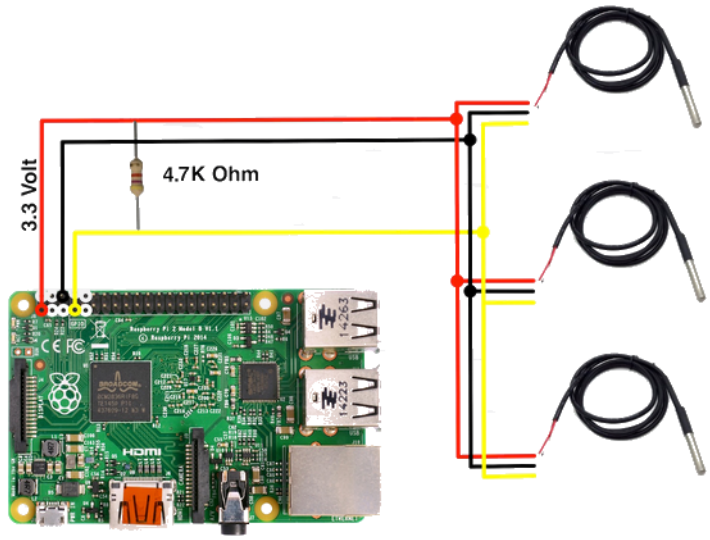


Figure 11. oneWire bus connection..

For the shake of hardware connectivity, class oneWire in this library activates a generic GPIO pin to supply 3.3 V to the oneWire bus. The most usual bus implementation -Figure 11 above- uses some pins that make the connection of the most common 3.5" LCD screens -Figure 9- impossible. A different pin may also be activated for the bus itself (again the *usual* pin is in the LCD screen header area. See annex II, "Raspbian configuration"). Discard all this information in case you are not using a LCD: you may use the conventional layout.

2.3.2.- ADS1115.

The ADS1115 is a 16-bit ADC designed by "Texas Instruments". It has four channels that may give a direct reading (4, therefore -direct readings-) or differential (2, maximum; mixed configurations can also be used -two direct + one differential-; in fact, the mode selection is made on the go, so the choice, except for the connection of the probes which is supposed to be permanent, can be changed even when working). The ADS1115 connects to Raspberry via i2c bus.

The chip has 6 programmable reference voltages (or "gains", to adapt itself to the range of the sensor) and 8 different sampling rates, also programmable (actually the sampling rate -the rate at which it samples- is always the same, a little bit more than 1,000 times per second; what happens is that the chip averages all the samples it has taken in the interval

correspondent to the rate you program: if programmed for 475 samples per second (SPS) it will average 2 samples, if asked for 250 four samples, and so on. -up to 8 SPS-. If you ask for 860 it will result in a single sample; the more samples averaged the more accurate the final result). The ADS1115 may also compare the measured value with a programmable threshold, providing alarms in case the value goes out of the predefined value (option that we will not use in this blog).

Figure 12. ADS1115

Direct use of the chip requires a quite skilled hardware job: the chip is 3 x 3 mm and has 10 pins, so the best is to look for an assembled card that mounts it. The most popular is the *Sparkfun* one, but we have chosen a more than robust card made in China that has an header to connect directly to the Raspberry and screw terminals, so that you do not have to solder anything to use it (the fact that it is chinese is an assumption: beyond the reference "HW-660" that it shows in it has no other identification). It also has a built-in thermistor (NTC) that can be connected to channel one by a jumper -again without soldering anything-, and that will be handy to check the operation of the library with non-linear probes (see 5.2.3 .- "'tabulated' calculation agent"). The HW-660 can be purchased in eBay or Amazon for 12€ (even less).

Figure 13. ADS1115 boards. Sparkfun (left) and HW-660

A couple of paragraphs above the option -need- to program the ADS1115 has been mentioned. The right way is to have a look on the chip datasheet: you can consult and download it from “www.ti.com/lit/ds/symlink/ads1115.pdf”. This is the proper way, we say. Don't worry if it looks cumbersome, in paragraph 4.1., “Class ADS1115” the data sheet is

sufficiently crumbled to justify the elaboration of the corresponding class; It is much better to start there.

A final issue: the Raspberry standard configuration of the i2c bus works at 100 kbits/s; at this speed each sample takes -seems to take- (starting at the trigger order, plus the time it actually takes the chip to convert plus the the time that it takes the reception of the information) much more than what it takes just converting (just over a couple of milliseconds for transmission and about 1.0 millisecond for conversion at 860 SPS). See exercise at the end of chapter 4.1.10 and annex II, “Raspbian configuration”.

2.3.3.- Arduino ProMini.

Arduino is an Italian company that had the great idea of taking some Atmel (A North American company now owned by “Microchip Technology Inc”, also a USA company) microcontrollers and mounting them on boards so that they could be used easily; something quite similar to what was done with the ADS1115 (see previous point) by Sparkfun -and also by the unknown Chinese manufacturer- from the point of view of hardware and, somewhat more difficult to explain -what arduino did-, from the point of view of the software. In summary, arduino managed to have their boards, and much especially the arduino "UNO", a kind of standard in the microelectronic industry. The arduino ProMini that we will use in this blog is a minimal size version of the arduino UNO.

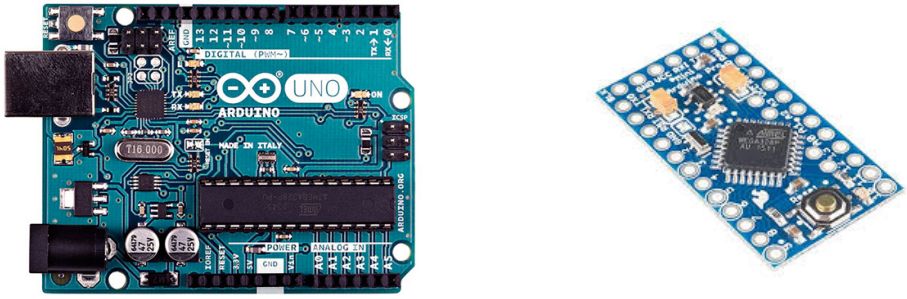


Figure 14. Arduino UNO (left) and Promini.

The Arduino boards -the Atmel microcontroller- are not -only- A/D converters, although they have one built-in (with eight conversion channels). They are, in that they resemble the Raspberry, miniature computers, although the Arduino is focused to the connection with the real world (say what is now known as IoT, although when Arduino was founded it did not exist -the name, the thing itself did-). Let's say that you cannot connect the Arduino to an HDMI display, or directly to a USB memory stick, for example (or not, at least, to basic models).

To use an Arduino it needs its own program, so you have to look for one that is already (programmed) to use the -its- ADC and you need some form of communication also programmed (all Arduinos have implemented the oneWire bus -as in the Raspberry you simply need a driver-, the i2c -the same- and several serial communication channels).

There is a Spanish manufacturer that has what is needed: a ProMini mounted on a board that, in turn, is prepared to be mounted directly on the Raspberry. Furthermore, although it has its own programming -see 4.3, “Arduino class”; this company uses it as part of a “data logger” - it behaves as intended: it converts the 8 analog channels (10 bits resolution) and sends the information packed by the serial channel on demand. We will use it as another option in the library.

Figure 15. Promini Raspberry hat.

2.4.- Programming language. Python. IDE.

Python's implementation of everything related to OOP is up to date. Add -to Python, no to OOP programming techniques, this is quite a different matter; see next paragraph- a more than generous learning curve and we have the perfect tool.

(Well, may be it is not perfect: Python is an interpreted -not compiled- language, so it is relatively slow. "Slow" means that, for example, a Python program that calculates the square of the first 1,000,000 numbers takes, on a veteran desktop computer, 0.23 seconds. That is 0.23 microseconds on average per calculation. There will be plenty of people who will cross out this ridiculous figure when compared to the speed of an iPad. By the way, on a Raspberry Pi 2 it takes 1.2 microseconds: it is five times slower!; on a Raspberry 3 B + it takes around 1.0 microseconds).

In what concerns the development environment (the IDE; *Integrated Development Environment*) to use there are discrepancies. The one that includes -it used to include, keep reading- by default the language, that is, what appears -it used to appear- when you select the option "Python 3" (or "2") in "Programming" -or a file " .py "- on the Raspberry is -it used to be- a good one. In the latest versions -distributions- of Raspbian ("Raspbian Buster" at the end of 2019) this has been replaced -damn!- by a certain "Thonny Python IDE". This environment is very likely to improve the old one, but when one gets older becomes reluctant to change (and much more if one is obliged to).

Should you are making your *debut* -following this blog- with Python, try that *Thonny:* sure it works well. If you are disobedient and prefer the old environment you have to mess around a bit: see Annex II, “Raspbian Configuration” to install the old IDLE.

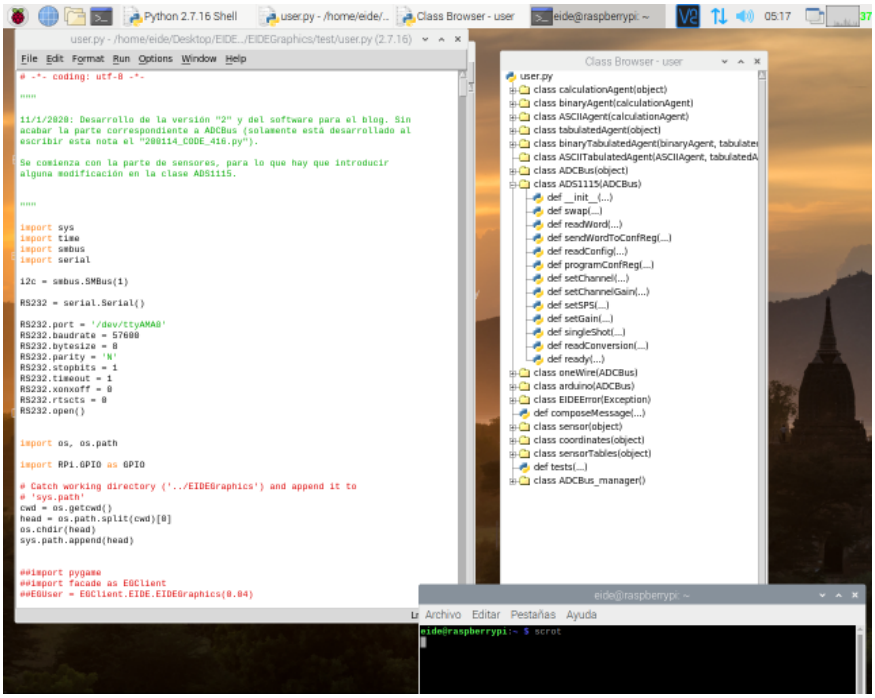


Figure 16. (Old) Python IDLE.

2.5.- Design using OOP.

2.5.1.- Abstract.

OOP is not an easy matter.

Procedural programming (for the sake of simplicity, the one used without resorting to OOP) has a generous learning curve: one knows the "for- (next)", the "if- (then) – else" (what is in parentheses is because it is not used in Python) and having an idea of what a *routine* is (*function* in Python) you can write programs to solve quite complex issues; there exist code written with these resources and a little else that would make the Apollo X computer programmers green with envy.

Yet, this type of programming has drawbacks, being the most remarkable one its poor *reusability*.

It is all about the chances of reusing the code -the whole or a part- previously written to be used for a certain environment into another. The concern was for a long time (let's say from the time first programs were written -in the 1950s - till the beginning of the 1990s) a very controversial one -and poorly solved. There was no way for the “reuse” to be painless. To a greater or lesser extent, the copy-paste did not work: best case -and always based on military discipline- routines could be reused almost without modification, but the certainty of not having maintenance problems afterwards was minimal. Any improvement - this is another matter, that of *improvements*, which deserves another blog- in the original routine always required the user -customer- some effort to readapt the code. Most occasions, the *built-in* routine ended up being completely rewritten and the intended time saving turned into a loss: there are many examples; anyone who has developed software for someone else knows perfectly well what we are talking about.

The OOP technique solves this and other problems related to software reusability and maintenance. The code with the slightest probability of being reused is implemented by *classes* whose subsequent *instantiation* enables the code use in an opaque way (encapsulation of code and attributes). The classes, in turn become part of libraries that help to solve one or more problems in a certain domain -area- (this last statement being a simplification).

2.5.2.- Example.

You are taking your first steps in Python. You want to program a basic calculator: by entering two numbers and an operation ("+", "-", "*" or "/") the operation is performed and the result is displayed.

Leave aside how you coded it (the calculation once the three elements have been entered into the computer). The point is that the basic -*native*- tools of Python to achieve a certain interactivity are really poor: the “input ()” function is quite primitive: your brand new calculator deserves something *à la mode*. Something like this:

Figure 17. Using wxPython .

Here is when the OOP becomes handy. If there is a library that covers this it will suffice (assuming that everything they say about libraries and OOP is true -it is), by using it. You mess around the Internet, social networks, ask someone you know he/she masters this stuff to finally get to the conclusion that the right thing is using "wxPython". (By the way, there is another widely used GUI library, "TkInter", which was the first for Python. There is a certain rivalry between the two -among their followers-: the authors of the blog opt for wxPython).

wxPython has a class *TextCtrl* that instantiated gives raise to an object which is the small window where the number "A" is to be written. The corresponding code would look something like this:

```
A = wx.TextCtrl(self.panel, -1, "", style = wx.PROCESS_ENTER)
```

(Do not run away: we have already told you that the OOP is not easy; by now you just want to make an idea). The other two, for "B" and the result would be:

```
B = wx.TextCtrl(self.panel, -1, "", style = wx.PROCESS_ENTER)
result = wx.TextCtrl(self.panel, -1, "", style = wx.TE_READONLY)
```

Let's say that in a certain *“self.panel”* (that is inside the window where the whole process will take place) there will be three boxes; in two of them you can write (“style = wx.PROCESS_ENTER”) and in another just look (“style = wx.TE_READONLY”; we will leave the “-1” and “” for a better day).

Now we need the labels ("A", "B" and "result"; the above is just to enter and see numbers). You have to write something like this:

```
ALabel = wx.StaticText(self.panel , -1, “A”)
BLabel = wx.StaticText(self.panel , -1, “B”)
ResultLabel = wx.StaticText(self.panel , -1, “Result”)
```

that shows up the labels (we have instantiated the *StaticText* class three times: three labels).

And so on: four instances of *wx.Button* (operations), tell where to put everything on the panel (alternatively it can be done automatically with another object of class *wx.GridBagSizer:* things get complicated), instantiate the *panel* where the whole drama will

take place (*self.panel = wx.Panel (self)*) into the window, which, of course, is an object of another class (¿*Calculator*, perhaps?) which in turn inherits from the *wxFrame* class.

Take a breath, the good new is that you do not have to program anything -anything else than the above- to have a window with a panel inside that, in turn, has buttons, labels, boxes (in which, by the way, you will have the functions of a word processor: insert, delete, etc. You can also move around the entire window with the mouse, write inside the box you have selected, press a button, etcetera). And all interactive: if you have any previous notion of programming you have a good idea of what the effort to get such an interface is. And surely you are starting to be intrigued by the above; else calm, it is a toll (the abstruse of the subject) that is worth it.

2.5.3.- OOP. Conclusions.

There are -the one you are right now reading too- a lot of blogs and WEB sites where you will find a lot of nonsense about the painlessness of the OOP: especially touching are those that introduce the subject with the class *mammal* whose subclasses are *cat* and *dog*, which, in turn, participate from the classes *tail* and *color*; its objects, "Boby" and "Sanson", move the tail (or not). Those of cars and drivers normally have also not waste.

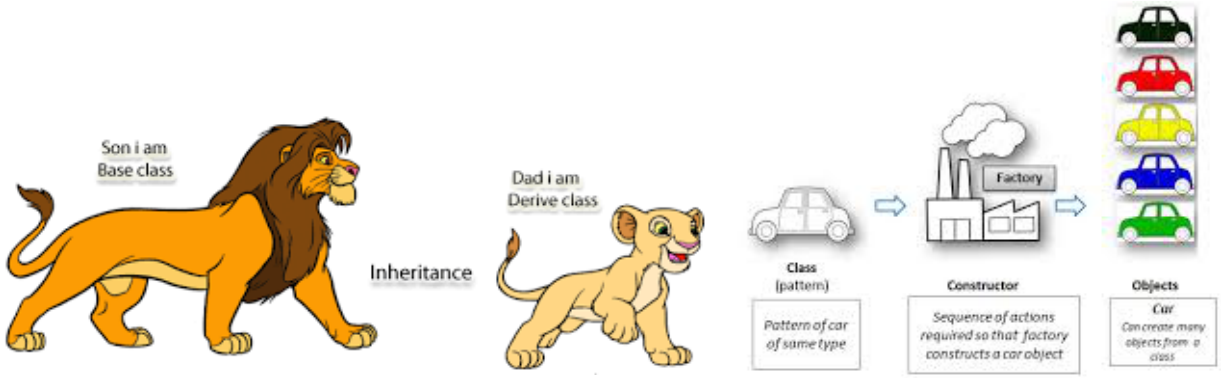


Figure 18. OOP made easy: mammals y cars.

(The point is not that this simple approach they propose being essentially false: it is not. The mistake is trying to explain a zoo with *mammal, dog, cat, Boby* and *Samson* or a Formula 1 race with just *car* and *driver*. There is much -much- more stuff than just that).

The OOP is not a one lesson subject -nor of a quarter, probably - so take it easy. This blog is intended to help you take the second steps (see 1.2.4., “[Basic knowledge to follow the blog. Prerequisites] Python. OOP” for the first ones). By the end of the blog you will be familiar to the following basics of OOP:

- Classes and objects.
  - Class instantiation.

    - Class attributes. Methods. Object attributes.
    - Inheritance. Multiple inheritance.
    - Object composition.
    - Project documentation. UML Notation (very basic notions).
- Libraries.
    - Use of. Library client.

3.- Library pattern. First approach.

There is no magic bullet for what we are going to do next paragraphs. Yet, it is the most important part of the project (at least for those who follow it to learn a bit on applications building using OOP).

Let's go for it. The first issue to solve is to fragment the complete task into subtasks that are easier to face than the whole one. We are going to do it using the "object oriented analysis" (so that OOP can be used later).

3.1.- Noun/verb.

We have to delineate the future classes and what *their* objects will be able to do (methods). As a first approach, the "noun/verb" technique is to be used. It helps clearing up the problem.

It is about:

1. Writing down the project spec (we've already done this).

2. Separate names from verbs: names would be the classes and verbs the methods.

The stated spec is:

- The goal is developing a library in Python 3 to use a Raspberry as a data acquisition system (**conversion** of analog to digital variables). The library must allow to **select** the ADC from three *native* alternatives (whether alternatively or simultaneously): 1/ the Raspberry Pi native oneWire bus, 2/ A "chinese" board -there is no explicit manufacturer- that mounts an ADS1115, and 3/ an arduino ProMini. The measurements taken by the ADC will be **converted** to a readable format.

- The library has to help connecting several sensor alternatives: those designed for the oneWire bus (see 2.3.1, "oneWire bus") and *generic* sensors for temperature, pressure, distance, sound, etc. -those ones that **convert** the measured parameter to volts- in the case of the other two options (ADS1115 and arduino). While to add a new ADC to the library involves writing a new class, new sensor types would be added without modifying the library code. It should even be possible to do so for tabulated (non-linear) sensors.

- Library should allow including new types of ADC converters easily. Classes that implement the control of *native* ADCs will be designed in such a way -interface- that it can be replicated for any ADC whose handling is standard. This way, the client application -the one of the user of the library- will not need to be modified even if a new type of ADC is added.

Some nouns (class candidates) stand out:

"Analog variable", "digital variable", "bus", "oneWire", "ADS1115", "arduino", "sensor", "ADC".

Verbs (methods candidates): “convert” (transform) is, undoubtedly, the star.

(Spoken language is not exact -is it computing?-. Some terms may be taken as nouns or verbs: conversion is a noun, but it derives directly from *to convert*; let us be a little ambiguous for the moment).

This “noun/verb” method is not quite scientific; we have to debug the result. Let's go to the next step: you have to sharp your mind because the next matter is a bit more elusive one.

3.2.- Classes as services suppliers.

It is about analyzing the -eventual- classes from the point of view of what they -the objects you get by instantiating them- can offer to the client (user). Classes have to help solving the problem, not complicating it.

Look at it this way: the programming language, Python this case, has what it must have: variables, operators, flow control commands, error handling, class management, etc. We cannot ask the language is to have commands for the specific problem: Python has not an instruction "convertChannel" to manage an ADC or "tableLookup" to get a value from a given table.

We are addressing one of the core aspects of the OOP: class methods have to supplement the programming language so that to the client is -almost- as easy to use a Python primitive instruction -”print”, for example- as a *new* one -*convertChannel,* for example-.

In addition, if the classes are conceived in such a way that they are valid for any similar environment -or maybe not so similar-, we are getting closer to the correct way of facing the problem. We get into matter below.

3.2.1.- Bus classes. Sensor class. Conversion.

Classes managing the *ADCBus* devices (oneWire, ADS1115, arduino; -see note at 4.3-) are quite a direct ones. They contain -encapsulate- what is essential for the *ADCBus* to work and little else; they touch the bare metal -the hardware. Taking a preliminary look at points 4 and 6, you can see that a good part of them (especially the ADS1115, which we will use as a standard throughout the blog) are almost the literal transcription of the datasheets. We will also make an effort to keep them -the *ADCBus* classes- as independent as possible (*loosely coupled*) from the rest of the library.

Much subtler, however, is the issue of sensors and the one of the human readable value calculation (They appear as candidates to a class -sensor- and a method -calculation-; the readout itself will end up being a sensor attribute). As we will see below, the classes related to these topics are much more abstract than the previous -*ADCBus*- ones.

Except for the oneWire bus, which delivers the sensor reading in an almost a readable format (it gives it directly in ASCII, in centigrade degrees millis), the conventional format for an ADC to give the reading is binary; so it is in the case of the other two ADCs that we use in the blog: in *two's complement* the ADS1115 and *direct binary* in the case of the Arduino. And we need to transform them into human readable format (for example, 00010000 = 32 for *direct binary* -Arduino- and 10000001 = - 127 in two's complement -ADS1115-).

3.2.2.- *readout* method.

To convert the value the ADC gives (remember, normally a binary one), and whoever does it and wherever it is done, the following is a must:

1. The format -*two's complement* or *binary*- is it. Put it another way, what ADC is it coming from.

2. The sensor parameters. The sensor gives a voltage -which is what the ADC converts: a voltage- proportional to what it is measuring: centigrade degrees, for example. It is highly unlikely that the ratio in between the parameter and the voltage is a "1 to 1" one, so the sensor is characterized by its *gain,* which is the ratio in between the units it measures and the volts it gives. Along the blog the LM35 is widely used as a reference sensor: it has a gain of 100 (see 5.1, "Sensors").

   Sometimes (keep seeing 5.1, "Sensors") the sensor has an offset in between its "0" (Volts) and the "0" of what it is measuring: we will call that value *zero.*

   There are sensors whose reading (or whatever they are converting) does not have a linear dependence such as that described in the previous paragraphs (keep seeing 5.1, "Sensors"). This case the conversion –in between what the ADC gives and what we want to see - is *tabulated*.

3. Finally there is a parameter which depends both on the sensor and the *ADCBus*: the *reference voltage*. It is not about the volts that the sensor withstands; rather it is about how much -voltage- it outputs. This is another element when computing the readout, so it becomes part of the repertoire of things that you need to convert the ADC output to the human readable format. This parameter, along with the gain, the zero and the -eventual- table, will be part of the sensor "library" (see a little later).

Summarizing, to calculate the readout you need the bus data (the data format) and the sensor info (gain, zero, reference voltage and a table -if applicable-); the question is: where -method- is the calculation made?; Which class will hold that method?.

Let's think a bit: it looks like we need a class "sensor" which, at least, encompasses the above list for each one of the sensors connected to the system: 1) assuming that the information for each *type* of sensor -LM35- is centralized -it is-, one sensor object attribute has to be the type -'LM35 ', or whatever it is- and 2) which bus is it connected to -ADS1115, for example-. (This could be done the way around: the *ADCBus* object has the information of the sensors it is converting; this case it would be -the object of the bus class- responsible for the calculation. It would be a perfectly valid alternative, although, as will be seen in due course, less flexible: you cannot instantiate a sensor - a *real* sensor,

not a *type* of sensor - until you know which bus it is connected to. It is possible, instead, to instantiate a bus without specifying which sensors it has connected: it may not have any).

So the sensor class has to be able of calculating the readout. As a first approach the class will have a method -*readout*- that computes, passed as an argument what the bus gives (binary; ASCII) the human readable value. To accomplish this task all the values listed above should have been passed as arguments at the time of -sensor- instantiation. Let's anticipate, it is a little early for this, that the instantiation will be something like this:

```
extTemperature = sensor('LM35', myBus, 2)
```

(We will come back to the subject in 5.4, "Class sensor. Code. Instantiation"; let's advance that *myBus* is the bus to which the sensor is connected and "2" the *ADCBus* channel).

Back to the analysis:

A landmark on an OOP analysis -and its subsequent implementation- is the absence of *do case* (or *switch*) clauses. These are at the heart of procedural programming: as the computing of the readout depends on many parameters (remember: type of sensor; *ADCBus* it is connected to), a procedural programming will select the computing *routine* by *switching* (or *do casing*) by the sensor type and then by the *ADCBus* type.

For tabulated sensors a table must be added to the above: Next paragraph is copied from 5.3.5. "Multiple inheritance. Classes 'binaryTabulatedAgent' and 'ASCIITabulatedAgent'".

"In a project like the one we are dealing with -and almost always- what happens if there is a tabulated sensor (the NTC of the Chinese board made with the ADS1115 for instance, see figure 19), is that it is connected to a channel of the *ADCBus*, the latter delivering a reading in a certain format (binary, ASCII, ...) that must be converted to decimal -like that of any other sensor-, then pass the computed value to the "tabulatedAgent". This one looks up into the table to calculate the final readout".

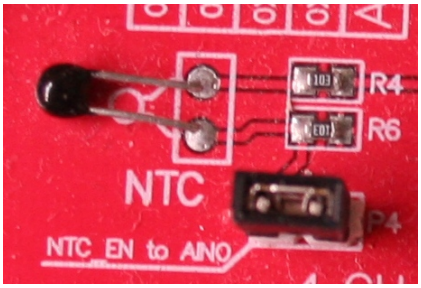


Figure 19. NTC

Should a tabulated sensor be present, the *ADCBus* value has to be converted as any other sensor: first the reading is converted to decimal and then the final value is extracted from the table.

Again, with procedural programming, this would lead to a "do case" or an "if-then-else" (which should not appear in the OOP either).

(It is beyond this blog scope going deeper into this matter. Let's say that the reason for what non-existence of "switch" clauses is a sign of good OOP analysis is because it is a symptom of little coupling in between classes, a desirable characteristic connected to software reusability).

3.2.3.- *Calculation agent* classes.

The issue of computing the final readout will be solved by devising a family of classes "calculation agents" (we will use the *inheritance* technique to derive some of them from the others) that implement the different calculation algorithms depending on the format (binary or ASCII), the sensor data and the fact of having or not -the sensor- an associated table (See 5.3, “Algorithms to computing readouts. Agent classes”).

Nevertheless, the point is not how this family of classes is created - inheritance-, but rather the technique that we will use to link the one needed to each concrete sensor: is the *sensor* class object what in turn will instantiate the needed class from the family of *calculation agents*: this way we will get the sensor class *readout* method, which is the one in charge of computing the readout, to do it by delegating to the calculation agent object. The code will be exactly the same whatever are the sensor and bus; a mere line:

```
return self.agent.convert(value)
```

(as will be seen in due course).

So we have an initial library pattern:

1. Classes for each one of the ADCbuses that can be connected to the Raspberry: oneWire, ADS1115 and Arduino. As will be seen in due course, an abstract class -*ADCBus*- will also be created, from which these three inherit; It is -the abstract class- half a matter of style and half of utility: there are a couple of basic (essential) methods that all bus classes share.

2. A class *sensor*, whose main purpose is to act as a data warehouse, will enclose a *readout* method that, by means of a previously -and conveniently- instantiated calculation agent class, computes the final human readable value. At the moment of instantiation the objects of this class perform another key task by bringing together all the objects that will collaborate with each other (See 5.4.- “Class sensor code. Instantiating”).
3. A *calculation agent* family in charge of computing readouts.

Less than a dozen classes: the stuff to solve is not complicated.

3.3.- Library pattern. Classes.

Using an OMT (https://en.wikipedia.org/wiki/Object-modeling_technique) based notation the figure below shows the intended pattern.

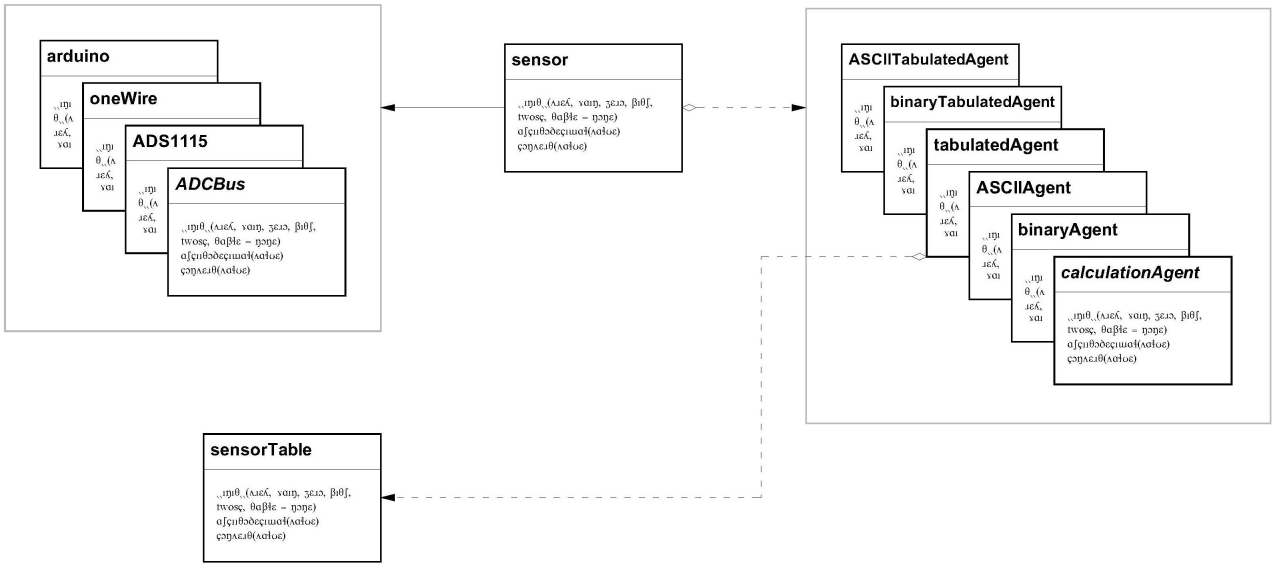


Figure 3-2. Library EIDEAnalog. Preliminary class pattern.

(Notation is derived from the excellent book “Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides”, that in turn is derived from the OMT standard).

4.- Class ADS1115

Let us summarize a bit before designing the classes to control the devices to which the sensors are connected: "ADS1115", "oneWire" and "Arduino".

Our goal is to design a Python library (those classes being a part of) by which the use of a Raspberry to acquire data will be eased.

The design (the classes interface: the "methods" to interact to the class objects) that directly control the different hardware options -ADC- has to be done in such a way that changing the analog to digital converter may be achieved with a minimum of modifications in the *client* code: just the instantiation of the bus. These classes have the purpose of just communicating with the bus (See below the -proposed- meaning of these terms - "bus" and "ADC").

Figure 20. ADS1115 board.

About the terms **"ADC" and "bus"**: These two terms are neither interchangeable nor about the same thing: the former refers to a "chip" (or electronic board, or module), while the latter refers to the way two or more chips, either in a small system like the one in this project or in much more complex equipment, communicate among them. In this project -EIDEAnalog- is the case for both the *conventional* ADC chips -the ADS1115- and the small ADC converters that are integrated into the individual devices that are connected to the Raspberry by means of a oneWire "bus" (the DS18B20 thermometers). This bus -the oneWire- thus solves the two concerns at the same time: first the sensors give the conversion already done and the bus transmits it to the Raspberry; as you can see, the whole is a mixture of sensors, digital analog converter(s) and a communication bus that, from the point of view of the library we are conceiving, will be controlled by a single class - "oneWire" -.

The ADS1115 connects to the Raspberry through the i2c "bus" and the Arduino by means of a conventional "serial" bus -RS232 (buses that, on the other hand, allow you to connect many kinds of things to the Raspberry: displays, video cameras, keyboards, printers, ...).

Being the object of this project the design of classes that address all of the aforementioned possibilities in an homogeneous way, (and -being- the access to the bus and the capture of sensor data intertwined from the logical point of view), when referring to the classes that control the bus and ADC chips we will use both terms almost interchangeably (and even the -nonexistent- word "*ADCBus*", which is not rigorous at all).

ADCBus control classes -their objects- will not be *aware* of what sensors are connected to them; so we have to think on other classes whose objects handle the sensor information -type, gain, etc- and that convert the data the ADC delivers into human readable values that will be logged, shown on the screen or whatever you intend to do with them. Thus, the ADC control classes end their function by giving the raw value, which normally is a binary value (either direct or in two's complement), although oneWire bus DS18B20 thermometers deliver almost the human readable readout, as we will see in due course.

Depending upon the real system we are working on, *ADCBus* classes will be instantiated as many times as necessary. Say several ADS1115s are connected to the Raspberry (four is the maximum), as many instances of the corresponding class will control them simultaneously and separately. They could even be managed together with any oneWire and Arduino *buses* that may coexist (provided the hardware is suitable for that purpose).

For the sake of clarity, we divide the class analysis and development in two stages, first one goal being to just convert the default channel (the first one) with the default gain, mode (*single shot:* the ADC has to be triggered each time it has to convert) and samples per second -SPS-. We will call this “the *minimal* version”; this allow us to read a sensor, which is the basic purpose of the project. Next, we will add the necessary methods to refine the control of the ADS1115 so that we can change the gain, the SPS and activate the conversion for all the four channels.

Two important guides to do the analysis:

1. *ADCBus* classes *interface* will be the same for all of them (ADS1115, oneWire and arduino).

2. Methods will include from the very beginning the required coding according to the specs: we will flee from *provisional* solutions.

May be that applying those criteria leads to an apparent complication in the class structure; it will be in our own reward. If this circumstance occurs, it will be duly commented.

4.1.- Analysis.

Let us go to it: to reading the first channel of an ADS115 one has to proceed as follows:

1. Send the “start conversion” command.
2. Wait till the ADS1115 finishes converting.
3. Ask for the -analog to digital- conversion (value).

Each one of these three steps correspond to three different class methods (*singleShot*, *ready* and *readConversion*). Now we have for free a class interface sketch: these methods.

As we will see those methods will demand other methods, more specialized, that we will delineate on the fly.

4.1.1.- Start conversion command.

In order for the ADS1115 to convert, a “1” must be written to position 15 of its *configuration register*. And this must be done by leaving the other 15 bits untouched so, taking into account that to change any bit on the configuration register you have to write all the 16 bits (the trigger -conversion start- and the other 15), you must first read its contents -the one of the configuration register-, change just this bit and send the result back to the ADS1115.

Doing things so, reading the configuration register, changing just what you need to and sending it back, is the usual way to program the ADS1115. It looks like we are going to need two “specialized” methods, one that reads the configuration register and another that writes it (sends information -commands- to it).

On the other hand, the ADS1115 has another register -the *conversion register*- that holds the last analog to digital conversion -the binary value. The code to read it is almost identical to the one used to read the configuration register; is not good having a similar code twice -repeated-. We devise a single method that does it -reading both registers, one at a time-: we have to pass which register number -or whatever- as an argument to this new method.

We will call those methods "*readWord (register)*" -the one that reads both registers: which one is passed as an argument *(register)*- and "*sendWordToConfReg*" -the one that writes the configuration register-.

---

i2c bus usage: In this blog we cannot submerge into the deepest details on how the i2c bus works. i2c is a protocol to manage the communication in between chips by two wires; Python has a "*smbus*" library that provides the instructions -methods- necessary to handle it. As the "smbus" protocol is actually a *low speed* subset of the complete "i2c" protocol, it is amazing that the library is not so called -"i2c"- instead of "smbus". To finish messing it up -or not-, this blog calls "i2c" the instance of "smbus" that we use to interact with the ADS1115: you may excuse us -or not-.

The library "smbus" command -the *object method*, to be precise- to read a register is “*read_word_data*”, that *returns* a "word" (16 bits) in 2 bytes. Taking the datasheet as a reference (on how to order these two bytes), the command returns the two bytes swapped: the second one on the first place and the first one on the second place. The write command -”*write_word_data*”- swaps them again on sending them back, so we could forget about it, locate the key bit -or bits- in the -swapped- word, change it and send the word back: it works. The issue is that this way of working would be confusing when working on the configuration register (you would have to swap for yourself the whole datasheet to make things clear). It is much better having another utility method to which we pass a “word” and returns it to us swapped: we will call it “*swap*” (of course).

A final issue: it is quite convenient the class interface -the methods the *client* is going to use- being as *friendly* as possible. A client code that has a *readWord(0)* line to read the configuration register and *readWord(1)* to read the A/D conversion will work nicely, but its reading -and probably its usage- will be much more confusing than if we "*wrap*" these two options with two separate methods: *readConfig* and *readConversion*. A few more lines of code and an priceless advantage to not make mistakes and read the code more easily.

Hence, we devise seven methods for our class ADS1115:

Class: ADS1115	
Methods **readConversion(self)** **ready(self)** **singleShot(self)** swap(self, word) readWord(self, register) readConfig(self) sendWordToConfig(self, word)	

Table I. Class ADS1115 *(minimal*). Interface methods in bold characters.

Explanations and caveats preceding the table should be enough as to understand the content for each one of the methods (it follows). We will remark some aspects as we present them so that the ideas get completely clear (many code lines are not fully explained, except for the case that they have any detail that goes beyond the basic knowledge of Python):

swap(self, word)

```
def swap(self, word):
    """ Swaps the two bytes of a word """
    valor = ((word&0xff00)>>8) | ((word&0x00ff)<<8)
    return valor
```

- Returns a word with its two halves (bytes) swapped.
- The *self* pseudo-parameter is a Python syntactic requirement for all the class methods. It is also used as a prefix whenever a class references its own attributes and methods (for example: *self.swap, self.addr*; see below).

readWord(self, register)

```
def readWord(self, register):
    """Read word from ADS1115 register """
    valor = i2c.read_word_data(self.addr, register)
    valor = self.swap(valor)
    return valor
```

- Returns the contents of the register whose number is passed as an argument *(readWord(self, **register**)).* The *word* is *returned* conveniently *swapped (valor = self.swap(valor)).*
- The code gets the info by means of the "*read_word_data*" method of the *smbus* object "i2c". We will come back on the subject later.
- *self.addr* is the i2c bus address for the -physical- ADS115 that the instance of the ADS1115 class is managing (you can connect up to four ADS1115 simultaneously to a i2c bus). *self.addr* value is loaded at the instantiation time (see below).

readConfig(self)

```
def readConfig(self):
    """ Read config reg. """
    r = self.readWord(ADS1115.confReg)
    return r
```

- This method uses the precedent one -*readWord*- to read the configuration register contents and return it.
- The register is addressed by the contents of the class attribute *ADS1115.confReg.* It is a matter of style, using attributes instead of raw values ("0", for the configuration register; the conversion register is "1") that improves code readability and maintenance.

sendWordToConfig(self, word)

```
def sendWordToConfReg(self, word):
    """ Send a word to the ADS1115 configuration register """
    # swap byte order
    word = self.swap(word)
    i2c.write_word_data(self.addr, ADS1115.confReg, word)
```

- Sends (*write_word_data*) the *word* passed as argument to the *configuration register (ADS1115.confReg)* of the ADS1115 in *self.addr* .

singleShot(self)

```
def singleShot(self):
    """ Start an analog to digital conversion """
    #Read config register
    valor = self.readConfig()
    # Set "single shot" bit
    valor = valor | ADS1115.startConv
    self.sendWordToConfReg(valor)
```

- See paragraphs preceding the table above.

The preceding five code snippets make up almost the *minimal version* entire core of the class "ADS1115". Remember that it is about being able to read the ADS1115 default channel (“0”). It is worth discussing it a bit: it is the first Python code implementation of the blog.

- Let us have a look on the *singleShot* method (which is the intended goal of this block of code; it is about telling the ADS1115 to convert to later read the conversion). They are nothing else than three lines of code: *value = self.readConfig (), value = value | ADS1115.startConv and self.sendWordToConfReg (value)*. Only *value = value | ADS1115.startConv* is native to Python; the other two are calls to methods of the class itself. This is one of the OOP cornerstones: it is about generating a language of your own so that the code becomes a chain of calls in between methods of the different objects (or from the object methods to the object methods themselves). This is a recurrent idea that has already been discussed (see 3.2, “Classes as service providers.”) and that will be widely analyzed in the remaining of the blog.

- The class (and its *interface*) is designed without a particular specification. In our case the specification is quite generic: it is about reading analog variables by means of a Raspberry; one of the devices used is the ADS1115. It is not specified whether the converted values are to be displayed, to be dumped to a WEB page or if they will lead to alarm warnings. We don't even know if they are going to be transformed to a “visible” readout (23.2 centigrade degrees; 7.0 kg/cm^2). Yet, all the work we have done will be used exactly as we have completed it. This is another of the OOP pillars: the classes -their objects- offer the client an interface that is an extension of the programming language. From the very moment you instantiate the

class, *singleShot()* becomes a (new) command. It does not matter if the class changes -the class code-: it is enough by keeping the interface so that the client that uses it is oblivious to these modifications; your code -the *client* application program- will continue working exactly the same, as long as the class does. We will also come back to this idea once and again.

- The client has access to *all* the object methods: *readWord(0)* -a non *public* (interface) method- can be used instead of *readConfig()* -a *public* one-. It is this, the issue of making certain class methods available to the client or not, more a methodology issue than a practical one; by coding in Python it is not possible to deny direct access to the class methods and attributes to the *client* (should he/she endeavors to). In this project the class table (summary) says which methods are considered “public” (which are recommended to be used as the *interface*) and which ones are not.

- A classical programming style avoids explicit constants in the code (*ADS1115.confReg* instead of "0"; *ADS1115.startConv* instead of "0x80"). This leads to adding a few lines of code into the class *heading* (see later what is the *heading*: “*Class Attributes*” and method “*__init __ ()*”). The effort is payed back by an improved readability and ease of code maintenance.

4.1.2.- End of conversion.

The hard work done up to here (in order to have a *conversion start* command: *singleShot* plus other four utility methods) will begin to pay off right now. The method to check whether the ADS1115 has finished the conversion is:

```
def ready(self):
    """ Return true if last conversion finished """
    valor = self.readConfig()
    ready = valor & ADS1115.done
    return ready
```

that returns “1” (*True*) if conversion is over or “0” (*False*) otherwise. The code needs no further explanation (at least no more than that given in previous paragraph 4.1.1, “Start conversion command”).

4.1.3.- Conversion reading.

Once again the work done up to now eases code writing:

```
def readConversion(self):
    """ Returns the conversion register contents """
    valor = self.readWord(ADS1115.convReg)
    return valor
```

that returns the converted value in two’s complement format.

We are almost done: the *minimal version* code just needs the above mentioned *heading.* Here you have.

4.1.4.- Class attributes.

A Python distinguishing feature are the *class attributes.* All the class instances -objects- share these attributes.

We will use the class attributes to store the ADS1115 working parameters, that is, all the chip parameters needed for its use (all instances of the class will use them). For the moment we have detected the following ones, whose first appearance in the code is:

```
class ADS1115():
    confReg = 0b00000001
    convReg = 0b00000000
    startConv    = 0b1000000000000000
    done         = 0b1000000000000000
```

Those attributes are referenced as *ADS1115.XXXX* along the code (*ADS1115.confReg*). They can be referenced as object attributes also (*myBus.confReg*).

4.1.5.- __init__() method. Instance attributes.

Each ADS1115 class instance has attributes of its own, that are not shared with the other instances (see previous paragraph). For the moment we have used one, *self.addr*, which is the address of each ADS1115 connected to the i2c bus. Its value is initialized in the *__init__* method which is implemented like this:

```
    def __init__(self, address,  ordinal=0, name=None):
        # ADS bus and address (i2c)
        self.addr = address
        self.ordinal = ordinal
        self.name = name
```

The *init* method is the so-called *constructor* in the OOP jargon. Aside from its ugly look, it is just a method that is run every time a class is instantiated. Sometimes (see 5.4, “Sensor class code. Instantiation”) the *init* method has its own algorithms so that the mere instantiation endows the object with some sophistication to ease later tasks.

It is not the ADS1115 class case: the classes for the use of buses are simple and implement just what is necessary for their functioning. For the ADS1115 we will use it, by the moment -remember that we are writing a *minimal* version-, to initialize the address of the board (see figure 22) and to numbering the instance -*ordinal*- and give it a name -name-. These are assigned by default "0" and "None" in case the client indicates nothing.

In the case of the Chinese board, the address is selected by means of a jumper on the board to choose among four different addresses; ADS1115 instantiation has to be done with the selected address (0x48 in the picture), so that the instance is assigned to the board that has that address. The instantiation is:

```
myBus = ADS1115(0x48)
```

An object called “myBus” is generated. It is an object of the class ADS1115 with attribute *self.addr* 0x48; It corresponds to a *physical* board that contains an ADS1115 and that has the address 0x48 selected.

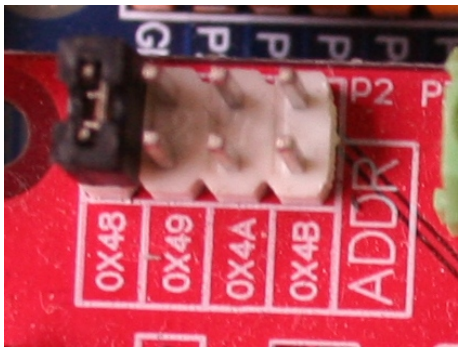


Figure 22. ADS1115 with address 0x48.

4.1.6.- Class ADS1115 minimal version code.

We have got the complete *minimal version* code:

```
class ADS1115():

    # Registers
    confReg = 0b00000001
    convReg = 0b00000000

    # Start conversion
    startConv    = 0b1000000000000000
    # “Done” bit
    done         = 0b1000000000000000

    def __init__(self, address, ordinal=0, name=None):
        # ADS bus and address (i2c)
        self.addr = address
        self.ordinal = ordinal
        self.name = name

    def swap(self, word):
        """ Swaps the two bytes of a word """
        valor = ((word&0xff00)>>8) | ((word&0x00ff)<<8)
        return valor

    def readWord(self, register):
        """Read word from ADS1115 register """
        valor = i2c.read_word_data(self.addr, register)
        valor = self.swap(valor)
        return valor

    def readConfig(self):
        """ Read config reg. """
        r = self.readWord(ADS1115.confReg)
        return r

    def sendWordToConfReg(self, word):
        """ Send a word to the ADS1115 configuration register """
        # swap byte order
        word = self.swap(word)
        i2c.write_word_data(self.addr, ADS1115.confReg, word)

    def singleShot(self):
        """ Start an analog to digital conversion """
        #Read config register
        valor = self.readConfig()
        # Set "single shot" bit
        valor = valor | ADS1115.startConv
        self.sendWordToConfReg(valor)

    def ready(self):
        """ Return true if last task accomplished """
```

```
        valor = self.readConfig()
        ready = valor & ADS1115.done
        return ready

    def readConversion(self):
        """ Returns the conversion register contents """
        valor = self.readWord(ADS1115.convReg)
        return valor
```

4.1.7.- Class ADS1115 instantiation and usage.

In order to use the resources (methods) the class has you have to instantiate it. The code is (write it below the class code):

```
myBus = ADS1115(0x48)
```

it deserves some notes:

- 0x48: Is the argument that the *__init__* method is going to assign to the *addr* attribute. The line

  ```
  self.addr = address
  ```

  at *__init__* will assign the passed value to *self.addr*.

- The instance line (*myBus = ADS1115(0x48)*) does not include any other argument -apart from the address- so the attributes *ordinal* and *name* will get the default values “0” and “None” (see above 4.1.5, “__init__() method. Instance attributes").

- As soon as the code instantiates the class, its methods and attributes are *ready to use*. The way to use them is "*myBus.XXXX ()*" (so are all the attributes -for example *myBus.addr*)

This way, to read the conversion for the first channel, the code is (write it following the line “*myBus = ADS1115(0x48)*”):

```
myBus.singleShot()
while not(myBus.ready()):
      pass
print (myBus.readConversion())
```

The above code is fine. However, if you try to run it, the error “global name ‘i2c’ is not defined” will come out: the class *smbus* object ‘*i2c*’ is not defined yet. The following lines of code must be added *at the beginning* of the module (there is a somewhat more detailed explanation in 4.1.1, “Start conversion command”, see also Annex II, “Raspbian configuration”):

```
import smbus
i2c = smbusSMBus(1)
```

When running this code the binary value for the ADS1115 first channel conversion will be *printed*: it has to be a value in between 0 and 65535 (both included; 65535 is 2^16 – 1).

May be that the reader wants to interpret the binary value. “0” (“0000 0000 0000 0000”) means “0” Volts at the sensor connection; 32767 (a “0” followed by 15 “1”: “0111 1111 1111 1111”) is the biggest positive voltage value that the ADS1115 may convert according to the *reference value* programmed. As stated, the ADS1115 is working with the default -2.048 Volts-, so any voltage equal or grater that this one will give 32767; for intermediate -binary- values you have to use a *rule of three*. Values greater than 32767 correspond to negative values (that may not be converted nor connected).

One is tempted to include right here two lines of code to convert the binary value to decimal: we won’t. In due course the we will give a full explanation -and code- on the *calculation agent* classes; those are the ones in charge of performing this conversion. Patience.

4.1.8.- ADS1115. Configuration methods.

The coding written up to now is limited to using the ADS1115 with the default values (the ones it defaults to when it is switched on). Of course, this is a (very) limited use; remember that it can be also programmed to:

1. Select which channel is to be converted.
2. Select the *gain* (reference voltage). Every sensor has a maximum output voltage: it may depend upon the *maximum absolute value* the chip stands or upon another characteristic (encapsulation, for example). The own nature of the sensor makes a difference too: a thermocouple gives barely one mV (millivolt) at ambient temperature (they are rarely used for such purpose) while a LM35 outputs 250 mV at 25 centigrade degrees. The ADS1115 may be trimmed for every situation; the *top of scale* value may be selected from 0.256 V to 6.144 V in eight steps: the lowest reference -0.256 V- suits a thermocouple measuring hundreds of degrees and one of the intermediates -1.024 V- is good for a LM35 that is not intended to be used over 100 centigrade degrees (1.00 Volts).
3. Select the *sample rate* (SPS). The ADS1115 samples at a steady rate (around 1000 samples per second; see exercise below); you can tell it indirectly to average samples to increase accuracy by setting the SPS from 8 -samples per second- to 860. “8” SPS means that the ADS1115s will average 125 samples; “860” is not to average at all.

4. Set a warning threshold that the ADS1115 uses to give an alarm signal (not covered in the blog).
5. Set the *continuous conversion mode* (in opposition to the *single shot mode*; is not covered in the blog either).

To program the ADS1115 the issue is just about modifying the configuration register: every option means changing different bits; the trick is how to modify the required bits while leaving the others untouched. A similar -though a little simpler; keep reading- issue was solved to set the *single shoot* bit (see 4.1.1, “Start conversion command”; *singleShoot* method). Setting a bit unconditionally to “1” requires just a logical “or” operation; setting several bits to “0” or “1” is a bit more difficult. We need four steps to do so.

1. Read the configuration register.
2. Set the bits to change to “0” (by means of a logical “and”).
3. Set to “1” the bits that have to take this value.
4. Send the result back to the configuration register.

The procedure may seem a little wordy: it is easy to understand, though. As a first step let us adding some new constants to the class attributes:

```
                                  #O|MUX|PGA|M|DRT|COMPR
reset        = 0b0000010110000000 #0|000|010|1|100|00000
# Start conversion
startConv    = 0b1000000000000000 #1|000|000|0|000|00000
# “Done” bit
done         = 0b1000000000000000 #1|000|000|0|000|00000
# Channel selection (Multiplexer)
channelReset = 0b0000111111100000 #0|000|111|1|111|00000
setChannel1  = 0b0100000000000000 #0|100|000|0|000|00000
setChannel2  = 0b0101000000000000 #0|101|000|0|000|00000
setChannel3  = 0b0110000000000000 #0|110|000|0|000|00000
setChannel4  = 0b0111000000000000 #0|111|000|0|000|00000
# Gain
gainReset    = 0b0111000111100000 #0|111|000|1|111|00000
gain_6_144   = 0b0000000000000000 #0|000|000|0|000|00000
gain_4_096   = 0b0000001000000000 #0|000|001|0|000|00000
gain_2_048   = 0b0000010000000000 #0|000|010|0|000|00000
gain_1_024   = 0b0000011000000000 #0|000|011|0|000|00000
gain_0_512   = 0b0000100000000000 #0|000|100|0|000|00000
gain_0_256   = 0b0000101000000000 #0|000|101|0|000|00000
# Samples per second
SPSReset     = 0b0111111100000000 #0|111|111|1|000|00000
SPS_8        = 0b0000000000000000 #0|000|000|0|000|00000
SPS_16       = 0b0000000000100000 #0|000|000|0|001|00000
SPS_32       = 0b0000000001000000 #0|000|000|0|010|00000
SPS_64       = 0b0000000001100000 #0|000|000|0|011|00000
SPS_128      = 0b0000000010000000 #0|000|000|0|100|00000
SPS_250      = 0b0000000010100000 #0|000|000|0|101|00000
SPS_475      = 0b0000000011000000 #0|000|000|0|110|00000
SPS_860      = 0b0000000011100000 #0|000|000|0|111|00000
```

the preceding code is just an ADS1115 datasheet portion crumbled so that it can be used later with ease. One way or another every one of the listed values have to be used later to program the ADS1115. There are just three new reset words –*channelReset, gainReset* and *SPSReset*- that ease the coding, as we will see a bit later.

Such a way, to select channel three you have to write code for the following four tasks:

1. Reading the configuration register.

2. Set to “0” the bits correspondent to *channel selection* (by means of a logical *and of* the converted value and the *word channelReset*.

3. Set to “1” the needed bits by executing a logical *or* of the result of step 2 and *setChannel3*.

4. Send the result back to the configuration register.

And so on.

We have organized things so that we need a different method for each one of the three features -*gain, channel, SPS*-, because each reset (step “2”) uses a different word –*channelReset, gainReset* and *SPSReset*-. To further reduce the code it is a good idea grouping the *reset* and *set* words like this:

```
setChannels = (
    (channelReset, setChannel1),
    (channelReset, setChannel2),
    (channelReset, setChannel3),
    (channelReset, setChannel4),
)

setSPSs = (
    (SPSReset, SPS_8),
    (SPSReset, SPS_16),
    (SPSReset, SPS_32),
    (SPSReset, SPS_64),
    (SPSReset, SPS_128),
    (SPSReset, SPS_250),
    (SPSReset, SPS_475),
    (SPSReset, SPS_860),
)

setGains = (
    (gainReset, gain_0_256),
    (gainReset, gain_0_512),
    (gainReset, gain_1_024),
    (gainReset, gain_2_048),
    (gainReset, gain_4_096),
    (gainReset, gain_6_144),
)
```

The trick is having three *double tuple tuples*: each *double tuple* holding the couple of words for each one of the possible programming option: four channels, eight SPS’s and six gains. This pattern allows a quite simple coding:

```
myBus.setChannel(3)     # Select channel “3”
myBus.setSPS(7)         # Select 860 samples per second.
```

(remember that tuple numbering starts by “0”)

Let us see the methods code:

```
    def setChannel(self, channel):
        """ Sets a channel -multiplexer-"""
        self.programConfReg(ADS1115.setChannels[channel - 1])

    def setSPS(self, option):
        """ Sets the "samples per second" parameter"""
        self.programConfReg(ADS1115.setSPSs[option])

    def setGain(self, option):
        """ Sets the "gain" parameter"""
        self.programConfReg(ADS1115.setGains[option])
```

By having organized programming *words* so (*double tuple tuples)*, an utility method *programConfReg* is the single one to program the ADS1115:

```
    def programConfReg(self, lista):
        """ Sends a -programming- command to the ADS1115"""
        #Read config register
        valor = self.readConfig()
        # Reset value by lista[0]
        valor = valor & lista[0]
        # Set value by lista[1]
        valor = valor | lista[1]
        self.sendWordToConfReg(valor)
        return valor
```

the already coded *readConfig* and *sendWordToConfReg* methods being used.

We have completed the ADS1115 class core code: we can program the channel to read, the SPS it is going to be sampled with and the gain.

The ADS1115 has differential inputs. As the individual inputs do not accept negative voltage, it is about being able to connect sensors that may output negative values. The bits that are programmed and the sensor connection correspond to this table:

Bits “MUX”	Connection
000	Positive: AIN0; Negative: AIN1
001	Positive: AIN0; Negative: AIN3
010	Positive: AIN1; Negative: AIN3
011	Positive: AIN2; Negative: AIN3

1. Find the suitable attributes to program these options.
2. Find a new method (*setDiffChannels*?) to use differential inputs

An final issue remains undiscussed.

4.1.9.- *chooseVref* (reference voltage) method. *SetChannelGain* method.

In paragraph 4.1.8, “ADS1115. Configuration methods” we have explained that changing the gain of the ADC serves to set it according to the sensor expected span: measuring a thermocouple, which outputs millivolts, is not the same as measuring a sensor from which we can have some volts; you must adjust the ADS1115 *gain* (voltage reference) according to the sensor.

Therefore, the reference voltage has to be selected according to the sensor. For example, a conventional waterproof LM35 -the one you can find easily in Internet- is intended to work from 0 up to 100 centigrade degrees, that is to say from 0.00 to 1.00 Volts. As one of the reference voltages for the ADS1115 is 1.024 Volts, it suits perfectly to this sensor.

To set the reference you can use the *setGain* method: for the mentioned reference -1.024 Volts- the code would be *myBus.setGain(2)*. But, is it possible to automatically set the *voltage reference*?. It should, as each sensor type has a maximum voltage and the *ADCBus* -the ADS1115- has the voltage references it has.

The matter deserves a bit of thinking. Which -method of which object of which- class have to assign the correct -voltage- reference for a particular sensor?. To each physical sensor connected to the system an object of the class *sensor* is to be instantiated; the same happens with the *ADCBus* -the ADS1115-: each one will correspond to an object of the class *ADCBus*. In fact this issue of setting the reference value for later using it is a double stage process: when the sensor is instantiated it is assigned to an *ADCBus*. It looks like this -the instantiation one- is the right moment to look for the right reference; then the sensor class object stores the -reference voltage- value somewhere inside it (into an attribute, of course).

Now the issue of setting the reference prior to convert the analog value arises: the one who are going to actually convert is the *ADCBus* object, so we need to pass the voltage reference to it in some way. As we will see a couple of paragraphs below, just before giving the *ADCBus* the *convert -singleShoot-* command, we will pass it not just the sensor voltage reference, but the complete sensor object itself. It -the sensor object- has

attributes for both the reference voltage and the -*ADCBus*- channel number it is connected to; the former was calculated at the sensor instantiation time, the latter is passed as an argument when instantiating the sensor (just an instant before).

OK. But the issue of what the method to determine the voltage reference is and where is it to be placed remains unsolved.

In fact it maybe placed either into the -object of the- class sensor or into -the object of- the class *ADCBus*: both need to know about each other, so the sensor object is aware of some of the *ADCBus* features (binary format); and the *ADCBus* object some of the sensor ones (span, channel it is connected to) . We have finally decided to place it into the *ADCBus* class.

The ADS1115 class has to be completed by adding the voltage references. From the ADS1115 datasheet:

```
refVoltageList = (0, 0.256, 1.024, 2.048, 4.096, 6.144)
```

and the method to calculate the reference (and its place into the tuple) is:

```
    def chooseVref(self, sensorMax):
    """ Return an ADC reference index and value """
        anterior = self.refVoltageList[0]
        for position, i in enumerate(self.refVoltageList):
            if sensorMax == i:
                return (position - 1, i)
            elif ((sensorMax < i) & (sensorMax > anterior)):
                return (position - 1, i)
            anterior = i
        return (position - 1, anterior)
```

This method is included in the class ADS1115 from now on.

As stated before, to prepare the ADS1115 object before converting, the sensor object is passed to it so that the correspondent channel and gain are set. Our ADS1115 class already has two separate methods to do so, but it is much easier for the library *client* to pass the sensor info -object- just once. A new *setChannelGain* method will be on charge of calling the two individual ones. Its code:

```
    def setChannelGain(self, sensor):
        """ Set the channel # -multiplexer- and gain """
        self.programConfReg(ADS1115.setChannels[sensor.channel - 1])
        self.setGain(sensor.vRefPos)
```

The argument *sensor* is passed when calling this method. Yes, it is the *whole* sensor object. And yes, the two lines call the two class methods with arguments belonging to the sensor object: *sensor.channel* and *sensor.vRefPos*

Passing an object to a method as an argument entails certain implications. Let's say that the method code could modify the passed object, what is against one of the OOP paradigms. As it has been noted, in Python is impossible to hide an object to a *fool programmer* (a deeper explanation of what a fool programmer is for a better occasion), so let this comment being an opportune warning.

4.1.10.- Class ADS1115 complete code. Usage examples.

From https://*github*.com/Clave-EIDEAnalog/EIDEAnalog (EA_4111_ADS1115_COMPLETE_USAGE.py), you can download the class code. Should you are learning, the best is to copy-paste it so that you can mess about as much as you want. Remember you have to import smbus and instantiate it (see 4.1.7, " Class ADS1115 instantiation and usage").

Up to now we just know how to convert the first channel; now we are in position to read all the four channels, each one with its own gain. Do connect whatever you want to the ADS1115: sensors, potentiometers; tie them to "0" volts or to 3.3 Volts. Remember that you can connect the on board NTC (thermocouple) to the channel #1 (see figure 19). The following code performs a complete -the four channels- conversion.

```
myBus = ADS1115(0x48)

for i in range(1,5):
    myBus.setChannel(i)
    myBus.singleShot()
    while not(myBus.ready()):
      pass
    print ("channel", i, ":", myBus.readConversion())
```

the IDLE will show something like this:

```
channel 1: 17957
channel 2: 2533
channel 3: 8665
channel 4: 0
```

needless to say that the values depend on what you have connected to the ADS1115: the NTC outputs something around 18000.

The *time* module has a method -*time.time()*- that returns with sufficient precision the actual time (*now = time.time()* saves the current time -in seconds- in *now*; don't worry about the time count origin: *time=0* -it's irrelevant for the purpose of this exercise)

Using this method write a code -do not use *print* commands- to calculate the effective sampling rate of the ADS1115 (The default channel; change the "SPS" to 860. Tip: program a loop that executes the conversion a number of times; 1000, for example).

Result: around 400 samples per second. This *apparent* speed is low because the shuffle of commands -*read.word, write.word*- that the i2c bus has to perform. The default i2c bus speed -100,000 bits/s- is low compared to the conversion lapse. Annex II, "Raspbian Configuration" shows how to change the speed to 400,000 bits/s; should you change it you will notice an increase in the apparent speed up to approximately 580 samples/s.

4.1.11.- Class ADS1115. Caveats. Conclusion.

An ADS1115 chip control class has been fully discussed and coded. Some examples have been included and a second level ones have been proposed as exercises (All of them are available -solved- at https://github.com/Clave-EIDEAnalog/EIDEAnalog).

Exhaustive comments have been included, especially in what concerns the class code core. We hope the reader has to make just a reasonable effort to following them.

Beyond each method algorithms complexity there are some issues regarding the class coding and pattern that should be pointed out:

- While developing the code a subsidiary explanation of the ADS1115 functioning has been made. (Perhaps) it is worthwhile reading it before addressing the ADS1115 datasheet itself .

- No error detection code has been included in order to not complicate the overall development.

- Class interface: A great effort has been made in order to introduce the concept of *class interface* and the benefits and drawbacks of using it as an analysis tool. The ADS1115 class interface is shared with the other *ADCBus* classes: oneWire and arduino (see 6.1, “Class oneWire” and 6.2, “Class arduino”).

- *Class attributes* are used to store the attributes common to all the -ADS1115- class objects. This feature may be a bit *heterodox*, so a pertinent warning has been issued.

- Class usage examples are included as the code is presented. Some examples are not directly included in the text, but presented as exercises whose solution can be found (and downloaded) at https://github.com/Clave-EIDEAnalog/EIDEAnalog.

5.- *Sensor* class. Calculation agents.

Once coded the *ADS1115 class* it is time for the *sensor class*.

The ADS1115 class development was made stab by step and was fully commented. Moreover, the analysis was divided into two halves, such a way for the coding being simpler.

In this chapter we will analyze and code the *sensor* class. We will also develop a supporting family of classes that will be in charge of calculating the *readouts* -the *human readable* values- for every sensor. These classes are to be created by using *class inheritance*, another cornerstone of OOP; at the same time the term *polymorphism* is to be introduced and explained: it is another pillar on OOP theory).

Logical objects vs. *real* objects: a common noob's mislead is to think on the OOP *object* term as corresponding -always- to a physical object: a person, a car, an animal, a *sensor*. In fact an *object* rarely corresponds to a physical one; even this being the case, the code *object* is a logical construct: maybe its attributes being *name* or *address*, but, normally, the object exists as long as it does something in the context of the program, not because the program deals with *persons* or whatever.

Finally, the cream on the cake are the *composite objects*, also basic to the OOP: we will use them when instancing the class sensor.

5.1.- Sensors.

Let's see which sensors are to be used in the library examples.

1. **Analog sensors.** An *analog readout* is the output of a sensor that is proportional to the measured parameter (temperature, pressure, humidity); to calculate the actual value one has to make a rule of three (hence the term *analog*. It is not a joke). These sensors are the most common ones. The temperature sensor LM35 outputs a volt for every 100 centigrade degrees (centigrade degrees): 0.0 Volts are 0 centigrade degrees; 1.0 Volts are 100 centigrade degrees and so on.

   The sensor output uses to be a *voltage*. The use of *current* is common also: it is much more immune to interference.

2. ***Quasi proportional* sensors:** They are quite similar to the analog sensors just described. The only difference is that they add an offset to the output so that this is not "0" when the measured parameter is "0". The LM50 is identical to the LM35 except for it gives an offset of 0.5 Volts: 0 V → -50 centigrade degrees; 0.5 V → 0 centigrade degrees; 1.0 V → 50 centigrade degrees …

3. **Tabulated sensors:** There are sensors whose output is neither proportional even linear with relation to the measured parameter. Its use is declining (those in sections “1” and “2” are increasingly cheaper), but you can’t discard facing one of this; in fact, a very popular ADS1115-based ADC board has an on board thermistor -a temperature dependent resistance- that can be easily connected to the first channel of the ADC (See Figure 19). We will use it as an example of this type of sensors in the blog.

   (It is not only about non linear sensors: many times a device gives a voltage that varies according to some parameter and the dependency is tabulated: an example is the water flow through a valve whose stem position is sensed by a linear sensor. A bulb luminosity depending on the applied voltage or the vapor pressure of water depending on its temperature are other examples. May be that a “formula” exists for the dependency, but many times it is much easier to use a table. (By the way, you can show off in front of your friends in case you know what the vapor pressure is).

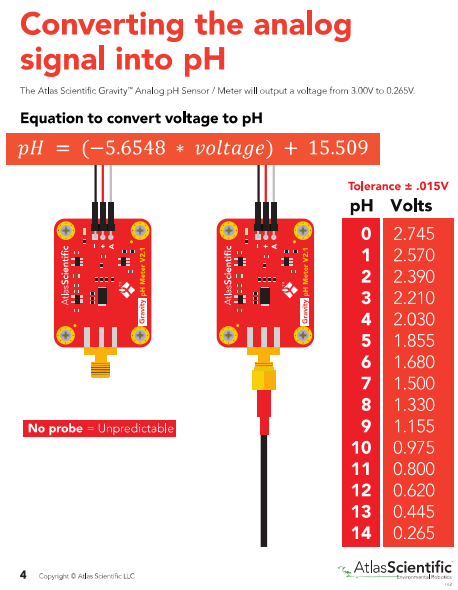


Figure 23. pH sensor table (Atlas Scientific)

4. **ASCII (or almost ASCII) sensors:** There are sensors that are actually miniaturized devices that include an ADC. Along the blog we will use the temperature sensor DS18B20, that gives the temperature directly in ASCII; it encloses not only the temperature sensor itself, but an ADC and the circuitry to convert the voltage to ASCII; see 2.3.1, “Bus *oneWire*”).

5.2.- Class sensor. Preliminary approach.

Let us start by analyzing the *sensor* class. An issue must be pointed out: although the class is to be instantiated once for each physical sensor in the system, these instances deal with converting what the ADC delivers for each sensor to the *readable* (human) format. It is a subtlety: the -*sensor* class- code has nothing to do with a physical sensor *thing*, but with the information that the ADC delivers and its conversion to readable format (on the other hand, it is not easy to imagine what code could be written related to a sensor that is nothing else that a little silicon chip withstanding high -or not so high- temperature, pressure, etc.).

About the use of the term *sensor* for the class: It is not the best. As will be seen almost immediately, it should actually be called "*sensorChannel*" (or something like that), for each -sensor- class instance corresponds to a physically different sensor that, in turn, might be connected to different types of *ADCBus*. To an LM35 connected to an ADS1115 an object of this class will be assigned; its attributes are a bundle of the ones of such a sensor -the physical sensor- and those of the ADS1115 chip. An LM35 connected to an arduino will have an object with different attributes than the one instantiated for the same LM35 connected to an ADS1115. Two LM35 type sensors connected to the same ADS1115 will also be represented by two different objects (in this case the only different attribute will be the channel number to which each one is connected -*self.channel*-).

The sensor class won't have many methods, and only a few of them are essential: 1) *readout*: the one that calculates (better: tells somebody else to calculate) the readout from what is delivered by the ADC and 2) the one that assigns to the -*sensor*- object A) the *calculation agent* (the algorithm that transforms what is delivered by the ADC into the readout), B) the *ADCBus* reference voltage to be used for this sensor, and C) the conversion table -if any-. With the exception of these two methods, the sensor class has mainly an *administrative* role. It is a data container -type of sensor, gain, *ADCBus* to which it is connected, conversion tables for tabulated sensors, etc.-; you won't find a single line of code inside the *readout* method: it passes the *hot potato* to the *calculation agent*, previously instanced depending on the type of sensor and the *ADCBus* to which it is connected. See below.

Hence, for the *sensor* class is as important analyzing the *attributes* than the *methods*. Let's see which ones -attributes- are essential for the objects of this class performing their job. Namely:

1. Sensor type: for example, LM35
2. Which *ADCBus* is it -the sensor- connected to: there may be several working at the same time connected to the Raspberry.
3. Which *ADCBus* channel the sensor is tied to.
4. Which reference voltage the sensor is assigned.
5. The sensor ID (if available).

"*Which ADCBus is it connected to*" actually refers to the *object* that has been instanced for the *ADCBus* (see 4.1.7, "Class ADS1115 instantiation and usage" as an example). This object is passed as an argument -to the sensor object- at the moment the instantiation command is issued: this is the way the sensor object knows what format the data is going to arrive. In fact passing an object as an argument to other object makes all the data into the former available to the later (see a deeper explanation a couple of paragraphs below).

The need for the *sensor* object knowing which channel it is connected to is obvious (remember that the pattern -the library architecture- used makes the sensor object responsible for telling the ADCBus about everything it needs to performing the AD

conversion: it is obvious that it -the ADCBus- needs to know which channel -sensor- has to be converted).

The sensor ID is its *plate*. In this blog the only ones having ID are the DS18B20 (the ones that are connected to the Raspberry by means of the oneWire bus; the ID is recorded into each one electronics). As will be seen in due course the oneWire object will have methods both to get the ID from the sensors and to perform the conversion by using the sensor ID (instead of using the *sensor channel*).

A *dictionary* "**standardSensorsData**" in the *class attributes section* of the class *sensor* will store the data for the different types of sensors the library deals with. The dictionary holds the characteristics common to all sensors of a type. Thus, for example, the LM35 type will correspond in the table with the parameters "100, 0, 1.0" that are the gain (100 centigrade degrees / volt), the zero (0 volts -> 0 centigrade degrees) and the maximum output voltage that these sensors output (1.0 volts -> 100 centigrade degrees); the information for a *tabulated sensor* has the name of the *text file* that contains the table itself.

```
standardSensorsData = {
    'DS18B20':      ['DS18B20', 0.001, 0, 0, None],
    'LM35':         ['LM35', 100, 0, 1.0, None],
    'LM50':         ['LM50', 100, 50, 1.5, None],
    'rawVoltage' :  ['rawVoltage', 1, 0, 5.0, None],
    'ADS1115_NTC':  ['ADS1115_NTC', 1, 0, 3.3, 'ADS1115_NTC'],
    {
```

For the sensor class code explanation -and comprehension- being optimal, is much better deferring it till the *calculation agent* classes are introduced. The sensor class code is presented and discussed in 5.4, "Class sensor code. Instantiation"; you may have a look over there right now, but better following the proposed sequence. Keep reading.

5.3.- Readout calculation algorithms. *agent* classes.

All the LM35 sensors are the same (and all the LM50 are the same, and all the DS18B20 are the same, …). The point is that they -the LM35- give 1.0 volt for every 100 centigrade degrees (or 10 mV by every centigrade degree, as you wish). You can use -read- them without any ADC: with just a battery (3.3 – 5.0 volts; red and black wires) and a polymeter (yellow wire) you can read the temperature.

No all the sensors -the same type- are *exactly* the same: there may be very small differences that require *calibration*. For the purpose of this blog we can assume that all of them are the same.

On the other hand, not all the ADCs work the same: the ADS1115 delivers the conversion in two's complement (16 bits), while an arduino "ProMini" gives the reading in 10 bits direct binary (0.0 V. -> 00 0000 0000; 5.0 V. 11 1111 1111, assuming that the arduino is powered

at 5.0 V: such a case 11 1111 1111 would not correspond exactly to 5.0 V, but to 5.0 * [1023/1024]).

The point is that the *calculation algorithm* is not only dependent on the sensor, but also on the ADC.

Let us go back to the *encapsulation* concept: we ask the *sensor* class objects to calculate the readout (by passing as an argument the ADCBus output). Hence they -the *sensor* class objects- need to have the resource to do so. On the other hand, the calculation algorithm does not belong to a specific sensor either: the difficult part -converting binary information to decimal; let's also include into the difficult category to change the decimal point of the reading of the oneWire sensors– depends on which ADC the sensor is connected to.

It is advisable to to start thinking on some class(es) -*calculation agent(s)*- whose objects manage the above suggested algorithm(s): a *binaryAgent* class, for example, will produce objects that will be able to calculate the readout from the data delivered by ADS1115 (or an arduino); needless to say that the agent will *know* too what are the other necessary parameters, those of the sensor.

5.3.1.- Binary calculation agent (binaryAgent). Inheritance.

The ADS1115 and the arduino convert to different formats (16 bits two’s complement the former and 10 bits unsigned the latter). There are, on the other hand, dozens of ADC converters that can be connected to a Raspberry, some will convert to 10 bits, others to 12, 13, 14, 15 or 16 (and there exist up to 24 bits: anyone with data acquisition skills questions anything above 16 bits; 20 is a real luxury; 24 may be actually too much); many give the data in *two's complement*, although the direct format -unsigned- is also frequent.

Therefore the first algorithm, based on a bunch of parameters -see below-, calculates the *readout* from the binary information provided by an ADC converter. This algorithm -the reader guess that the “algorithm” is (part of) what we have been calling *calculation agent*- has -*attributes*- these parameters:

1. Number of bits (16 for the ADS1115, for example).

2. Whether the binary data is signed (negative numbers).

3. The gain, the zero and the reference voltage used for every particular sensor (100, “0” and 1.024 V for an LM35 connected to an ADS1115; should you don’t understand why 1.024 volts please go to 4.1.9, “*chooseVref* (reference voltage) method. *SetChannelGain* method”).

As will be seen in due course, the code of this class -*binaryAgent*- does not depend on the bus to which the sensor is connected, nor on the sensor. It is -the agent- *general purpose. Or* at least it is so for all linear sensors -the voltage they deliver varies linearly with what they measure- whose information comes in binary format. Every time a sensor is instanced, the sensor instance itself will instantiate the calculation agent that it needs to calculate its readout.

Please find below the complete code for this class. Code deciphering -except for the *__init__* method, which is quite a different matter-, the reader having a minimum knowledge of binary algebra, should not imply any problem. Even in the case you are an expert, read it diagonally; before going deeper into it is better reading the paragraphs corresponding to the analysis of the *agent* classes -*ASCIIAgent* and *tabulatedAgent* – that follow and that will lead you to the next point (5.3.3., "Class *calculationAgent*"). Sure you will have to go back to this -*binaryAgent*- code.

```
class binaryAgent(calculationAgent):

    def __init__(self, Vref, gain, zero, bits, twosC, table=None):
        calculationAgent.__init__(self, Vref, gain, zero, bits, twosC, None)
        self.name = 'binary'

    def binaryToDecimal(self, value):
        if self.twosC:
            # Two's complement format
            value = value - int((value << 1) & 2**self.bits) # Int value
            value = value / float(2**(self.bits - 1))
        else:
            value = value / float(2**self.bits)

        value = ((value * self.Vref) - self.zero) * self.gain
        return value

    def convert(self, value):
        return self.binaryToDecimal(value)
```

Conversely, the *__init__* method looks quite cryptic. It actually is. It is the first time on the blog that we have come across a class that *inherits* from another; the *calculationAgent* class this time. The code heading the class itself makes it explicit: "*class binaryAgent (calculationAgent)*".

Let us have a first approach to the *inheritance* issue. There will be -there is- a *parent* class (*calculationAgent*) that has *attributes* -*data* and *methods*- that concern to the *child* class, *binaryAgent* this case. Let's take a preliminary look -also diagonally- at the code of the ancestor -*parent*- class:

```
class calculationAgent():

    def __init__(self, Vref, gain, zero, bits, twosC, table=None):
        self.Vref = Vref
        self.gain = gain
        self.zero = zero
        self.bits = bits
        self.twosC = twosC
        self.table = table

    def convert(self):
        pass
```

it appears to be the other half of *binaryAgent.* The latter has no attributes; they seem to be defined and initiated within the *calculationAgent* class code.

(*calculationAgent* class code will be discussed in detail in section 5.3.3, “class calculationAgent”; we have included it to ease the explanation of the *binaryAgent* code).

The *inheritance* matter consists on:

- The *child* class -*binaryAgent*- endorses all the methods and attribute definitions of the *parent* class -calculationAgent-.

- Methods that appear both in the *parent* and the *child* class do not have any conflict each other: the ones in the *child* class prevail: *__init__* and *convert* this case.

- For the *binaryAgent* and *calculationAgent* affair the *child* class (the former) *__init__* method calls the parent (the latter) one and passes to it the necessary arguments: *calculationAgent.__init__(self, Vref, gain, zero, bits, twosC, None).* The five leading ones have just being passed at the moment as arguments by the *client code* when instantiating the *binaryAgent* class: this one just adds a *None* one because a binaryAgent has no table associated.

(a) Write a client code to use the *binaryAgent* class: the goal is to get the readout corresponding to a 10 bits positive value -binary code- (not two’s complement) whose full scale is 5.0 (Vref = 5 ; 0 → 0, 1024 → 5.0). Write a code that calculates the readout for these input values: 2, 5, 10, 20, 50, 100, 200, and 1000.

At the end of paragraph 4.1.7, “Class ADS1115 instantiation and usage”, we proposed converting the ADC reading to a human readable format. Add to the code proposed there the necessary code to convert the ADC reading (Tip: create an instance of the binaryAgent and use it - "convert (value)" - to convert the output).

5.3.2.- ASCII calculation agent (ASCIIAgent).

As said above (2.3.1, “oneWire Bus”) the *oneWire* bus is a (quite) special case of an analog to digital converter. It is not really a single ADC, but as many ADCs as sensors connected; by means of a singular (and very slow) technique, the driver of this bus places the readings already converted by the wired sensors into text files: this files contents are, among other information, the reading of the sensor (for example the temperature in the case of the DS18B20) although with the decimal point outplaced (the DS18B20 it gives it in thousandths of centigrade degree, that is, 23456 instead of 23.456 centigrade degrees).

Hence, a *calculation agent* is needed for the case; It will be a trivial algorithm, since all it has to do is to change the decimal point place.

```
class ASCIIAgent(calculationAgent):
    """  'ascii' calculation agent """

    def __init__(self, Vref, gain, zero, bits, twosC, tabla):
        calculationAgent.__init__(self, Vref, gain, zero, bits, twosC, None)
        self.name = 'ASCII'

    def ASCIIToDecimal(self, value):
        """ Return a float from (ascii) """
        return value * self.gain

    def convert(self, value):
        """ Return a float from the raw value (ascii) """
        return self.ASCIIToDecimal(value)
```

Should you have analyzed the *binaryAgent* class code, this one has to look familiar to you: the *__init__ method* -except for the *self.name = 'ASCII'* line; sure you're getting the hang of it- is identical, so it is -almost- the method *convert*. It just changes the core of the matter: the *algorithm* itself *-ASCIIToDecimal* method-; so that everything explained in previous paragraph is valid for this class.

5.3.3.- *calculationAgent* class.

The agents *binaryAgent* and *ASCIIAgent* contain what is strictly necessary to execute the part of the task that differentiates one from the other: compare them. Should the library we are developing had the need of just one of them (if we only had to deal with ADS1115 the *binaryAgent* would suffice, if the project included the oneWire bus only the *ASCIIAgent* is enough) they probably would not exist as such, and their function would be implemented as a method of the *sensor* class.

But the fact is that we *do* have two different algorithms (not only that: we have great *prospects of future* for the library).

Being that the situation, the need of several calculation algorithms based on -different- sensors and -different- ADC converters, we have decided to design a *family of calculation agents*, see figure 24. The *calculationAgent* class will be the common ancestor to all of them (except for the *tabulatedAgent;* ignore this fact for the moment). The *calculationAgent* class will contain all the attributes and methods shared by its *heirs* though, as will be explained a in due course, there is only one -common- method, *convert*, which is just enounced in this -*calculationAgent*- class.

This is the class code (a detailed explanation follows). Please, read it carefully as many times as needed to fully understand it.

```
class calculationAgent():

    def __init__(self, Vref, gain, zero, bits, twosC, table=None):
        self.Vref = Vref
        self.gain = gain
        self.zero = zero
        self.bits = bits
        self.twosC = twosC
        self.table = table

    def convert(self):
        pass
```

```
class calculationAgent():
```

Class has no *ancestors: "*class calculationAgent**()***"*

```
def __init__(self, Vref, gain, zero, bits, twosC, table=None):
    """ Init agent data. 'Table = None' unless specified """
    self.Vref = Vref
    self.gain = gain
    self.zero = zero
    self.bits = bits
    self.twosC = twosC
    self.table = table
```

__init__ is the *class constructor* method. It does nothing else than assigning to the class attributes the arguments that are passed to it (if nothing is passed for "*table*" it assigns the default value *None* to it; "*... table = None*)". As all the calculation agents are directly -or second generation- inheriting from this *calculationAgent*, these attributes are part of all of them; they -the attributes- are given initial values when the child *__init__* method calls the *__init__* parent method. Now you can go back on the code of the *calculationAgent(s)* already presented and you will better understand some code lines that seemed ununderstandable.

```
    def convert(self):
        pass
```

Every *calculation agent* has its own *convert* method: they are all different to each others' and they *override* the one in the common ancestor class -*calculationAgent*-.

Let's summarize a bit this *inheritance* issue. It is a bit tricky: may be you have to go back and forth a couple of times to fully understand it.

- *binaryAgent* and *ASCIIAgent* __init__ methods are similar: they both call the *__init__* method of the *calculationAgent* class, passing it the arguments that they, in turn, received from the *client* that instantiates them.

- Including as an argument in the definition of a class the argument "*calculationAgent*" -class *binaryAgent (calculationAgent)*- causes the class so defined to *inherit* what is defined in the *ancestor* class -*calculationAgent*-. For the moment, the *binaryAgent* and *ASCIIAgent* classes do not inherit much from the common ancestor, but things will happen.

The moment has arrived to analyze and summarize the attributes common to all the *calculation agents*. (Two future classes -*binaryTabulatedAgent* and *ASCIITabulatedAgent*- are included*)*

Attribute	Description	binaryAgent	ASCIIAgent	binaryTabulatedAgent	ASCIITabulatedAgent
`Vref`	*Voltage reference* (maximum voltage at the sensor output): Is the reference used to convert the sensor output. The agent needs it to know what the *top of scale* is.	X	-	X	-
`gain`	Gain: output/input ratio. “100” for the LM35 (one volt every hundred centigrade degrees).	X	X	X	X
`zero`	Offset output voltage corresponding to zero input (the LM50 gives 0,5 V at 0 centigrade degrees)	X	-	X	-
`Bits`	Number of bits (16 for the ADS1115, for example)	X	-	X	-
`twosC`	Negative input: *True* if the ADCBus accepts negative values (normally handled in **two’s c**omplement)	X	-	X	-
`table`	Conversion table (a *sensorTable* class object. See 7, “Sensor tables [*sensorTable]*”).	-	-	X	X

Calculation agent classes share these attributes (not all of them). Should a common *ancestor* class does not exist it would be necessary to repeat them in each one of the declaration of the classes.

(Not just this: if you decide to do some checks on the attributes -*gain* cannot be zero, nor the number of bits- it is very convenient that those -the checks- are *centralized*, that is, as *calculationAgent* class methods. Things will happen again).

5.3.4.- Tabulated agent (*tabulatedAgent*).

There are sensors whose output is neither proportional nor linear (see 5.1, “Sensors. Tabulated sensors”). Such a case we need an algorithm -whose logic hasn’t any similarity with any of the previous *agents*- to which an input is passed and returns the corresponding value from a table that describes its behavior. We will call it *tabulatedAgent*.

The table will not be passed as such (list or Python tuple) when instantiating the agent. For this purpose, we will devise a new class (see 7, “Sensor tables [sensorTable]”) whose goal is managing (reading the table from a text file, checking it, looking up into) the table; what will be passed to the agent will be an instance of this class.

Table content are *discrete* -individual- values of the sensor behavior, so to calculate the intermediate values it is necessary to use some criteria (try with excel: enter a table, convert it to a graph and add a *trend line*; there are a dozen possibilities). In this blog we will choose the most modest of all, which is the *linear interpolation*: this is the one used in *sensorTable;* it requires just geometric calculations.

The *tabulatedAgent* class is a somewhat spurious member of the *calculation agent* family: we could have made it directly *heir* to the *calculationAgent* class, but this would have had the inconvenience of making it to carry with too many attributes (*titles*?) that are not worth at all: should you want to use it stand alone -from the rest of the library-, the instantiation would border on the incomprehensible ("*agent = tabulatedAgent (0, 0, 0, 0, 0, 'NTC')*", for example). It is much better that it joins the family as an *uncle in law*, without so many attributes.

The class code is:

```
class tabulatedAgent():

    def __init__(self, table):
        self.table = table
        self.name = 'tabulated'

    def lookupTable(self, value):
        return self.table.lookup(value)

    def convert(self, value):
        return self.lookupTable(value)
```

```
class tabulatedAgent():
```

Class does not inherit from nobody.

```
    def __init__(self, table):
        self.table = table
        self.name = 'tabulated'
```

table is the object of the class *sensorTable* that has been passed as an argument at the instantiation time.

```
    def lookupTable(self, value):
        return self.table.lookup(value)
```

(self.table.)lookup is a *sensorTable* object method: it just looks up in the table to extract the data corespondent to *value*.

```
    def convert(self, value):
        return self.lookupTable(value)
```

for the sake of consistency, a *convert* method is included (it just uses *lookup* to calculate the output).

Write the code to instantiate the *tabulatedAgent* class. Assume that you already have an object of the *sensorTable* class called "NTC".

5.3.5.- Multiple inheritance. *BinaryTabulatedAgent & ASCIITabulatedAgent* classes

If there is a *tabulated sensor* in the project (for example, the NTC of the ADS1115 board, see figure 19) what happens is that it is connected to an ADCBus channel, therefore delivering a reading in a certain format (binary, ASCII, …). This -the reading- must be converted to a decimal format -like every other sensor- prior to passing it to the *tabulatedAgent* that, finally, looks up into the table to calculate the definitive readout.

We face a double challenge, that is the aggregation of two ones already solved: 1) converting an analog reading -*binaryAgent, ASCIIAgent*- and 2) looking up into a table -*tabulatedAgent*-. We devise a new class -perhaps two- combining two other classes skills. As we want to explore the basics of OOP, and being the jobs assigned to this new class(es) the ones of some other -and existing- classes, we will try the *multiple inheritance* Python's ability. Let's see it.

```
class binaryTabulatedAgent(binaryAgent, tabulatedAgent):

    def __init__(self, Vref, gain, zero, bits, twosC, sensorTable):
        binaryAgent.__init__(self, Vref, gain, zero, bits, twosC, sensorTable)
        tabulatedAgent.__init__(self, sensorTable)
        self.name = 'binary tabulated'

    def convert(self, value):
        return self.lookupTable(self.binaryToDecimal(value))
```

This is the complete code for a class that implements both a binary to decimal conversion then a table extraction. The goal is to get a result that models a tabulated sensor. Having in mind that the line *self.name = 'binary tabulated'* can be omitted without affecting the class performance, it looks like the *fine-grained* development of the *calculation agent* classes family starts being rewarded.

BinaryTabulatedAgent class. Code analysis.

```
class binaryTabulatedAgent(binaryAgent, tabulatedAgent):
```

Is the class *declaration*. As arguments both ancestor classes are listed: can be as many as needed. In this case the class inherits from two that are previously declared: *binaryAgent* and *tabulatedAgent*, and this time the *inheritance* matter is the hearth of the issue: by inheriting from these two *ancestors* the new class has, in addition to the necessary attributes (Vref, gain, etc.), all the methods they -the ancestors- house. Therefore, it knows how to *convert* a reading in binary format to decimal, and it knows how to *look up into* the table to convert it to its final value.

```
    def __init__(self, Vref, gain, zero, bits, twosC, table):
        binaryAgent.__init__(self, Vref, gain, zero, bits, twosC)
        tabulatedAgent.__init__(self, table)
```

The constructor of this class -*__init__*- uses those of the ancestors to what it passes the parameters that, in turn, have been passed to it when it has been instantiated; thus, all the attributes are declared. Note that while on the call to the *binaryAgent "__init__" self.table* will be assigned -provisionally- to *None*, on the call to the "*__init__"* of *tabulatedAgent* that value will become the argument passed as *table* in the *binaryTabulatedAgent* instantiation. (Take a breath and re-read it a couple of times: should you still do not fully understand it, don't worry, get *the melody,* go forth and back more calmly when you are done with everything related to the *calculation agents).*

```
    def convert(self, value):
        return self.lookupTable(self.binaryToDecimal(value))
```

This -new- method *convert* is the one that best summarizes inheritance benefits: the new class inherits all its ancestors methods: hence, it *knows* how to convert a binary content (*self.binaryToDecimal(value)*) and how to look into a table searching for a value (*self.lookupTable(...)*).

The new *convert* method overrides any *convert* ones its ancestors may contain (by the way, both of them this case. Should a specific *convert* method had not been defined for the new class, the one of the ancestor furthest to the left would have prevailed in the definition of the class: that of the *binaryAgent* class, in this case).

Using the same term -*convert*- for every *converting* method among all the members of the family is not unintentional. Though in a little unorthodox way it corresponds to the *polymorphism paradigm* in OOP: the point is using the same name for different contents (algorithms in this case).

Good. The above is not all you need to know about -multiple- inheritance, but it is a lot. The code for the class *ASCIITabulatedAgent* being:

```
class ASCIITabulatedAgent(ASCIIAgent, tabulatedAgent):

    def __init__(self, Vref, gain, zero, bits, twosC, sensorTable):
        binaryAgent.__init__(self, Vref, gain, zero, bits, twosC, sensorTable)
        tabulatedAgent.__init__(self, sensorTable)
        self.name = 'ASCII tabulated'

    def convert(self, value):
        return self.sensorTable(self.ASCIIToDecimal(value))
```

whose meaning should be crystal clear (as long as you fully understand the *binaryTabulatedAgent* class code).

5.3.6.- *calculation agent family*. Summary. Conclusion.

Next chart summarizes previous paragraphs:

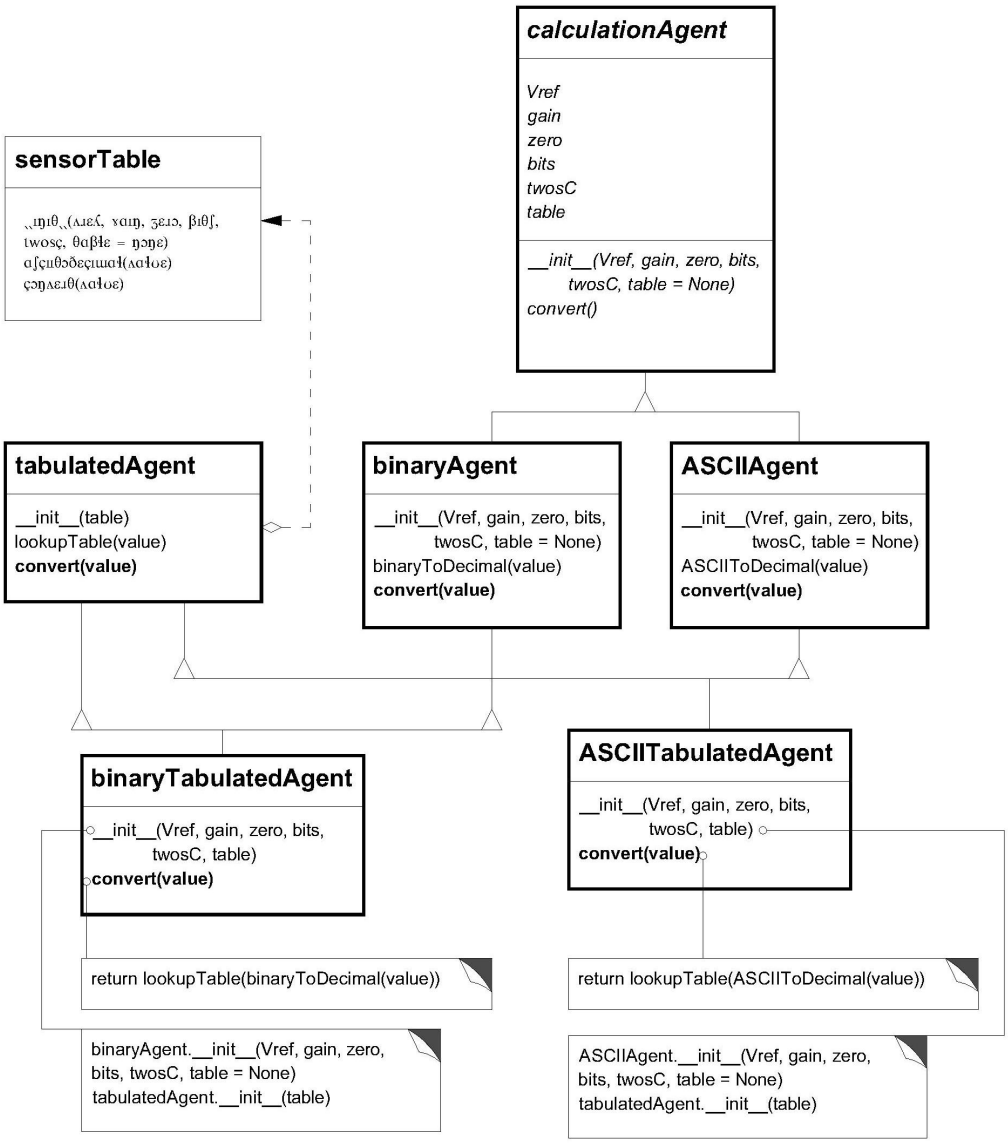


Figure 24. *calculation agent* family tree.

- The library under development solves the problem of calculating the readout corresponding to a particular sensor connected to a certain ADC by means of a family of *calculation agent* classes; these implement the necessary algorithms according to the format of the ADC converter and the characteristics of each sensor. To implement the classes that house the algorithms, the OOP ***inheritance*** technique is used.

- The ***calculationAgent*** class acts as a *common ancestor* for all classes in the family (except for *tabulatedAgent* which, however, is an ancestor of the last *two generation* classes). This class -*calculationAgent*- does not house any of the calculation algorithms; it simply has the attributes common to all classes (and which are necessary, however, for the actual calculation of the readouts). In an extended version of the library it will centralize, at least, the checking of the passed arguments for the instantiation (error -*exception*- management).

- Two classes, ***binaryAgent*** **and** ***ASCIIAgent***, implement the basic algorithms (data in binary and ASCII formats).

- A ***tabulatedAgent*** class implements a simple algorithm (actually the algorithm is delegated to an object of the *sensorTable* class) to looking up in a table that contains discrete values (couples of reals). It is about handling either sensors that are not linear (NTC), or sensors -processes- whose readout -behavior- is tabulated. This class is instantiable *per se*, and would serve to implement a *table query* (in the context of ADC conversion or any other).

- Two *double-inheriting* classes, ***binaryTabulatedAgen****t* and ***ASCIITabulatedAgent***, inherit the algorithms of the three classes described in the previous two paragraphs.

- This technique -***(multiple) inheritance***- leads to implement complex algorithms by using quite simple classes -coding-.

- The calculationAgent class is an **abstract class**: it cannot be instantiated (better, it could be instanced, but it is useless). This type of classes serve to enunciate the methods common to all the *child* classes (the *interface* of the class; we will come back to this concept once and again).

- **convert**: all the calculation agent classes have a *convert* method whose use is exactly the same in all of them: convert the passed *value* to the readout. This technique is known as *polymorphism*: the same command uses different algorithms that, in turn, are *encapsulated* in different objects whose classes have defined different methods with the same name and purpose.

- **Class explosion**. The -multiple- inheritance technique used for the development of this family of classes can sometimes lead to what is called “class explosion”. A detailed explanation of the term -phenomenon- is beyond the scope of this blog; Let's say it has to do with the -many- combinations that could appear if there are (many) more algorithms to implement in addition to those that already exist. Let’s assume that in addition to the format -binary, ASCII- and the eventuality of needing -or not- a table, there is a third possibility on converting: say that the result may -or may not- be multiplied by two -it is an improbable nonsense, but it serves to illustrate what would happen. This case, following the pattern we have already used, there would be another class (*doubleAgent*, or something like that) that would multiply the reading by two. Hence, we would need the *binaryDoubleAgent, ASCIIDoubleAgent, binaryTabulatedDoubleAgent, tabulatedDoubleAgent*, ... classes to implement the corresponding algorithms. It is obvious that this is not functional, so this technique is not recommended if there is no a reasonable certainty that something like this won’t happen.

Noise is measured in decibels (db). It is a unit whose definition is conventional. An old device that delivers a signal in volts that senses the ambient noise is available. A lab has calibrated the device: an approximate relationship between the voltage -volts- it delivers and the ambient noise measured in dB is:

$A_n(db) = (e^{V(volt)} - 0.9) * 4,2$

Find what modifications (or additions) are necessary to adapt the "EIDEAnalog" library to this device.

(Tip: There are several possibilities one of them being to devise a brand new calculation agent for this purpose (decibelsAgent? In this case, a “spurious” use of the general parameters of the classes is proposed, for example, *Vref* for the value “0.9”, *gain* for “4.2”).

5.4.- Class *sensor* code. Instantiation

Once set the classes for the buses (paragraph 4) and the calculation agent classes puzzle (paragraph 5.3) solved, it is time to frame everything with the sensor class so that the whole set is functional.

The sensor class is the center of the library: when instantiating each sensor its type and the bus to which it is connected are passed as arguments, the latter as an object of the ADCBus class that, of course, must have been previously instantiated. With the bus attributes and the sensor data all the attributes of the -new- sensor class object itself are complete: these are used in turn to create on the fly other two instances 1) one of the *sensorTable* classes -if applicable; tabulated sensors- and 2) the corresponding *calculation agent* -this is always applicable- and that are linked to the *sensor class object* as attributes. The following figure is an outline of the above.

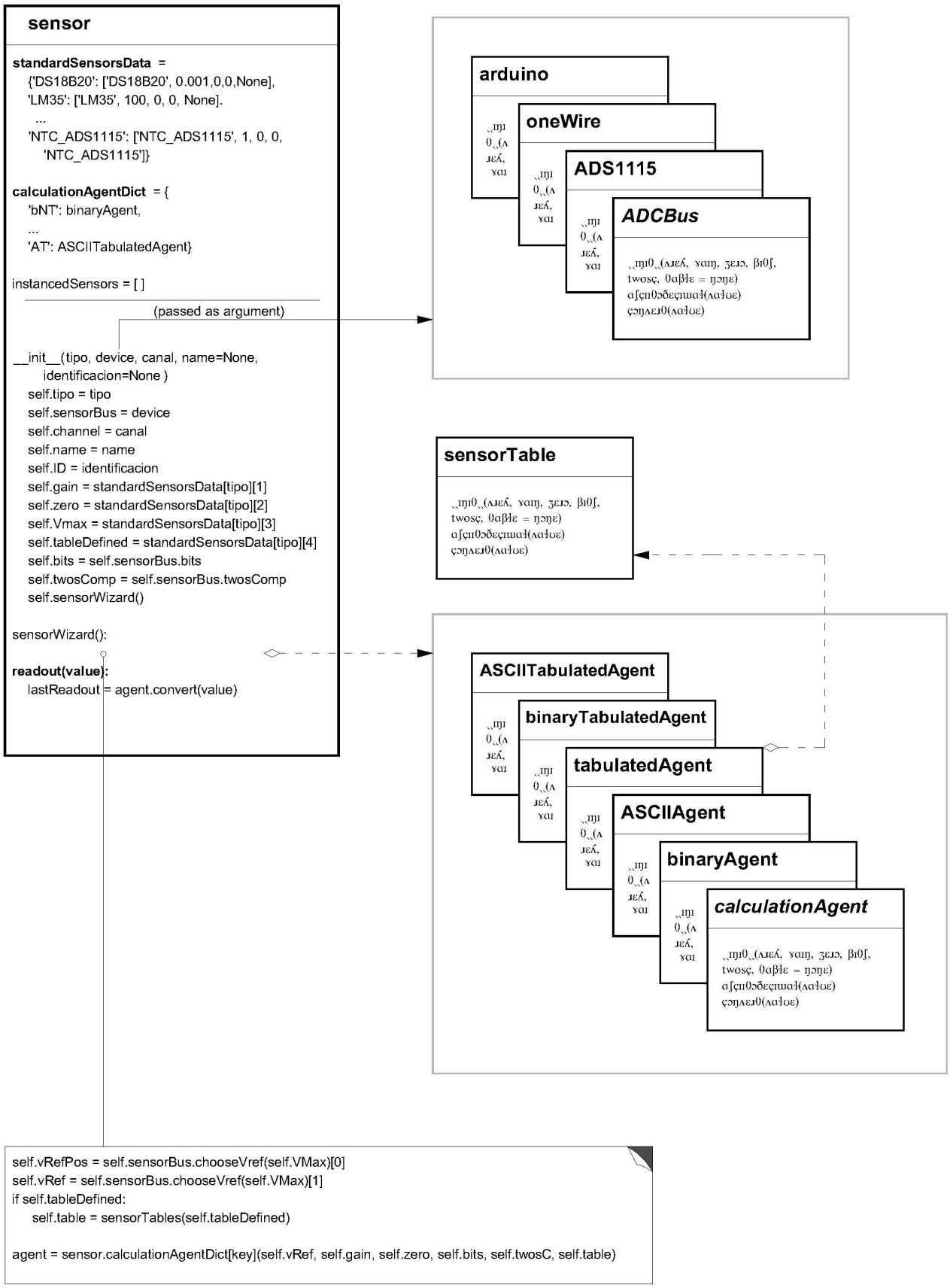


Figure 25. Class sensor outline.

May be that it is not easy to understand at the drop of a hat so, even at the risk of repeating things, we are going to discuss it a little more:

- Note that the *sensor class* -its objects- do not represent a specific sensor *type*, but a specific sensor connected to a specific ADCBus. There is no a *sensor class* that, instantiated, represents a specific sensor type; in this library this job is delegated to the *standardSensorsData* dictionary (see a little below), which is an attribute of the *sensor class*. It seems like a tongue twister (and, in a way, it is), but until you fully understand it, it is much better not to continue.

- To instantiate a *sensor*, at least one *bus* must have previously instantiated; this instance has to be passed as an argument when instantiating the sensor: some characteristics of the bus -data format- to which the sensor is connected are necessary so that it -the sensor object- can in turn instantiate the *calculation agent* it will need.

- When instancing a sensor, the object being created in turn creates an instance of the *sensorTable* class in case a table is associated with the sensor type. For example the 'ADS1115_NTC' sensor has an associated the table ADS1115_NTC; the ‘LM35’ has no associated table. The *sensorTable* instance becomes one more attribute of the object being instantiated (another tongue twister). This kind of collaboration between objects is called *aggregation*; you may forget about it to continue. The concept itself may -just- remain as an intuitive idea: an object of one class can be helped by the skills of objects of a different class.

- In a somewhat more elaborate way, the object of the *sensor class* also instantiates an object of one of the *calculation agents* in paragraph 5.3, the one it -the sensor class object- needs depending on its type -sensor- and the bus to which it is connected. To do this -to select the right calculation agent- it needs the information in the *calculationAgentDict* dictionary which is an attribute of the sensor class (as the *standardSensorsData* dictionary above). This is also an aggregation (instantiating a particular *calculation agent* class).

- Collaboration in between the object of the ADCBus class that has been passed as an argument to instantiate the sensor and its instance –the one of the sensor- is functionally identical to those mentioned for the *(sensor)table* and the *calculation agent*, although its implementation -it is passed as an argument to instantiate the sensor- is a little unorthodox. As the methods and attributes of the ADCBus object are available to the sensor object, it is also an aggregation, although, unlike the previous two, the ADCBus object is not exclusive to the sensor object that has been passed as an argument: It will be shared by all the sensors it has connected (the ADCBus).

Let's see what the sensor class code is:

```
class sensor():
    """ Sensor information and readout calculation """

    standardSensorsData = {
        'DS18B20':      ['DS18B20', 0.001, 0, 0, None],
        'LM35':         ['LM35', 100, 0, 1.0, None],
        'LM50':         ['LM50', 100, 0.5, 1.0, None],
        'rawVoltage' :  ['rawVoltage', 1, 0, 5.0, None],
        'ADS1115_NTC':  ['ADS1115_NTC', 1, 0, 3.3, 'ADS1115_NTC'],
        }
    # Agent selection dictionary
    calculationAgentDict = {
        'bNT': binaryAgent,         # Binary; not table
        'bT': binaryTabulatedAgent, # Binary; table
        'ANT': ASCIIAgent,          # ASCII; not table
        'AT': ASCIITabulatedAgent,  # ASCII; table
        }

    instancedSensors = []

    def __init__(self, tipo, device, canal, name=None, identificacion=None):

        self.name = name
        self.ID = identificacion
        self.tipo = tipo
        self.sensorBus = device
        self.channel = canal
        self.gain = float(sensor.standardSensorsData[tipo][1])
        self.zero = float(sensor.standardSensorsData[tipo][2])
        self.VMax = float(sensor.standardSensorsData[tipo][3])
        self.tableDefined = sensor.standardSensorsData[tipo][4]
        self.bits = self.sensorBus.bits
        self.twosC = self.sensorBus.twosC
        self.sensorWizard()

        self.lastReadout = 0

    def sensorWizard(self):
        """ Assign sensor other data """
        # Search for reference voltage and index it.
        self.vRefPos = self.sensorBus.chooseVref(self.VMax)[0]
        self.vRef = self.sensorBus.chooseVref(self.VMax)[1]
        # Instantiate table (if any).
        if self.tableDefined:
            self.table = sensorTable(self.tableDefined)
        else:
            self.table = None
        # Calculate and instance calculation agent
        key = ''
        if self.bits > 0: key = key + 'b'   # binary
        else: key = key + 'A'               # ASCII
```

```
        if self.table: key = key + 'T'      # Tabulated
        else: key = key + 'NT'              # Not table

        self.agent = sensor.calculationAgentDict[key](
            self.vRef, self.gain, self.zero, self.bits,
            self.twosC, self.table)

    def readout(self, value):
        """  """
        self.lastReadout = self.agent.convert(value)
        return self.lastReadout
```

Class *sensor*. Code analysis.

```
class sensor():
    """ Sensor information and readout calculation """
```

Class has no *ancestors*.

```
    standardSensorsData = {
        'DS18B20':      ['DS18B20', 0.001, 0, 0, None],
        'LM35':         ['LM35', 100, 0, 1.0, None],
        'LM50':         ['LM50', 100, 0.5, 1.0, None],
        'rawVoltage' :  ['rawVoltage', 1, 0, 5.0, None],
        'ADS1115_NTC':  ['ADS1115_NTC', 1, 0, 3.3, 'ADS1115_NTC'],
        }
```

We use the *class attributes section* of the sensor class to house the dictionary with the basic data of the sensors that the project has -or may have- connected. This will be one of the section in the library that will need to be modified to add new sensors; successive dictionary entries contain this information: *sensor type, gain, zero, reference voltage, and table;* the latter being the name of the file holding the table data (See 7, “Sensor tables [sensorTable]”).

May be that two identical sensors are used with different characteristics: an example would be two tabulated sensors physically identical that use a different table; just define them with two different names in sensor.standardSensorsData. The same happens if, for example, the reference voltage is different: it is necessary to include two different sensors in the dictionary.

```
    # Agent selection dictionary
    calculationAgentDict = {
        'bNT': binaryAgent,         # Binary; not table
        'bT': binaryTabulatedAgent, # Binary; table
        'ANT': ASCIIAgent,          # ASCII; not table
```

```
        'AT': ASCIITabulatedAgent,  # ASCII; table
        }
```

This dictionary is also included as a *class attribute*; it indexes the *calculation agent* classes (see 5.3, “Readout calculation algorithms. *agent* classes”). It is used, as explained below -*sensorWizard*-, to instantiate the *calculation agent* required by the sensor.

```
InstancedSensors = []
```

*InstancedSensors i*s a python list to keep the instantiated sensors -the objects-. It will be -optionally- used by the library *client classes*. You can ignore it for now.

```
    def __init__(self, tipo, device, canal, name=None, identificacion=None):

        self.name = name
        self.ID = identificacion
        self.tipo = tipo
        self.sensorBus = device
        self.channel = canal
        self.gain = float(sensor.standardSensorsData[tipo][1])
        self.zero = float(sensor.standardSensorsData[tipo][2])
        self.VMax = float(sensor.standardSensorsData[tipo][3])
        self.tableDefined = sensor.standardSensorsData[tipo][4]
        self.bits = self.sensorBus.bits
        self.twosC = self.sensorBus.twosC
        self.sensorWizard()

        self.lastReadout = 0
```

This *__init__* method is one of the most important parts of the library. In addition to 1) initializing the attributes of the class necessary to identify the sensor, the ADCBus to which it is connected and to which channel, (“self.name =” ... “self.tableDefined =”), 2) extracts from the bus object the required parameters (bits, twosC) to later instantiate the *calculation agent* that this instance of the *sensor class* needs.

The *calculation agent* thus bind to the *sensor class object* that is being initialized remains, let's say, "property" of the sensor (remember that it is about a specific sensor of type *self.type* connected to the *self.channel* channel of the ADCBus whose object has been passed as an argument to instantiate the sensor -*device*).

Let’s come back to it (at the cost of repeating concepts once again):

1. Each *sensor class object* (it has been already noticed that the class name is not the best; perhaps "*sensorChannel*" would have been better) contains -it must contain- the right values for all its attributes, so that it can instantiate the *calculation agent* it needs and with the correct arguments. Thus, as we will see below, the mere call to the *convert* method of the associated *calculation agent* (with the argument of the ADC reading) returns the sensor *readout*.

2. The sensor class object delegates the calculation of the readout to the object of the *calculation agent* class (*convert* method). Hence, the objects of the calculation agent class are so verbose while the code of the sensor class is small (although dense).

3. The library pattern requires that before instantiating the sensors at least one ADCBus -object- exists (the sensor does not connect directly to the Raspberry, but through a bus).

4. At the instantiation time *the client* passes the sensor basic information (*type, bus, channel, name* and *identification*; the *name* is optional and the *identification* is only necessary for the oneWire bus sensors); the remaining -of the information necessary to calculate the readout (which, we emphasize, is almost the only thing that the -objects of the- sensor class does)- comes from 1) the sensor dictionary (*sensor.standardSensorsData*) by means of the type (*self.type*) and 2) the *ADCBus object* that is passed as an argument.

In order to fix ideas, let's see an instantiation example:

```
hW = sensor('LM35', myBus, 4, name='Hot water')
```

This code line instantiates the class sensor and creates an object named *hW* (the object; the sensor itself is named "*Hot water*"). Sensor type is *LM35* and it is connected to channel 4 of the ADCBus *myBus*.

The object *__init__* method extracts the data format and the reference voltage from the object of the ADCBus class -*myBus*- and adds the sensor parameters ("LM35" entry on *standardSensorsData* dictionary) to these. With all this information it -the sensor object- instantiates the agent -*self.sensorWizard* below- that it needs for the readout calculation (and which it will use every time it is asked to calculate it -the readout).

```
def sensorWizard(self):
        """ Assign sensor other data """
        # Search for reference voltage and index it.
        self.vRefPos = self.sensorBus.chooseVref(self.VMax)[0]
        self.vRef = self.sensorBus.chooseVref(self.VMax)[1]
        # Instantiate table (if any).
        if self.tableDefined:
            self.table = sensorTable(self.tableDefined)
        else:
            self.table = None
        # Calculate and instance calculation agent
        key = ''
        if self.bits > 0: key = key + 'b'   # binary
        else: key = key + 'A'               # ASCII
        if self.table: key = key + 'T'      # Tabulated
```

```
        else: key = key + 'NT'              # Not table

        self.agent = sensor.calculationAgentDict[key](
            self.vRef, self.gain, self.zero, self.bits,
            self.twosC, self.table)
```

This method is actually the continuation of *__init__*; it has been detached to improve readability:

1. It indexes (locates) into the bus *refVoltageList* the sensor *top of scale* (see 4.1.9, “chooseVref method [reference voltage]).

2. It assigns to the *self.table* attribute the *sensorTable* object in case it -the table- exists.

3. It finally determines which class of the *calculation agent* family corresponds to the sensor according to its characteristics and those of the bus to which it is connected. It instantiates and assigns it to the attribute *self.agent.*

```
    def readout(self, value):
        """  """
        self.lastReadout = self.agent.convert(value)
        return self.lastReadout
```

This method -*readout*- synthesizes all the above. As an advantage from the complexity introduced in the elaboration of the *calculation agent* classes (and in the *__init__* method of the sensor class), it elegantly solves the problem of calculating the readout (which corresponds to such sensor connected to such ADC) with a single line of code.

5.5.- *sensor* Class. Conclusion.

- Unlike the bus control classes, which encapsulate the code necessary for their use (therefore being directly instantiable and operational in themselves), the sensor class -its objects- manages the operation of the ADCBus classes and those of the calculation agents. This class is not useful -not even instantiable- without the prior instantiation of a bus and the existence of, at least, one *calculation agent*.

- The *__init__* method (the *sensorWizard* method included) links the operation of the other library classes. It is a good example on a *constructor* method having an internal logic that makes the rest of the code much simpler.

- To assign the *calculation agent* to the sensor a dictionary -*calculationAgentDict*- and a piece of code -*sensorWizard*- are used; it uses the sequence "*if then else*" a couple of times. This is not the part of the library that best meets OOP standards. The logic contained in *sensorWizard* should not be implemented like this, but as a class method that would contain the *calculationAgentDict* dictionary as a class attribute. This method, on the other hand, would be instantiated only once, since it only implements the allocation algorithm. Although this is the orthodox procedure, it gives rise to some intricate code that has been avoided.

The ADS1115 class, the calculation agent classes family and the sensor class presented -and fully explained- up to here form a complete and operational (sub)library. Should you just intend using an ADS1115 with conventional (or not so conventional; without *table*, any case) sensors, you can use just this part of the library and discard the remaining. Any case we strongly recommend reading chapter 8, “Library usage”.
