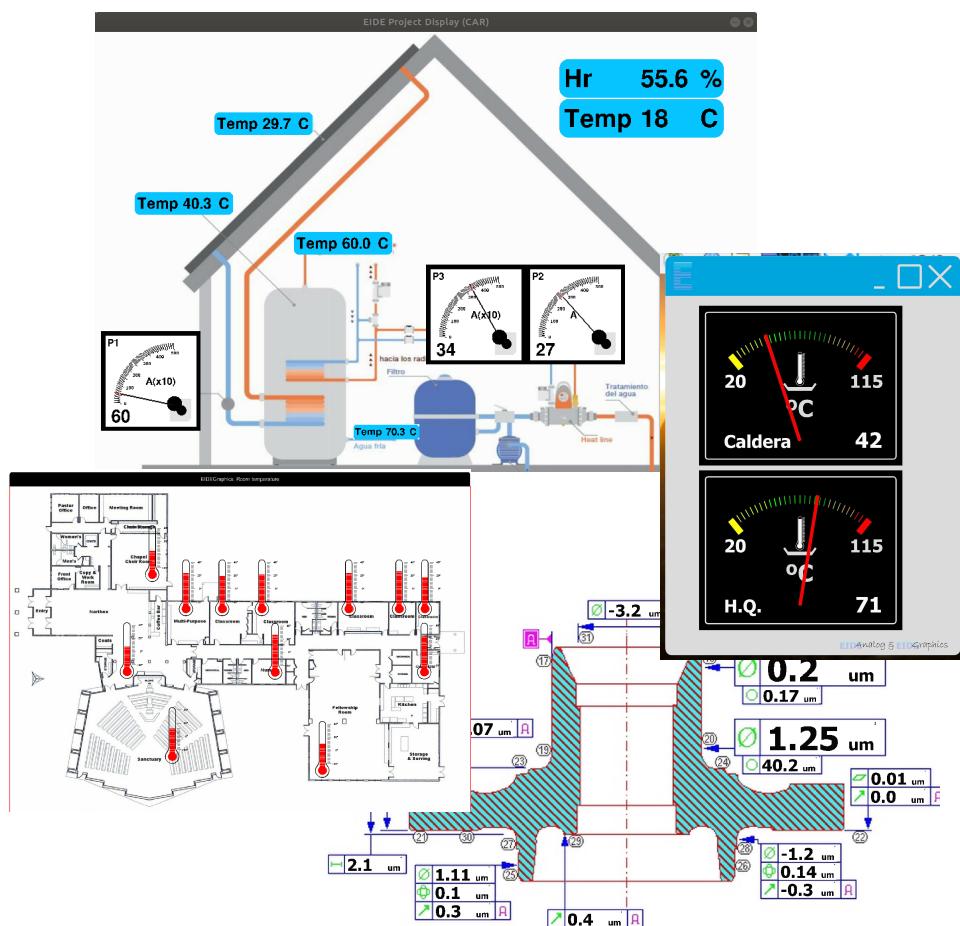


Captura de parámetros físicos con Python y Raspberry Pi



Librería *EIDEAnalog*
(Así se hizo)

Tabla de revisiones

Nº	Fecha	Párrafo	Descripción
0	27/04/2020	-	Primera edición

Table of Contents

1.- Presentación (breve).....	5
1.1.- Objetivo del blog.....	6
1.1.1.- Arquitectura de aplicaciones en OOP.....	7
1.1.2.- Sistemas de adquisición de datos. Conversión A/D.....	7
1.1.3.- Desarrollo del blog. Pedagogía.....	8
1.1.4.- Conclusión. Objetivos.....	8
1.2.- Conocimientos -previos- necesarios para seguir el blog.....	8
1.2.1.- Electrónica básica. Conversor A/D. Sistema de numeración en base 2.....	9
1.2.2.- Raspberry.....	10
1.2.3.- Linux.....	12
1.2.4.- Python. OOP.....	14
1.2.5.- Otros conocimientos previos necesarios.....	15
2.- Proyecto: librería ADC. Especificación y herramientas.....	16
2.1.- Especificación.....	16
2.2.- Raspberry.....	16
2.3.- Conversores ADC.....	18
2.3.1.- Bus oneWire.....	18
2.3.2.- ADS1115.....	20
2.3.3.- Arduino ProMini.....	21
2.4.- Lenguaje de programación. Python. IDE.....	24
2.5.- Análisis (y programación) OOP.....	25
2.5.1.- Generalidades.....	25
2.5.2.- Ejemplo.....	26
2.5.3.- OOP. Conclusión.....	27
3.- Análisis preliminar de la arquitectura de la librería.....	29
3.1.- Análisis preliminar “nombre/verbo”.....	29
3.2.- Las clases como proveedoras de servicios.....	30
3.2.1.- Clases para buses. Clases para sensores. Cálculo.....	30
3.2.2.- método <i>readout</i>	31
3.2.3.- Clases agentes de cálculo.....	33
3.3.- Esquema de clases de la librería.....	34
4.- Clase ADS1115.....	35
4.1.- Desarrollo.....	37
4.1.1.- Orden al ADS de inicio de conversión.....	37
4.1.2.- Comprobación de fin de conversión.....	41
4.1.3.- Lectura de la conversión.....	41
4.1.4.- Atributos de clase.....	42
4.1.5.- Método <i>__init__()</i> . Atributos de la instancia.....	42
4.1.6.- Código completo de la versión mínima de la clase ADS1115.....	44
4.1.7.- Instanciación (uso) de la clase ADS1115.....	45
4.1.8.- ADS1115. Métodos para su configuración.....	46
4.1.9.- método <i>chooseVref</i> (tensión de referencia). Método <i>setChannelGain</i>	50
4.1.10.- Clase ADS1115 completa. Ejemplo de uso.....	51
4.1.11.- Clase ADS1115. Conclusión.....	52
5.- Clases para sensores. agentes de cálculo.....	54

5.1.- Sensores.....	54
5.2.- Clase sensor. Preliminar.....	56
5.3.- Algoritmos de cálculo de la lectura. Clases “agent”.....	58
5.3.1.- Agente de calculo “binario” (binaryAgent). Herencia.....	59
5.3.2.- Agente de calculo “ASCII” (ASCIIAgent).....	62
5.3.3.- Clase <i>calculationAgent</i>	63
5.3.4.- Agente de cálculo “tabulado” (tabulatedAgent).....	66
5.3.5.- Herencia múltiple. Clases “binaryTabulatedAgent” y “ASCIITabulatedAgent”.	67
5.3.6.- Familia de clases agente de cálculo. Resumen. Conclusiones.....	70
5.4.- Código de la clase sensor. Instanciación.....	73
5.4.- Clase sensor. Conclusiones.....	81
6.- Clases para buses <i>oneWire</i> y arduino. Clase ADCBus.....	82
6.1.- Clase <i>oneWire</i>	82
6.2.- Clase arduino.....	90
6.3.- Clase ADCBus.....	94
6.4.- Clases para buses. Diagrama. Conclusión.....	97
7.- Tablas de sensores (<i>sensorTable</i>).....	99
7.1.- Tablas de sensores. Conclusiones.....	107
8.- Uso de la librería.....	109
8.1.- Uso básico.....	109
8.1.1.- Instanciación directa de buses y sensores.....	110
8.1.2.- Uso con <i>lista de sensores</i>	114
8.2.- Uso con clases intermedias. Clase “ADCBus_manager”.....	115
8.3.- Uso de la librería EIDEAnalog. Conclusiones.....	120
9.- Conclusión. Continuidad.....	121
Anexo I.- Descarga de EIDEAnalog.....	123
Anexo II.- Configuración de Raspbian.....	125
II.1.- Instalación básica. <i>Distribuciones</i>	125
II.2.- Instalación de <i>Raspbian</i>	125
II.3.- Personalización de <i>Raspbian</i>	126
II.3.1.- Bus i2c.....	126
II.3.2.- Bus <i>oneWire</i>	130
II.3.3.- Conexión serie.....	131
II.4.- Python.....	131
II.5.- µSD preconfigurada.....	132
Anexo III.- Uso conjunto con ‘EIDEGraphics’.....	133
III.1- Descarga e instalación de EIDEGraphics.....	134
III.2- Funcionamiento con EIDEAnalog.....	137
Referencias.....	139

1.- Presentación (breve)

Trataremos en este ‘blog’ de explicar como se ha desarrollado y puesto a punto una ‘aplicación’ (o ‘programa’, o ‘librería’ -que es lo que es en realidad-) para captura de parámetros físicos (conversión analógico digital) empleando Python como lenguaje de programación, y OOP o Programación Orientada a Objetos. Como hardware se emplean -sucesivamente- A) una tarjeta Raspberry Pi sin hardware añadido para el bus oneWire, B) la Raspberry más un conversor analógico digital tipo ADS1115 y C) la Raspberry conectada a un Arduino ProMini; en el primer caso los sensores son del tipo DS18B20, para los otros dos se emplean media docena de sensores convencionales -temperatura, tensión- que se describen en su momento. Como se irá explicando en el blog, se hace mucho énfasis en la justificación de la arquitectura de (clases que componen) la librería y del desarrollo de las clases, cuestiones que habitualmente resultan las más complejas en este tipo de aplicaciones.

El blog puede ser de interés:

- Como guía de prácticas de programación en cursos de grado medio (formación profesional); es posible que incluso en ciertas especialidades de grados universitarios sirva para idéntica finalidad.
- Para aficionados a (o profesionales de) la programación que deseen mejorar o comprobar sus conocimientos de OOP.
- Profesionales en el campo de la ‘adquisición de datos’; aunque el hardware a emplear no tiene grado industrial (no, al menos, el que específicamente se ha empleado para elaborar este blog), no es descartable que pueda ser de su interés.

Se procura que las explicaciones sean todo lo concisas que permite el asunto, que, como se comenta más adelante, no es un problema simple.

(Si solamente quiere familiarizarse con el uso de la librería y no con su diseño puede ir directamente al punto 8, “Uso de la librería”).

1.1.- Objetivo del blog.

El objetivo declarado del blog es, pues, la elaboración de una -pequeña- librería escrita en Python para facilitar el uso de varios tipos de adaptadores AD, en principio un ADS1115, el bus “oneWire” y un Arduino Promini. El primero y el ultimo son añadidos de hardware al Raspberry, mientras que el bus oneWire es nativo de las placas Raspberry: cargado el correspondiente driver se pueden habilitar los pines del conector GPIO que se deseen para implementarlo.

Configuración del Raspberry: Dependiendo de la versión del Raspberry que tenga y de la “distribución” de Raspbian, tendrá su Raspberry configurado para el bus i2c (necesario para que funcione el ADS1115) y/o el oneWire (necesario para el bus del mismo nombre) y/o la comunicación serie (necesaria para el Arduino). Consulte el anexo II, “Configuración de Raspbian” para más detalles.

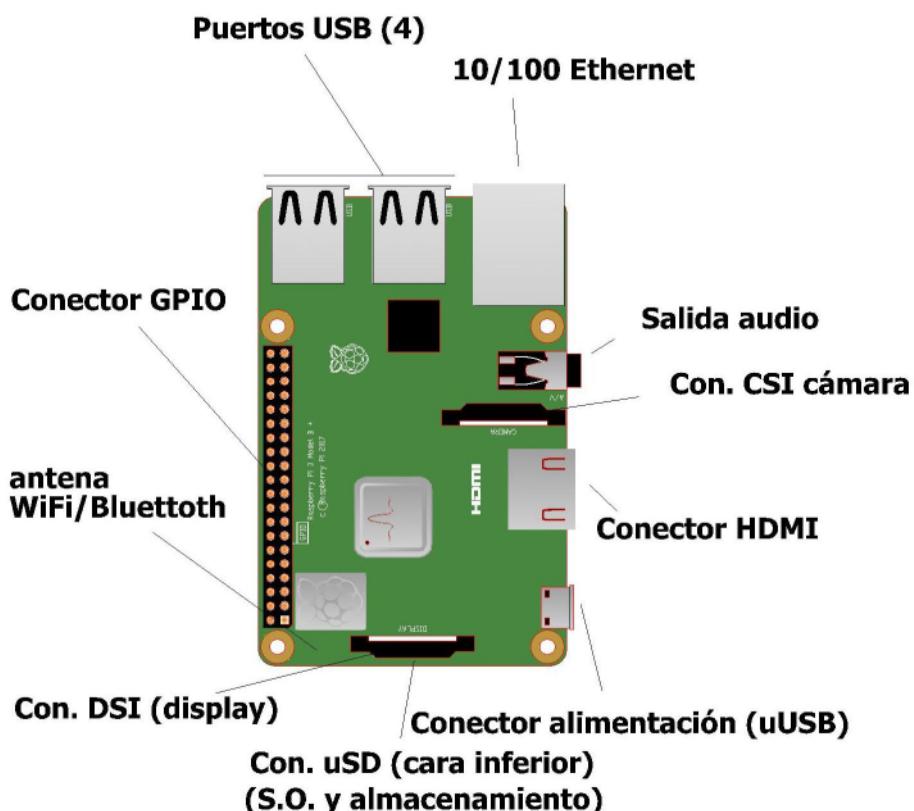


Figura 1. Raspberry 3B +

Y, sin embargo, el blog no se limita a esto: aunque efectivamente se desarrolle -se escriba- la librería tal y como se ha descrito en el párrafo anterior, hay otro objetivo tan importante como el anterior (o más), al menos para quién desee aprender algo de análisis y programación OOP. Esperemos que el viaje, con el análisis de necesidades, la descomposición del problema en clases, el estudio de la arquitectura de la librería, sea tan provechoso como el destino del mismo, que no es otro que el mero uso de la misma.

1.1.1.- Arquitectura de aplicaciones en OOP.

Para centrar la cuestión: al abordar el análisis de un desarrollo con OOP, uno de los problemas centrales es la arquitectura del sistema; i.e.: ¿Cómo encuentro las *clases* -que, después, se instanciarán para dar lugar a los *objetos*- que forman mi aplicación?.. .
¿Cuánto debo desmenuzar -*granularity*- el problema?. ¿En cuántas clases?. ¿Qué relación tienen entre ellas?. ¿Y los objetos entre sí?. ¿Cómo se documenta?.

El problema específico -la librería propiamente dicha- que se aborda en este blog es relativamente simple: se resuelve con una docena mal contada de clases (como veremos en su momento); su mejor calidad es que está desarrollado (que se dispone del código de la librería y que, además, funciona). El lector, al que en más de un momento es probable que le *tiemblen las piernas*, por lo menos tiene por seguro que llegará a buen puerto.

Es decir, que la mejor virtud de este blog es precisamente que se trata de un ejemplo completamente desarrollado, y no de describir las bases del análisis OOP, cuestión compleja y que requiere de bastante más que este blog. Si desea seguirlo -el blog- tenga en cuenta esto: se trata de un ejemplo desarrollado (programado, funcionando). Nada más (ni nada menos).

1.1.2.- Sistemas de adquisición de datos. Conversión A/D.

Si de lo que se trata es de hacer pedagogía de OOP podríamos haber escogido otra materia que no fuese la de conversión A/D (analógico a digital). ¿Qué tiene este campo que no tengan otros para haberlo elegido como hilo conductor del blog?.

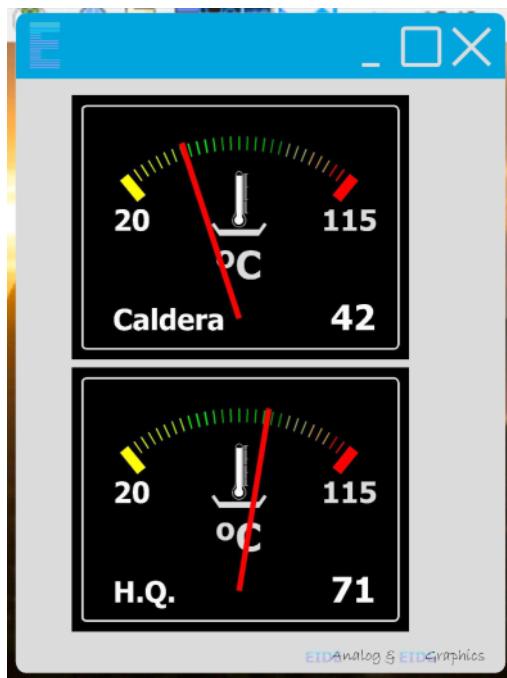


Figura 2. Sistema de adquisición de datos con Raspberry.

Que los autores -del blog- lo conocen bien (y son perfectamente conscientes de que la elección de una librería para “captura de parámetros físicos”, como se dice en la tercera línea del primer punto de este blog, le quitará una buena parte de seguidores que, de haber elegido otro tema, quizás se habrían animado a seguirlo).

Pero, por otra parte, el asunto tiene también sus atractivos, no siendo el menor de ellos que el resultado nos permitirá conectar sensores a un ordenador (cuestión que, en muchos sentidos, se puede considerar una excentricidad). Es este un asunto que interesa de forma transversal a un buen número de actividades, tanto en el ámbito de la industria como en el de la formación, y en este -el de la formación- a todas las ingenierías y grados en ciencias físicas o químicas. Al fin y al cabo, y salvo las interioridades de funcionamiento del dispositivo de conversión A/D propiamente dicho -que sí son específicas de esta disciplina-, el resto no es mucho más que álgebra y lógica binaria; una vez que se está conforme con la finalidad de lo que se está haciendo -la librería-, la forma de hacerlo -el análisis- sí que es válido para problemas similares, o de arquitectura similar.

1.1.3.- Desarrollo del blog. Pedagogía.

En lo que sí se ha sido muy cuidadoso es en el desarrollo del blog en sí mismo, adoptando un estilo fundamentalmente pedagógico. No se trata, solamente, de escribir unas clases para que con un “corta-pega” se incorporen a un proyecto (y, sin embargo, esto se puede hacer sin ningún problema -y funciona). El objetivo principal del blog es, como ya se ha dicho, ayudar a quien desee analizar un ejemplo de cómo encarar el análisis y desarrollo de una aplicación empleando las técnicas de la OOP.

1.1.4.- Conclusión. Objetivos.

Al acabar el blog (con aprovechamiento), usted habrá adquirido ciertas habilidades en lo referente a la OOP, vea 2.5.3, “OOP. Conclusión.”. Subsidiariamente, aprenderá:

- (Conceptos básicos) de adquisición de datos. ADC (conversor AD).
- Manejo de un Raspberry a nivel de usuario. Conceptos (muy) básicos de Linux.
- (Rudimentos de) funcionamiento de los buses oneWire, i2c y la comunicación serie (UART).
- Funcionamiento de algunos sensores comerciales (y *populares*).
- Termistores. NTC. Sensores tabulados.

1.2.- Conocimientos -previos- necesarios para seguir el blog.

Los autores del blog han sufrido en sus carnes, y en más de una ocasión, manuales que empezaban -entusiastamente- explicando que para usar el ratón de un ordenador hay que manejarlo con la mano, moverlo sobre la alfombrilla y pulsar el botón de la izquierda para seleccionar y el de la derecha para las opciones (y que se puede configurar al revés para zurdos); inevitablemente el manual -o lo que fuera- abandonaba este nivel de detalle al segundo párrafo para, lo que es peor, dar instrucciones indescifrables apenas tres párrafos más adelante (y que, cada una de ellas -de las instrucciones indescifrables-, habrían merecido por sí mismas un manual).

Procuraremos no caer en este error; sería de ilusos pensar que para seguir este blog no haga falta ningún conocimiento previo en las materias que se manejan. Quien quiera seguirlo deberá tener *conocimientos de usuario* (signifique esto lo que signifique; ustedes ya entienden) de informática, de ordenadores personales. Como, además, usamos un Raspberry como base, usted debería hacerse con el manejo de este (naturalmente, siempre que se haga referencia al Raspberry estamos incluyendo también el sistema operativo -Linux).



Figura 3. ADS1115 conectado al Raspberry.

Tampoco le tiene que hacer ascos a mancharse de grasa: elija la opción que elija para poner en práctica el contenido del blog tendrá que conectar algo al Raspberry -ver figura anterior- y apretar algún tornillo que otro (terminales). Tiene que tener el arrojo suficiente para conectar cosas entre sí: el Raspberry con el conversor A/D, los sensores al conversor, la fuente de alimentación del Raspberry (es como la de un teléfono móvil), ... nada excesivamente complicado.

Vayamos por partes.

1.2.1.- Electrónica básica. Conversor A/D. Sistema de numeración en base 2.

Conviene que sepa manejar un polímetro (el *tester* de toda la vida); casi vale con que sepa medir tensiones en II (corriente continua). Repase la ley de Ohm, que no le vendrá mal (aunque para el blog no se necesite).



Figura 4. Polímetro (tester).

Lo de la conversión A/D es un poco más peliagudo.

Si se trata, como es muy habitual, de capturar valores *deprisa*, el asunto de la conversión A/D tiene esquinas por todos los lados. *Deprisa* es miles de veces por segundo -o más; o mucho más-, lo que es necesario, por ejemplo, en un estudio de audio para capturar la música sin que pierda calidad (44.100 veces por segundo); parecidas velocidades se precisan en muchísimos problemas en la industria de todo tipo, en investigación, telecomunicaciones, imagen (en este caso mucho más).

No es el caso de este blog; aquí basta con entender que se trata de convertir valores del mundo real a números con mucha más calma. Si bien habrá que profundizar en cómo se controla el conversor A/D, este es un problema más *administrativo* que de ingeniería, como veremos cuando toque.

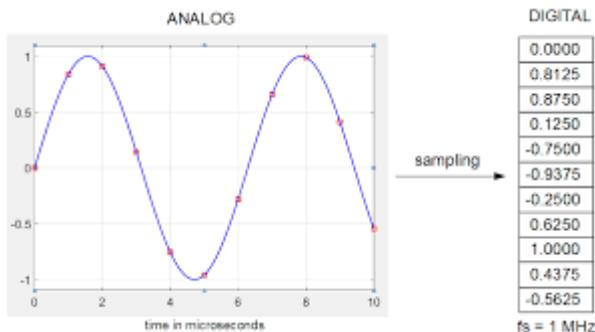


Figura 5. Conversión A/D.

(Como dice alguna página WEB, usted sentado al lado de un termómetro de los de columna de mercurio tomando notas cada, digamos, 10 segundos, es un conversor analógico digital: está transformando -y registrando- valores analógicos a números)

El asunto es que el ordenador se maneja con el sistema de numeración en base 2 (los famosos 1010001010 ...), que, en este blog, se convierte en el principal problema científico (al menos diferente del de fondo: el análisis en OOP). Si usted ya se maneja con este asunto enhorabuena, pase al siguiente punto; si no es así se tiene que imponer en ello: las páginas de la wikipedia de las entradas “binario” y “bitwise” son un buen comienzo. Tiene que saber, también, lo que es el formato “en complemento a 2”.

Se usa el término *word* -la traducción literal al castellano, “palabra” se usa muy poco con el mismo sentido- para designar conjuntos de 16 bits. Aunque el término se puede usar para cualquier longitud -de bits; normalmente potencias de “2”- su uso sin especificar más suele referirse a 16 bits.

Aunque de forma tangencial, también se maneja el sistema de numeración hexadecimal; échale un vistazo en la wikipedia.

1.2.2.- Raspberry.

Prácticamente todo lo que necesita saber usted del hardware del Raspberry, al menos para empezar, es 1) cómo conectarlo y 2) cómo configurar la µSD (microSD). Hay, literalmente, miles de WEBS en las que explican como conectarlo; incluso la documentación -en papel- que lo acompaña lo incluye.

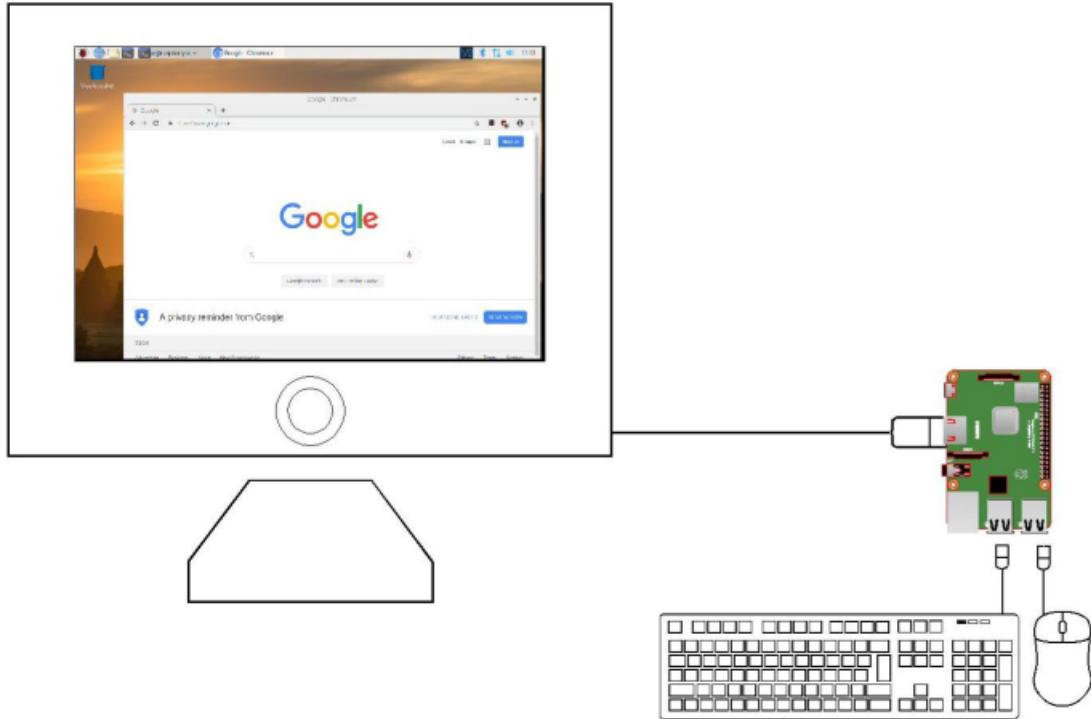


Figura 6. Raspberry conectado como ordenador de sobremesa.

(La forma fácil de ponerlo en marcha es conectar un monitor HDMI al puerto de video, un teclado y un ratón USB a dos de los 4 puertos USB que tiene, insertar una tarjeta de memoria µSD debidamente configurada y darle corriente a todo).

Lo de la µSD tiene un poco más de miga: vea el anexo II, “Configuración de Raspbian” para más información.

Aunque a medida que vayamos presentando las diferentes opciones se irá explicando con cierto detalle, no viene mal que se vaya usted enterando de lo que es un *bus oneWire*, el *i2c* y la comunicación serie. Vaya echando un vistazo a las entradas de la wikipedia para “oneWire”, “i2c” y “Comunicación serie”: que le vayan sonando por lo menos.

1.2.3.- Linux.

El Raspberry tiene la enorme ventaja de funcionar con Linux ... si usted no conoce Linux usted se lo pierde, pero no se asuste, no será necesario convertirse en un experto para poder usarlo, y con la interfaz gráfica resultará sencillo.

(Perdón por la impertinencia: es fruto de los muchos disgustos que *el otro* sistema operativo le ha dado a los autores del blog).

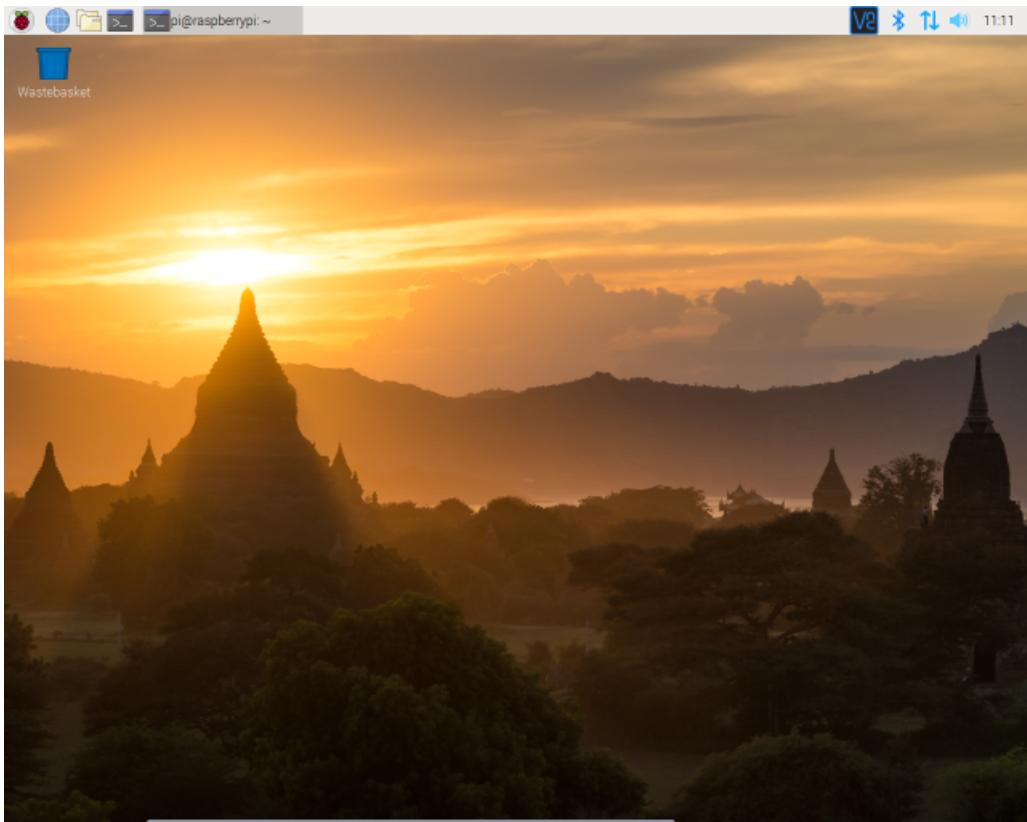


Figura 7. Interfaz de Linux (Raspbian).

Para seguir el blog no hay que saber mucho de Linux: si usted dispone de una *distribución* medianamente bien configurada, los mentados *conocimientos básicos de usuario* deberían ser suficientes para empezar (ver anexo II, “Configuración de Raspbian”).

Como en el caso anterior, si usted ya se maneja con Linux enhorabuena, pase al siguiente punto.

En caso contrario, le recomendaríamos que aprenda algo, digamos:

- Le conviene leer algo para, al menos, familiarizarse con los términos (“*Debian*”, “*Jessie*”, “*Raspbian*”, “*Noobs*”, “*Ubuntu*”, ...; no se alarme, son, en gran medida, los mismos perros con diferentes collares). Lea el artículo -en castellano- de la wikipedia para “*Raspbian*” (sea paciente: no entenderá muchas cosas al principio, pero hay que ir haciéndose con la música); la página WEB de Raspberry

(“www.raspberry.org”) no es lo mejor del mundo, pero tampoco pasa nada por echarle un vistazo.

- “Terminal”: el término hace referencia al uso del sistema operativo (Linux, en este caso, aunque se puede aplicar a cualquier S.O.) mediante una “línea de comandos”. Los más viejos del lugar recordarán la pantalla negra del MS-DOS anterior al Windows (y del que -del MS/DOS- los autores del blog tienen la misma opinión que del Windows propiamente dicho, o sea, mala): eso es el “terminal” (y, por cierto, bastantes “comandos” del MS-DOS están copiados de Linux; son los mismos).

Es conveniente que se maneje con un mínimo de comandos para configurar cosas del Raspberry (de nuevo se remite al lector al anexo II, “Configuración de Raspbian”). Consulte “<https://www.linuxadictos.com/5-comandos-imprescindibles-en-linux>”, “<https://franciscomoya.gitbooks.io/taller-de-raspberry-pi/content/es/intro/gnu.html>” y “recursostic.educacion.es/observatorio/web/ca/software/software-general/295-jose-ignacio-lopez” (página WEB que es buena y del ministerio de educación y ciencia ... tiene su aquel). Es muy bueno el libro “The Linux Command Line: A Complete Introduction. William E. Shotts Jr”.

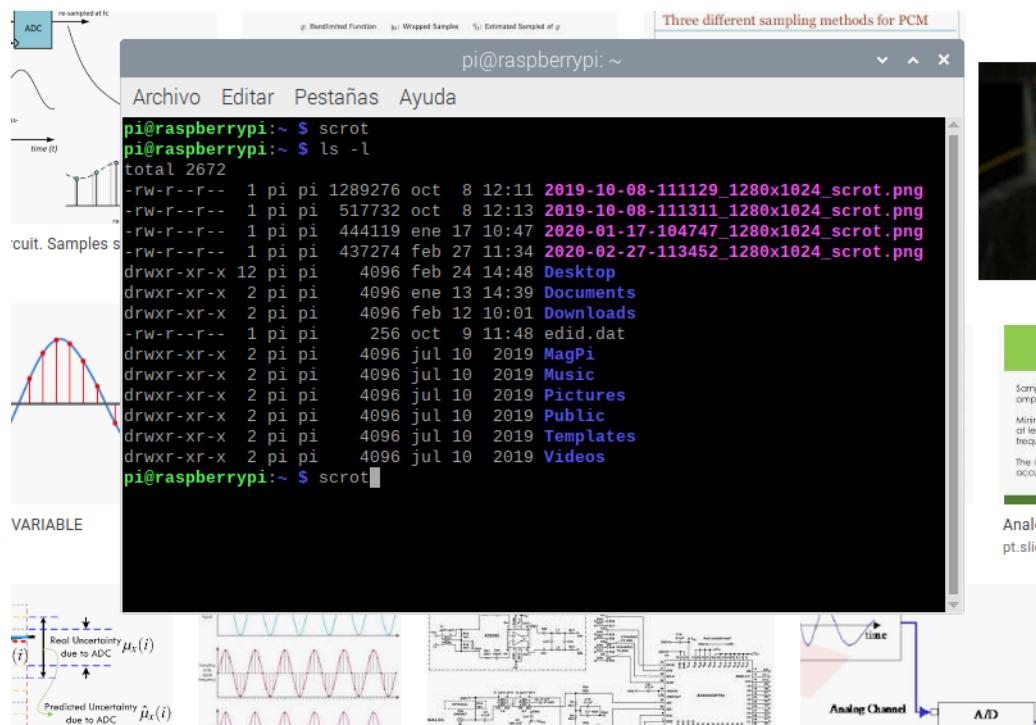


Figura 7. Terminal de Linux (Raspbian).

Copia de seguridad: Una de las ventajas fundamentales de Linux, al menos a juicio de los autores del blog, es la posibilidad de recuperarse del desastre de la pérdida del S.O. propiamente dicho.

! Usando otro sistema operativo la avería del disco duro le deja a usted a la intemperie. En el caso del Raspberry el equivalente sería la avería de la μSD (o del propio Raspberry, puestos a imaginar desastres); la buena -excelente- noticia es que es posible tener una copia de la μSD simplemente con otra μSD (en la que, eso sí, se tiene usted que tomar la molestia de copiar la buena periódicamente). Si va a *cacharrar* con el Raspberry, cosa que le recomendamos, es conveniente que tenga la copia de seguridad actualizada; o, simplemente, dos versiones del S.O., o sea, dos -o más- μSD con diferentes *situaciones* de su trabajo: son unos pocos euros a cambio de una tranquilidad impagable.

1.2.4.- Python. OOP.

Este blog *no* es un *tutorial* de Python (ni siquiera de OOP). Para seguirlo con cierta continuidad -sin tener que volver con frecuencia sobre cuestiones más básicas que las que se requieren para comprender lo que se va desarrollando- es necesario tener unos conocimientos *medios* de Python. Y tiene que haber escrito media docena de programas sencillos en Python.

(El término “*medios*” del párrafo anterior está utilizado con toda la intención -*mala leche*, si se permite la expresión-, porque, ¿quién es capaz de decir qué son unos conocimientos *medios* de Python?. Uno de los autores del blog lleva cuatro años usando Python, es un analista/programador con cierta experiencia, y, a día de hoy, no hay jornada de trabajo de más de cuatro o cinco horas en la que, desarrollando este blog o cualquier otra tarea que requiera del conocimiento previo de Python, no descubra algo nuevo relacionado con el uso del lenguaje -o, al menos, de los fundamentos de la OOP-.

Python es un lenguaje de los más fáciles (¿el más fácil?) de usar, y, al mismo tiempo, extraordinariamente potente. Y no se trata solo de que su sintaxis permita goyerías que resuman en una instrucción de 30 o 40 caracteres lo que con otros lenguajes requiere de varias líneas: Python tiene implementadas como básicas todas las instrucciones y arquitectura de cualquier lenguaje moderno, y eso sin contar lo que añaden las librerías que incluye la descarga normal. Además incorpora estructuras -los *diccionarios*, por ejemplo- y/o innovaciones -los *atributos de clase*, por ejemplo- que en su momento, y todavía actualmente, lo diferencian del entorno.

Es, en resumen, un mundo. Así que a ver quién es el guapo que acota los conocimientos *medios* de Python.

Procederemos, entonces, por extensión: tomemos como guía de la propia WEB de Python el *tutorial* de referencia (“<https://docs.Python.org/3/tutorial>”). Hay que saber:

- Definir y manejar variables numéricas, sus tipos -“*int(eger)*”, “*float*”, ... y formatos -decimal, hexadecimal, binario, Operaciones aritméticas y lógicas -todas-. Entienda bien qué es “*None*”.
- Definir y manejar *strings* (aquí sí, lo básico: qué es un *string* y poco más).
- “Control de flujo”. El mentado tutorial de Python no es de lo mejor de la *fundación* (“Python Software Foundation”): la primera mención que se hace a este respecto -control de flujo de un programa- en el punto 3.2 es francamente desafortunada; no se acaba de entender por qué el -primer- ejemplo se implementa con un “*while*” sin ninguna anestesia previa. Para seguir el blog sin sobresaltos pase al punto “4”: tiene que entender todo lo que contienen los puntos 4.1 a 4.6, *todo*.
- *Listas, tuples y diccionarios*: a fondo. Conviene estar familiarizado con todo el contenido de los puntos 5.1.1, 5.1.2. y 5.5 del tutorial. En la librería se usan -mucho- para iterar: estudie a fondo el contenido del punto 5.6 y repase el 4.2.
- Módulos: es suficiente con entender para que sirve la instrucción “*import*” y lo que se entiende por “módulos estándar” (6.2)
- *Errores y excepciones*. Merecen un comentario aparte: la implementación de una librería que sea digna de tal nombre no debe ponerse a disposición de los posibles usuarios sin que tenga un mínimo de ‘protección’ frente a los errores que, con toda seguridad, se cometerán al utilizarla (y eso sin contar algunos que, sin duda, irán sibilinamente ocultos en el código de la propia librería). La contrapartida es que el

uso de cláusulas de tratamiento de errores introduce una nueva complicación en el ‘aspecto’ del código, que, al menos en primera lectura, causa el rechazo de no pocos novatos.

Ha habido, pues, que buscar un compromiso: la versión inicial de la librería (la que se comenta con profusión en este blog y que puede usted descargar desde <https://github.com/Clave-EIDEAnalog/EIDEAnalog>), no tiene tratamiento de errores. Está en preparación la versión ‘Pro’, que si lo contendrá. Échale un vistazo al capítulo 8 del tutorial, no obstante.

- Clases. Objetos: Aquí viene cuando la matan. Ni que decir tiene que los términos clase y objeto no tienen que tener secretos para usted; y mal empezamos, no obstante, si usted frunce el cejo en cuanto aparece el primer “self” o sale despavorido cuando ve un “`__init__`” en el código. En el blog ni lo damos todo por sabido (en lo referente a los conceptos de clase, objeto, etcétera) ni podemos hacer pedagogía al respecto con los conceptos básicos de la cuestión (que, en cambio sí se hace -la pedagogía- cuando la cosa se complica: abstracción, encapsulamiento, herencia, polimorfismo, arquitectura). Por otra parte, la calidad didáctica del capítulo 9 del mencionado tutorial (“<https://docs.Python.org/3/tutorial>”) deja, de nuevo, bastante que desear. Y no es este un asunto fácil (ni la cuestión básica de la técnica de OOP ni la de cómo empezar en ello); a continuación se recomiendan un par de sitios WEB que usted debe estudiar si quiere, como ya advertimos más arriba, seguir el blog sin sobresaltos.
 - <https://www.programiz.com/Python-programming>
 - <https://realpython.com>

 **Instalación y configuración de Python:** Si usted no está familiarizado con Linux es posible que la puesta en marcha de Python en el Raspberry entrañe cierta dificultad. Aunque los autores del blog son partidarios impenitentes de Linux -y, por tanto, detractores confesos del “otro” sistema operativo-, sería un exceso sugerir al seguidor del blog que lo estudie a fondo -Linux- como paso previo para empezar con el blog. La puesta en marcha de un Raspberry -y su uso- no es complicada en absoluto si se dispone de una buena *distribución* (es la jerga para referirse a la versión del S.O.). Hay multitud de vendedores en Internet que le pueden suministrar una µSD debidamente configurada; tiene un interfaz gráfico muy similar al de Windows(R). Consulte el anexo II, “Configuración de Raspbian” para más información.

1.2.5.- Otros conocimientos previos necesarios.

- Tipos de ficheros: debe ser usted capaz de identificar y modificar un fichero de texto (*notepad* de Windows; “*leafpad*” en Linux).
- Ni que decir tiene que tiene que saber lo que es el código ASCII.
- Échale un vistazo a la entrada de wikipedia para “termistor” y “NTC”.

2.- Proyecto: librería ADC. Especificación y herramientas.

Bien, si después de tan prolíjos prolegómenos, de tantas recomendaciones de lectura, de tantas advertencias e, incluso, amenazas, si, después de todo ello, ha llegado usted hasta aquí, enhorabuena: es momento, por fin, de entrar en materia.

Veamos, ahora ya sí con detalle, qué vamos a hacer y con qué elementos contamos para ello.

2.1.- Especificación.

Así pues, enfrentamos el siguiente proyecto:

1. Se trata de desarrollar una librería en Python 3 para el uso de un Raspberry como sistema de *adquisición de datos* (conversión de variables analógicas a digitales). La librería tiene que permitir seleccionar el hardware ADC entre tres alternativas *nativas*: 1/ bus oneWire, 2/ placa “china” -no tiene fabricante explícito- que monta un ADS1115, y 3/ un Arduino ProMini. Las medidas tomadas por el ADC se convertirán a un formato *legible*.
2. La librería tiene que facilitar la conexión de varias alternativas de sensores: los adecuados para el bus oneWire en su caso (ver 2.3.1, “bus oneWire”) y sensores genéricos para temperatura, presión, distancia, sonido, etcétera que transformen la variable medida a voltios para las otras dos opciones. Así como para añadir un ADC no *nativo* será necesario escribir una nueva clase, los sensores se deberán poder añadir sin necesidad de modificar el código de la librería. Incluso se deberá poder hacer así para sensores tabulados (no lineales).
3. La librería debe permitir la incorporación de nuevos tipos de conversores ADC con facilidad. Las clases que implementen el control de los ADC *nativos* estarán diseñadas de tal forma -*interface*- que se puedan replicar para cualquier ADC cuyo manejo sea estándar. De esta forma la aplicación del cliente -la del usuario de la librería- no necesitará modificarse incluso si se añade un nuevo tipo de ADC.

2.2.- Raspberry.

En este blog no encontrará una mala palabra del Raspberry. Se trata de un ordenador (microordenador, nanoordenador, ordenador de bolsillo, como quiera) del tamaño de una tarjeta de crédito y completamente operativo (aplicaciones compatibles con office, navegación por Internet, reproducción de contenidos multimedia, etcétera). Y, por si fuera poco, el *anteúltimo* modelo -perfectamente válido a los efectos de este blog- cuesta menos de 40 €. Añádale una fuente de alimentación (la *oficial* cuesta 12 €; las hay estupendas por la mitad), la tarjeta µSD y tiene usted un ordenador hecho y derecho, como los de verdad. Una bicoca para los críos y para usted mismo a poco *cacharrines* que sea. Visite, ahora sí, la WEB (“<https://www.raspberrypi.org/>”) para más información.

Además tiene enlace bluetooth, wifi y, lo que es más importante para este blog, un conector de 40 pines al que se le pueden conectar cosas que a un ordenador de sobremesa o a un portátil de los convencionales no hay por donde -enchufárselas-.

El Raspberry fue concebido para estudiantes, proponiendo como lenguaje de programación el Python, que ya viene preinstalado en el *Raspbian*.

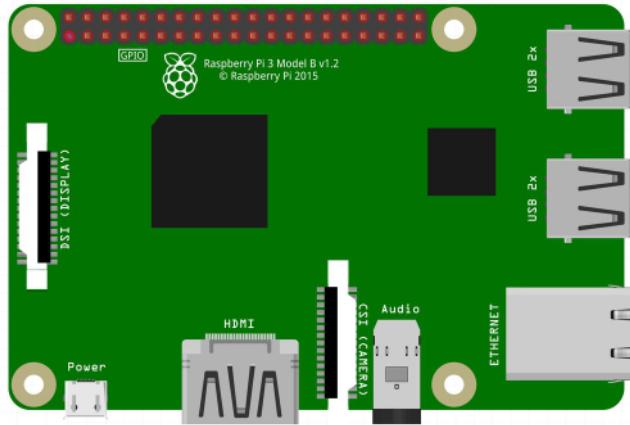


Figura 8. Raspberry Pi 3B

Y se acabaron las alabanzas; las encontrará a patadas en Internet, junto con miles y miles de WEBs en las que se describe el producto y las otras mil cosas que se pueden hacer con él. Baste finalmente con indicar que cualquier modelo a partir del “2” -Raspberry Pi 2- es más que suficiente para este proyecto.

2.3.- Conversores ADC.

Un conversor ADC convierte los voltios de entrada en un número con el que poder operar.

! “ADC” y “bus”: Estos dos términos ni son intercambiables ni se refieren a la misma cosa: mientras que el primero hace referencia a un “chip” (o placa electrónica, o módulo), el segundo se refiere a los sistemas de comunicación entre cualquier tipo de chips, ya sea en un pequeño sistema como el de este proyecto o en equipos de mucha más complejidad. En este proyecto -EIDEAnalog- se usan tanto chips ADC convencionales -el ADS1115- como los pequeños conversores ADC que van integrados en los dispositivos “sueltos” que se conectan, en nuestro caso, al Raspberry por medio de un “bus” oneWire (los termómetros DS18B20). Este bus -el oneWire- resuelve así los dos problemas simultáneamente: por un lado los sensores dan ya la conversión hecha y el bus -esa es su misión- lo transmite al Raspberry; como se ve, el conjunto es una mezcla de sensores, conversor(es) analógico digitales y bus de comunicación al que, desde el punto de vista de la librería que estamos diseñando, controlará una sola clase -“oneWire”-.

En el caso del ADS1115, este se conecta al Raspberry mediante el “bus” i2c y para el del Arduino se usa el bus “serie” convencional -RS232- (buses que, por otra parte, permiten conectar al Raspberry todo tipo de cosas: displays, cámaras de vídeo, teclados, impresoras, ...).

Al ser precisamente objeto de este proyecto el diseño de unas clases que hagan homogéneo el acceso a cualquiera de las soluciones mencionadas, y estar entrelazados -desde el punto de vista lógico- el acceso al bus y la propia captura de los datos de los sensores, es por lo que, al referirnos a las clases que controlan los conjuntos bus + ADC, usaremos ambos términos casi de forma indistinta (e, incluso, el *palabro* “ADCBus”, que carece de cualquier rigor técnico).

2.3.1.- Bus oneWire.

El bus oneWire lo inventó Dallas Semiconductor Corp. y es un protocolo (conjunto de especificaciones tanto de hardware como de software) mediante el cual se pueden conectar multitud de dispositivos electrónicos entre sí. Tiene la ventaja de que no necesita más que un hilo para la trasmisión de datos (dos, en realidad, el otro es “tierra” o “masa” o “0”, como prefiera llamarlo) para todos los dispositivos conectados (cuyo número es casi ilimitado; naturalmente hay limitaciones debidas a la fuente de alimentación, longitudes, interferencias, espacio físico, ...). La velocidad de transmisión no es muy alta 16 kbits/s (unos 1600 caracteres -letras- por segundo) aunque mucho más que suficiente para nuestro proyecto.

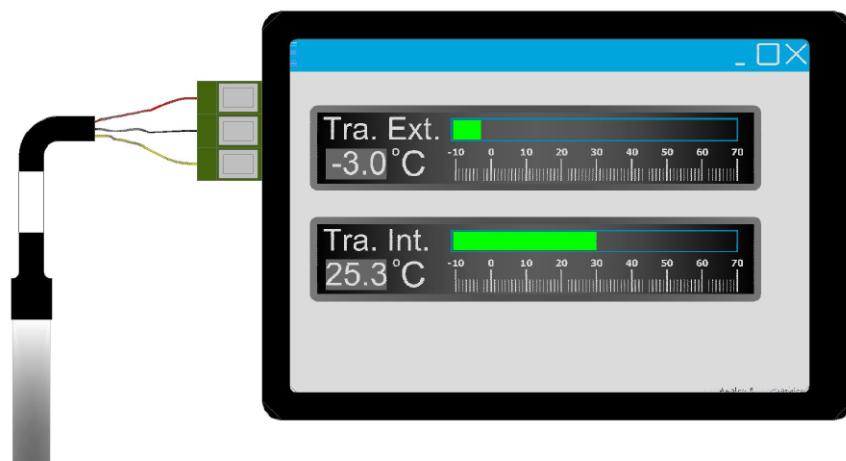


Figura 9. Kit comercial para medida de temperatura con Raspberry (oneWire)

El Raspberry puede actuar como “master” (el nombre lo dice todo) de un bus oneWire sin necesidad de ningún añadido de hardware; es, por tanto, *gratis* (ver anexo II, “Configuración de Raspbian”). Al bus se le pueden conectar varios tipos de dispositivos, sensores de temperatura entre ellos. En este blog usaremos el DS18B20 para los ejemplos; se trata de un sensor de temperatura con un microchip que implementa un conversor ADC y la electrónica necesaria para la comunicación -a través del bus-, todo debidamente encapsulado. Cada uno de los termómetros (y de cualquier otro dispositivo “esclavo” conectado al bus) tiene una *matrícula* diferente, que es un código hexadecimal de 12 caracteres.



Figura 10. Sensor de temperatura DS18B20. Chip y encapsulado a prueba de agua.

En este caso -DS18B20- se necesitan, además de los dos hilos mencionados, un tercero y una pequeña resistencia para dar corriente a los sensores.

El driver -software- estándar de Raspberry para su implementación -la del bus oneWire- funciona de una forma peculiar: al arrancar el Raspberry el driver “sondea” el bus, detectando todo lo que haya conectado al mismo, y crea una carpeta para cada uno de los dispositivos que encuentra (Ver 6.1, “Clase oneWire”). Después, cuando se quiere leer el sensor, lo que hay que hacer es mirar dentro del fichero de texto “w1_slave” de la carpeta correspondiente, identificada por el código hexadecimal del sensor -su *matrícula*-.

El procedimiento es cómodo, toda vez que sin necesidad de ningún software específico se puede saber la temperatura leyendo el fichero en cuestión. El problema es que es un procedimiento ridículamente lento, al menos en términos de lo que suele ser habitual para los dispositivos ADC, incluso los baratos: por término medio el tiempo que se tarda en obtener la lectura es de unos 0,9 segundos por sensor. A pesar de ello, es suficiente para nuestro proyecto, ya que, al fin y al cabo, se trata de presentar diferentes soluciones; esta es una de ellas.

(En realidad el sensor, incluida la electrónica que lo acompaña tarda una fracción de esos 0,9 s. Es la posterior lectura del fichero lo que hace el ciclo completo tan lento).

El bus oneWire se implementa en el Raspberry mediante tres de sus pines “GPIO”.

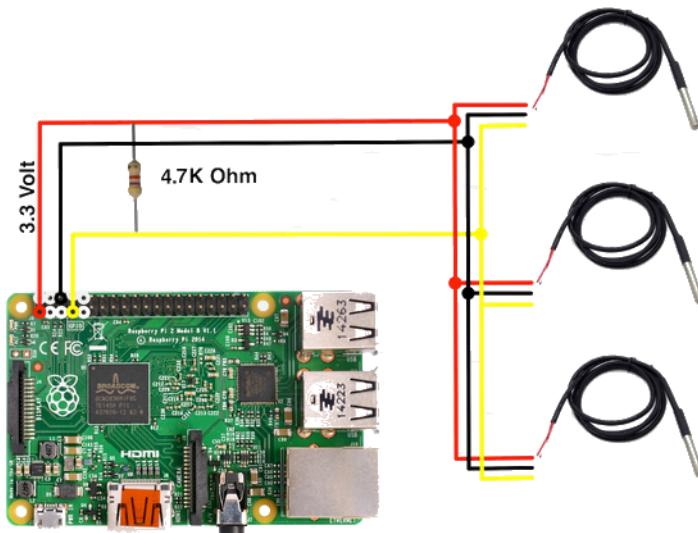


Figura 11. Conexión de un bus oneWire.

! Por una cuestión de ocupación del conector GPIO la clase oneWire de esta librería activa un pin del GPIO para que suministre 3.3 V al bus oneWire. En la implementación que normalmente se encuentra -figura 11 anterior- se usan unos pines que impiden la posterior conexión de las pantallas LCD de 3.5" más habituales -figura 9-. También se activa un pin diferente para el bus propiamente dicho (de nuevo el pin habitual está en la zona de conexión de la pantalla LCD. Ver anexo II, "Configuración de Raspbian").

2.3.2.- ADS1115.

El ADS1115 es un ADC de 16 bits diseñado por “Texas Instruments”; tiene cuatro canales que pueden dar una lectura directa (4, por consiguiente -lecturas directas-) o diferenciales (2, como máximo; también se pueden emplear configuraciones mixtas -dos directas + una diferencial-; en realidad la selección de la modalidad se hace sobre la marcha, con lo que la elección, salvo por la conexión de los sensores, que se supone permanente, se puede cambiar incluso funcionando). Se conecta al Raspberry mediante el bus i2c.

El chip tiene 6 tensiones de referencia programables (o “ganancias”; para ajustarse, dentro de lo posible, al rango del sensor) y 8 velocidades de muestreo diferentes, también programables (en realidad la velocidad de muestreo -la cadencia a la que lee un canal- es



Figura 12. ADS1115

siempre la misma, 860 veces por segundo; lo que hace es promediar las que le dé tiempo a hacer en función de la velocidad aparente que usted le pida: si le pide 475 promediará 2, si le pide 250 cuatro, etcétera -hasta 8-. Si le pide 860 le dará como resultado una sola medida; cuantas más medidas promedie mas preciso es el resultado). También tiene la posibilidad -que no usaremos en este blog- de comparar el valor medido con unos límites programables de referencia, suministrando alarmas en caso de que el valor se salga de lo predefinido.

Para lo que cuesta (céntimos en grandes cantidades; siga leyendo), *un pepino*.

Para usar el chip tal y como llega de fábrica hay que ser muy *cacharrines* (el chip mide 3 x 3 mm y tiene 10 patillas), así que lo mejor es buscar alguna placa que lo incorpore. La más popular es la “Sparkfun”, pero nosotros hemos elegido una más que robusta placa fabricada en china que se conecta directamente al Raspberry y tiene terminales -bornas-, de forma que no hay que soldar nada para emplearla. (Lo de que es china es un suponer, porque más allá de la referencia “HW-660” que figura en la misma no tiene otra filiación). Además tiene un termistor (NTC) incorporado que se puede conectar al canal uno mediante un “jumper” -de nuevo sin soldar nada-, y que nos vendrá al pelo para comprobar el funcionamiento de la librería con sensores no lineales (ver 5.2.3.- “Agente de calculo ‘tabulado’”). La HW-660 se puede comprar en eBay o Amazon por 12 € (incluso menos).

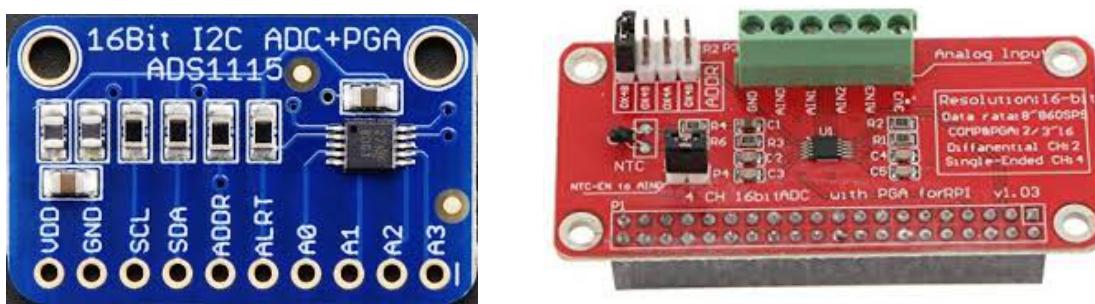


Figura 13. Placas con ADS1115. Sparkfun (izquierda) y HW-660

Un par de párrafos más arriba se ha hecho mención a la posibilidad -necesidad- de programar el ADS1115. La derecha para informarse es el *datasheet* del chip; lo puede consultar, y descargar, desde “www.ti.com/lit/ds/symlink/ads1115.pdf”. Esa es la derecha, decimos. No se preocupe, en el punto 4.1., “Clase ADS1115” está -el *datasheet*- bastante desmenuzado para justificar la elaboración de la clase correspondiente; igual es mejor que empiece por ahí.

Finalmente: la configuración estándar del bus i2c funciona a 100 kbits/s, con lo que cada lectura tarda, entre la orden de disparo, lo que tarda el chip en convertir y la recepción de la información, un pelín más que lo que tarda en convertir nada más (algo más de un par de milisegundos para la transmisión y 1,2 milisegundos para la conversión a 860 SPS).

2.3.3.- Arduino ProMini.

Arduino es una empresa italiana que tuvo la genial idea de usar algunos microcontroladores de Atmel (empresa norteamericana ahora propiedad de “Microchip Technology Inc”, también norteamericana) y montarlos en unas placas de forma que se

pudieran utilizar con facilidad; algo parecido a lo hecho con el ADS1115 (ver punto anterior) por Sparkfun -y también por el mencionado *chino desconocido*- desde el punto de vista del hardware y, cuestión algo más difícil de explicar -lo hecho-, desde el punto de vista del software. En definitiva, ha conseguido -arduino- que las placas que fabrica, y muy en especial el arduino “UNO”, se hayan convertido en una especie de estándar en la industria microelectrónica. El arduino ProMini que usaremos en este blog es una versión mínima en tamaño del arduino UNO, sin conectores.

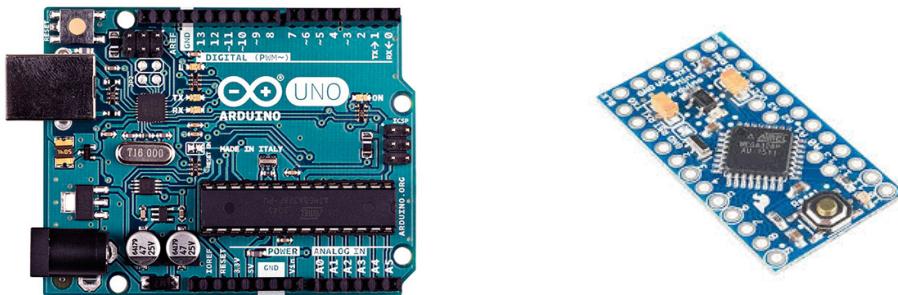


Figura 14. Arduino UNO (izquierda) y Promini.

(Por cierto, uno de los fundadores de Arduino es catalán -David Cuartielles; un sabio que además es una buenísima persona. No hay más que conocerlo; uno de los autores del blog tiene el gusto-. El *alma mater* del asunto fue -ver a continuación- Massimo Banzi, otro sabio y también bueno. La cuestión es que al cabo de unos años de fundar la compañía hubo algunas discrepancias entre cuatro de los socios -los dos mencionados entre ellos- y el que, al final, se quedó con la marca, hasta que Banzi y otros crearon la Fundación Arduino en 2017 y se hicieron con todas las marcas; ahora se entiende lo de buenas personas).

Los Arduino -el microcontrolador Atmel- no son -solo- conversores A/D, aunque tienen uno incorporado (tiene ocho canales de conversión). Son, en eso se parecen al Raspberry, ordenadores en miniatura, aunque el Arduino está más orientado a la conexión con el mundo real (entendido este, más o menos, como lo que ahora se conoce como IoT, aunque cuando se fundó Arduino el término no existía -la cosa sí-). Digamos que al Arduino no se le puede pedir que se conecte a una pantalla HDMI, o directamente a un lápiz de memoria USB, por ejemplo (o no, al menos, a los modelos básicos).

Además, para usar un Arduino este necesita su propio programa, de forma que hay que buscar alguno que ya lo esté (programado) para usar el -su- ADC y que, además, tenga alguna forma de comunicación también programada (todos los Arduino tienen implementado el bus oneWire -como en el Raspberry simplemente se necesita un driver-, el i2c -lo mismo- y varios canales de comunicación serie).

Hay un fabricante español que tiene lo que se necesita: un ProMini montado en una placa que, a su vez, viene preparada para montarse directamente en el Raspberry. Más aún, aunque se trata de una programación propia -ver 4.3, “Clase arduino”; la compañía en cuestión lo usa como parte de un “data logger” (un monitor/registrador de datos)-, responde perfectamente a lo que se pretende: convierte los 8 canales analógicos que tiene (10 bits) y envía las mediciones empaquetadas por el canal serie a petición. Lo usaremos como otra opción en la librería.



Figura 15. Placa con Promini para Raspberry.

2.4.- Lenguaje de programación. Python. IDE.

En este blog no encontrará ni una mala palabra de Python. Alguna loa al respecto ya se nos ha escapado más arriba, así que ahora nos las ahorramos.

Baste indicar que la implementación que hace Python de todo aquello orientado a la OOP está a la última. Añádale -al Python, que no a las técnicas de programación OOP, que esa es otra; vea el siguiente punto- una curva de aprendizaje más que generosa y tenemos la herramienta perfecta.

(Bueno, perfecta igual no: Python es un lenguaje *interpretado* -no compilado-, con lo que es relativamente lento. “Lento” quiere decir que, por ejemplo, un programa en Python que calcule el cuadrado de los 1.000.000 primeros números tarda, en un ordenador de sobremesa veterano, 0.23 segundos, o sea, 0.23 microsegundos de promedio por operación. No faltarán quien tache esta cifra de ridícula al compararla con la velocidad de un iPad, pongamos por caso: él -o ella- sabrá. Por cierto, en un Raspberry Pi 2 tarda 1,2 microsegundos: ¡Cachis!, es cinco veces más lento; en un Raspberry 3 B+ tarda alrededor de 1,0 microsegundos).

En cuanto al entorno de desarrollo (el IDE; “Integrated Development Environment”) a utilizar hay discrepancias. En principio vale el que trae -traía, siga leyendo- por defecto el lenguaje, es decir, lo que aparece -aparecía- cuando usted selecciona la opción “Python 3” (o “2”) en “Programación” -o un fichero “.py”- en el Raspberry. En las últimas versiones -distribuciones- de Raspbian (“Raspbian Buster” a finales del 2019) se ha sustituido éste -maldición- por el “Thonny Python IDE”. El entorno en cuestión, el *Thonny* este, es muy probable que mejore el antiguo, pero cuando uno está mayor ya se sabe que es renuente a los cambios (y mucho más si son impuestos).

Si se estrena usted -siguiendo este blog- con Python pruebe con el Thonny de marras; seguro que funciona bien. Si es usted desobediente y prefiere el entorno antiguo tiene que cacharrear un poco: consulte el anexo II, “Configuración de Raspbian” para instalarse el IDLE antiguo.

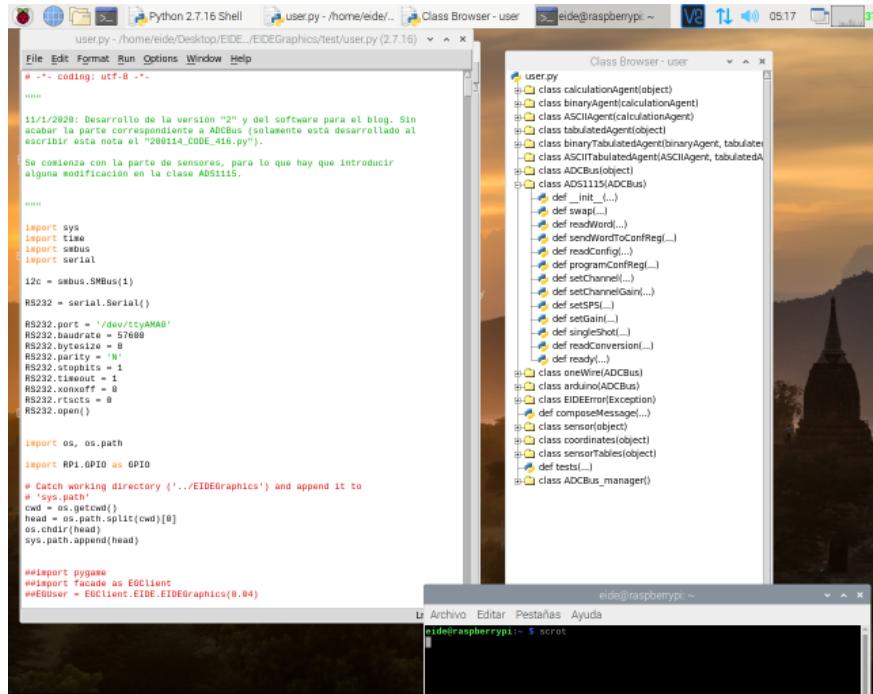


Figura 16. Entorno de desarrollo (IDLE) para Python.

2.5.- Análisis (y programación) OOP

2.5.1.- Generalidades.

Si alguien le ha dicho que la OOP es una cuestión fácil es A) porque no tiene ni idea y/o es un botarate ó B) porque, por alguna razón, miente como un bellaco.

Programar un ordenador puede ser, incluso, un pasatiempo (dicho sea en el mejor de los sentidos). Se trata de una máquina tan potente, y resulta tan entretenido -para multitudes, que no para mayorías- ver como acaba haciendo lo que queremos que haga, que hay quien le dedica a la cuestión las horas muertas. No digamos ya si lo que se pretende que haga -a base de programarlo- resuelve en minutos lo que hecho a mano llevaría horas (o días, o semanas).

La programación “procedural” (digamos, para simplificar, la que se usa sin recurrir a la OOP) tiene una curva de aprendizaje generosa: uno se maneja con el “*for-(next)*”, con el “*if-(then)-else*” (lo que está entre paréntesis es que no se usa en Python) y con el concepto de *rutina* (“función” en Python) y es capaz de escribir programas para resolver cuestiones bastante complejas; hay aplicaciones -programas- escritos con estos recursos y muy poco más que harían palidecer de envidia a los programadores del Apolo X.

Y, sin embargo, este tipo de programación presenta inconvenientes; hay que resaltar su -escasa- “reusabilidad”.

Este concepto hace mención a la posibilidad de volver a usar en una aplicación el código -todo o parte- previamente escrito para otra. Este asunto fue durante mucho tiempo

(digamos que desde que se escribieron los primeros programas -en los '50- hasta comienzos de los '90) cuestión muy discutida -y mal resuelta-. No había forma de que el “reuso” fuese indoloro: en mayor o menor medida el “copia-pegar” nunca funcionaba a la primera; en el mejor de los casos -y a base de una disciplina militar- las “rutinas” se podían reutilizar casi sin modificación, pero las garantías de que el código no presentaría *a posteriori* problemas de mantenimiento eran mínimas. Cualquier “mejora” -este es otro asunto, el de las “mejoras”, que daría para otro blog como este por sí sólo- en la *rutina* original requería siempre del usuario -cliente- cierto esfuerzo para (re)adaptar su programa. En la mayor parte de los casos la rutina incorporada se acababa reescribiendo y el supuesto ahorro de tiempo se convertía en una pérdida: los ejemplos son numerosísimos; cualquier persona que haya desarrollado software para que lo use otro sabe perfectamente de qué estamos hablando.

En este contexto, la técnica de OOP viene a resolver éste y otros problemas relacionados con la reusabilidad y el mantenimiento del software. El código que tenga la más mínima probabilidad de ser reutilizado se incorpora por medio de clases cuya instanciación posterior se traduce en la posibilidad de usarlo de forma totalmente opaca (el encapsulamiento de código y atributos) respecto a lo que contienen. Las clases, a su vez -estamos simplificando algo- pasan a formar parte de “librerías” -*libraries*- que contienen todas las clases que ayudan a resolver uno o varios problemas de un determinado ámbito.

2.5.2.- Ejemplo

Usted está empezando a hacer pinitos en Python; más concretamente quiere programar una calculadora elemental: introducidos dos números y una operación (“+”, “-”, “*” o “/”) se efectúa la operación y se muestra el resultado.

Dejemos de lado el cómo lo ha hecho (cosa, por otro lado, bastante simple; el que se haga lo dicho en el párrafo anterior -el cálculo- una vez que se han introducido en la máquina las tres cosas -los dos números y la operación-, queremos decir). La cuestión es que las herramientas básicas -*nativas*- de Python para lograr una cierta interactividad son penosas: la función “input()” es primitiva a más no poder: su flamante calculadora se merece algo más à la mode. Algo así:

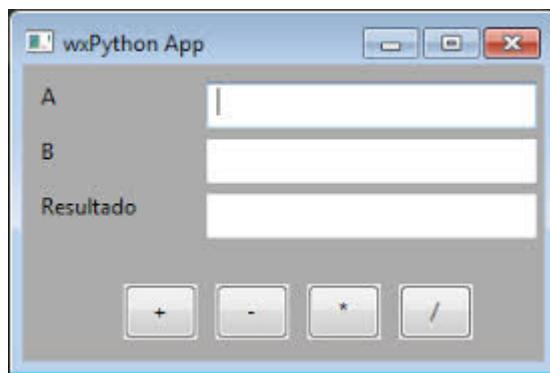


Figura 17. Ejemplo de uso de wxPython.

Aquí viene cuando la OOP acude en su ayuda. Si hay alguna librería para ello bastará, si es cierto todo lo que se dice (que lo es), hacer uso de ella. Usted consulta Internet, redes

sociales, le pregunta a algún conocido que está impuesto en el tema para, finalmente, llegar a la conclusión de que lo suyo es usar “wxPython”. (Por cierto, hay otra librería *GUI* muy usada, “TkInter”, que fue la primera para Python. Hay cierta rivalidad entre ambas - entre sus adeptos-: los autores del blog se decantan, sin ninguna duda, por wxPython).

La librería en cuestión tiene una *clase* “TextCtrl”, que, debidamente *instanciada*, produce un *objeto*, que es la ventanita donde se escribe la variable “A”. El correspondiente código sería algo así:

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

(No salga corriendo; ya le hemos advertido que lo de la OOP no es ninguna bicoca, y además, de momento, no tiene más que quedarse con la música). Las otras dos, para “B” y el resultado serían:

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

Digamos, simplificando, que en un *tal* “self.panel” (-lo que hay dentro de- la ventana en la que se ejecutará todo el proceso) habrá tres ventanitas; en dos de ellas se podrá escribir (“style = wx.PROCESS_ENTER”) y en otra sólo mirar (“style = wx.TE_READONLY”; deje lo del “-1” y el “” para otro día).

Ahora necesitamos los rótulos (“A”, “B” y “Resultado”; usted se creía que con lo anterior bastaba. No). Hay que escribir algo así:

```
rotA = wx.StaticText(self.panel, -1, "A")
rotB = wx.StaticText(self.panel, -1, "B")
rotResultado = wx.StaticText(self.panel, -1, "Resultado")
```

con lo que tendremos los rótulos (hemos instanciado la clase “StaticText” tres veces: tres rótulos).

Y, así, parecido con todo: cuatro instancias a “wx.Button” (las operaciones), decirle dónde tiene que poner cada cosa en el panel (o se puede hacer automáticamente con otro objeto, en este caso de la clase “wx.GridBagSizer”; la cosa se complica), instanciar el panel donde se desarrollará el drama (“self.panel = wx.Panel(self)”) dentro de la ventana que, cómo no, es un objeto de otra clase (¿“Calculadora”, quizá?) que, a su vez, hereda de la clase “wxFrame”.

Coja aire, la cuestión es que usted no tiene que programar *nada* -nada más que lo anterior, queremos decir- para tener una ventana con un panel dentro que, a su vez, tiene dentro los botones, los rótulos, las ventanitas (en las cuales, por cierto, tendrá usted la funcionalidad de un procesador de textos: insertar, borrar, etcétera. También podrá moverse por toda la ventana con el ratón y escribir donde haya seleccionado, etc, etc). Y todo interactivo: si usted tiene alguna noción previa de programación tiene una idea cabal de lo que cuesta conseguir un interface así, y seguro que lo anterior le empieza a intrigar; y si no, tranquilo, es un peaje (lo abstruso del tema) que merece la pena.

2.5.3.- OOP. Conclusión.

Hay -habemos- un montón de blogs y páginas WEB en los que encontrará muchas tonterías acerca de lo indoloro de la OOP: son especialmente conmovedoras las que

introducen el tema con la clase “mammal” (“mamífero”) que tiene como subclases “cat” y “dog”, las cuales, a su vez, participan de las clases “tail” (“cola”) y “color”; sus objetos son “Boby” y “Sansón” que mueven la cola (o no). Las de vehículos automóviles y sus conductores por lo general tampoco tienen desperdicio.

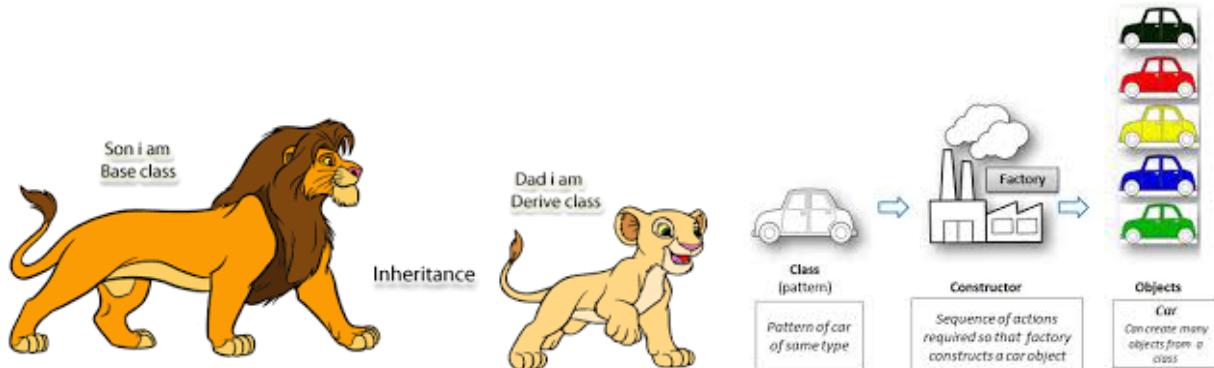


Figura 18. OOP made easy: mammals y cars.

(La cuestión no es que el planteamiento simplón que proponen sea esencialmente falso, que no lo es. El problema es intentar explicar un zoológico con “mammal”, “dog”, “cat”, “Boby” y “Sansón” o una carrera de Fórmula 1 con “car” y “driver”. Hay mucho - muchísimo- más *tomate* que simplemente eso).

La OOP no es un asunto de una tarde -ni de un trimestre, probablemente-, así que tómeselo con calma. Este blog no pretende otra cosa que ayudarle a dar los *segundos pasos* (ver 1.2.4., “[Conocimientos previos necesarios de] Python. OOP” para los primeros). En concreto, usted debería, al finalizar el blog con aprovechamiento, estar lo suficientemente familiarizado con los siguientes conceptos de la OOP:

- Clases y objetos.
 - Concepto. Instanciación de clases.
 - Atributos de clase. Métodos. Atributos de objeto.
 - “Herencia” (“Inheritance”). Herencia múltiple.
 - Composición de objetos.
 - Documentación del proyecto. Notación UML (nociiones).
- Librerías.
 - Uso. Cliente de la librería.

3.- Análisis preliminar de la arquitectura de la librería.

Tenga en cuenta que no hay una receta mágica para lo que vamos a desarrollar ahora, y, sin embargo, es la parte más importante del proyecto (al menos para quien lo sigue con el afán de aprender algo de arquitectura de aplicaciones usando OOP).

Vamos a ello. El primer asunto a resolver es fragmentar nuestro problema en subproblemas que sean más fáciles de encarar que el original. Además, aquí lo vamos a hacer empleando el “análisis orientado a objetos” (para poder, después, emplear la OOP).

3.1.- Análisis preliminar “nombre/verbo”.

En nuestro caso tenemos que delimitar las clases e ir pensando, casi al tiempo, en lo que los objetos de las clases tienen que ser capaces de hacer (métodos). Para una primera aproximación se puede usar el procedimiento “nombre/verbo”, que ayuda a ir desbrozando el problema.

Consiste en:

1. Redactar la especificación del proyecto (esto ya lo tenemos).
2. Identificar los nombres que lo caracterizan -al proyecto- y los verbos que aparecen: las nombres serían las clases y los verbos los métodos. (Quizá de forma no tan categórica: algunos nombres corresponden con algo que no tiene la *edad* suficiente para ser una clase en sí mismo; en ese caso será un *atributo* de los objetos).

A continuación se reproduce nuestra especificación (Punto 2.1), con los verbos subrayados y los nombres en negrita (evidentemente los que parece que puedan ser clases o métodos, no vamos a resaltar “trata”, “desarrollar” o “librería”, por ejemplo, aunque no descartaremos nada que no esté medianamente claro que no es ni una cosa -clase- ni otra -método-)

- Se trata de desarrollar una librería en Python 3.8 para el uso de un Raspberry como sistema de adquisición de datos (**conversión** de variables analógicas a digitales). La librería tiene que permitir **seleccionar** el hardware ADC entre tres alternativas nativas: 1/ bus oneWire, 2/ placa “china” -no tiene fabricante explícito- que monta un ADS1115, y 3/ un Arduino ProMini. Las medidas tomadas por el ADC se **convertirán** a un formato legible (“readout”).
- La librería tiene que facilitar la conexión de varias alternativas de sensores: los adecuados para el bus oneWire en su caso (ver 2.3.1, “bus oneWire”) y sensores genéricos para temperatura, presión, distancia, sonido, etcétera que **transformen** la variable medida a voltios para las otras dos opciones. Así como para **añadir** un ADC no nativo será necesario escribir una nueva clase, los sensores se deberán poder **añadir** sin necesidad de modificar el código de la librería. Incluso se deberá poder hacer así para sensores tabulados (no lineales).
- La librería debe permitir la incorporación de nuevos tipos de conversores ADC con facilidad. Las clases que implementen el control de los ADC nativos estarán diseñadas de tal forma -interface- que se puedan **replicar** para cualquier ADC cuyo manejo sea estándar. De esta forma la aplicación del cliente -la del usuario de la librería- no necesitará modificarse incluso si se **añade** un nuevo tipo de ADC.

Como estamos buscando sujetos y sus habilidades -del programa-, podemos descartar ya algunos de los términos seleccionados: por ejemplo, "Raspberry": el programa no le va a hacer nada al Raspberry, lo va a hacer en él. Tampoco va a conectar -el programa- los sensores a ningún sitio; esto se hace con un destornillador.

Cribando con este criterio, nos quedaríamos con:

Nombres (candidatos a clases): "variable analógica", "variable digital", "bus", "oneWire", "ADS1115", "arduino", "sensor(es)", "ADC", "(sensores) tabulados", "(sensores) lineales", "conversor(es)".

Verbos (candidatos a métodos): "tomar", "convertir", "seleccionar", "transformar", "añadir", "replicar".

Si, además, nos fijamos en la cantidad de veces que aparecen (de momento a ojo, sin contar), parece que "ADC", "sensor" y "bus" empiezan a destacar; los verbos, en cambio, no se repiten mucho, salvo, quizás "convertir" (y aparentemente se refiere más a las necesidades del usuario de la librería -el *cliente*- que a las habilidades del programa) y "añadir".

Ya se ha advertido que este método no es muy científico; hay que depurar el resultado. Vamos al siguiente paso; aguce los sentidos que lo que viene es mucho más escurridizo que lo anterior.

3.2.- Las clases como proveedoras de servicios.

Se trata de analizar las -posibles- clases desde el punto de vista de lo que pueden ofrecer -los objetos obtenidos al instanciarlas- al *cliente*. Las clases tienen que ayudar a resolver el problema, no a complicarlo.

Hay otra forma de verlo: el lenguaje -básico- de programación, Python en nuestro caso, tiene lo que tiene que tener: variables, operadores, elementos para el control del flujo, tratamiento de errores, gestión de clases, etcétera. Lo que no le podemos pedir al lenguaje es que tenga instrucciones para nuestro problema concreto: no le podemos pedir que tenga una instrucción "convierteCanal" para dársela a un ADC o "consultaLaTabla" para sacar un valor de una tabla interpolando linealmente.

Estamos entrando en uno de los aspectos nucleares de la OOP: los métodos de las clases tienen que *suplementar* al lenguaje de programación de forma que al cliente le resulte -casi- tan fácil de usar una instrucción primitiva -*print*- como una nueva -*convierteCanal*-.

Si, además, se conciben las clases de forma que valgan para cualquier problema similar -o, quizás, no tan similar-, nos vamos acercando a la forma correcta de afrontar el problema. Lo desarrollaremos a continuación.

3.2.1.- Clases para buses. Clases para sensores. Cálculo.

Las clases que controlarán los buses (oneWire, ADS1115, arduino) son muy *utilitarias*. Contienen -encapsulan- lo necesario para que estos funcionen y poco más; rozan con el *hierro* -el hardware-. Si echa un vistazo preliminar a los puntos 4 y 6, verá que una gran

parte de las mismas (en especial la ADS1115, que usaremos como estándar a lo largo del blog) no son más que la transcripción casi literal del *datasheet*. Haremos, además, un esfuerzo para mantenerlas todo lo independientes que sea posible (desacopladas; “*loosely coupled*”) del resto de la librería.

Mucho más sutil es, en cambio, el asunto de los sensores y del **cálculo** de la medida (candidatos, parece, a clases -sensor- y métodos -cálculo-; la *medida* acabará siendo un *atributo* del sensor). Como veremos a continuación, las clases relacionadas con estos asuntos son mucho más abstractas.

Salvo el bus oneWire, que entrega la lectura del sensor casi en formato legible (o sin casi: lo da en ASCII aunque en m°C -milésimas de °C-), lo normal es que un ADC dé la lectura en formato binario; así es en el caso de los otros dos ADC que usamos en el blog: en “complemento a 2” el ADS1115 y directo en el caso del Arduino. Y nosotros necesitamos **transformarlo** a formato *legible* (por ejemplo: 00010000 = 32 si es “directo” -Arduino- y 10000001 = - 127 en complemento a 2 -ADS1115-).

3.2.2.- método *readout*.

Para la transformación -la haga quién la haga y se haga donde se haga- se necesita:

1. Saber en qué formato -complemento a 2 o binario- viene la lectura codificada -del el ADC-. Dicho de otra manera, de qué ADC viene.
2. Saber qué parámetros tiene el sensor. En el caso más normal el sensor dará una tensión -que es lo que, en definitiva, convierte el ADC: la tensión- directamente proporcional a lo que está midiendo; °C, por ejemplo-. Como es altamente improbable que haya una correspondencia “uno a uno” entre el parámetro y la tensión, el sensor está caracterizado, de momento, por la “ganancia”: es el cociente entre las unidades de lo que mide y los voltios que da. En este blog se usa el LM35 como referencia en multitud de ocasiones: tiene una ganancia de 100 (ver 5.1, “Sensores”).

En ocasiones (siga viendo 5.1, “Sensores”) el sensor tiene un “desfase” entre su “0” y el “0” de lo que mide: a ese valor lo llamaremos *cero* -“zero”-.

Además hay sensores cuya lectura (o a lo que sea que queremos que convierta) no guarda una relación lineal como la descrita en el punto anterior (siga viendo 5.1, “Sensores”); en este caso diremos que la conversión -entre lo que nos da el ADC y lo que queremos ver- está *tabulada*.

3. Finalmente, hay un parámetro, que depende del sensor, aunque lo use también el ADC, que llamaremos, de momento, “tensión máxima” (“Vref”). No se trata de los voltios que *aguanta* el sensor, si no, más bien, de los que da -como máximo-. Este factor interviene en el cálculo de la lectura, así que pasa a formar parte del repertorio de cosas que se necesitan para convertir la lectura del ADC a la lectura “legible” (valgan las dos redundancias). Este parámetro, junto con la ganancia, el cero y la -eventual- tabla, formará parte de la “librería” de sensores (ver un poco más adelante).

De manera que, para calcular la lectura, se necesitan datos del bus (formato de datos) y del sensor (ganancia, cero, tensión de referencia y tabla -en su caso-); la pregunta es: ¿dónde -método- se hace el cálculo?; ¿quién -clase- contiene ese método?

Veamos: parece claro que tiene que haber una clase “sensor”, que, al menos, contenga los datos necesarios de cada uno de los sensores que hay conectados al sistema: 1) supuesto que la información de cada tipo de sensor -LM35- esté centralizada -lo estará-, se necesitará que uno de los atributos del objeto sensor sea el tipo -‘LM35’, o el que sea- y 2) a qué bus está conectado -ADS1115, por ejemplo- (se podría hacer al revés: que el bus contenga la información de los sensores que tiene conectados, en cuyo caso sería -el objeto de la clase bus- el responsable del cálculo). Sería una alternativa perfectamente válida, aunque, como se verá en su momento, menos flexible: no se puede instanciar un sensor -un sensor de verdad, no un tipo de sensor- hasta saber a qué bus va conectado. Sí es posible, en cambio, instanciar un bus sin especificar qué sensores tiene conectados: puede que no tenga ninguno).

Vamos a delegar la tarea de calcular la lectura, pues, a la clase sensor. Decidido. En una primera aproximación la clase tendrá un método -“readout”- que calculará, pasándole como argumento lo que da el bus (el binario; el ASCII) la lectura *legible*. Para ello se habrán pasado como argumentos todos los valores relacionados más arriba en el momento de la instancia -del sensor-. Adelantemos, todavía no toca, que la orden sería algo así:

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

(Se explica con mas detalle en 5.4, “Código de la clase sensor. Instanciación”; adelantemos que myBus es el bus al que está conectado y “2” el canal).

Volvamos al análisis, que ahora es cuando viene lo bueno.

Uno de los baremos para calificar si un análisis -y su implementación posterior- OOP es correcto es la ausencia de cláusulas “do case” (o “switch”). Estas son de muchísima utilidad en programación procedural: en un caso como el que estamos analizando el método *readout* -clase sensor- las incorporaría; como el cálculo no el mismo dependiendo del bus al que esté conectado el sensor (no es lo mismo transformar un número en complemento a 2 -ADS1115- que directo -Arduino- o que, solamente, añadirle una coma a la lectura del oneWire), la implementación sería un “*do case*”, como queda dicho. Además, en el caso de los sensores tabulados hay que añadir la tabla a lo anterior: Lo que sigue está reproducido literalmente del punto 5.3.5. “Herencia múltiple. Clases ‘binaryTabulatedAgent’ y ‘ASCII TabulatedAgent’.

“En un proyecto como el que tenemos entre manos -y casi siempre- lo que ocurre si hay un sensor tabulado (el NTC de la placa china hecha con el ADS1115, ver figura 19), es que éste está conectado a un canal del ADCBus, entregando por tanto una lectura en un determinado formato (binario, ASCII, ...) que hay que convertir a decimal -como la de cualquier otro sensor- para, después, pasársela al “tabulatedAgent” y que éste la pase por la tabla para calcular la lectura definitiva”.

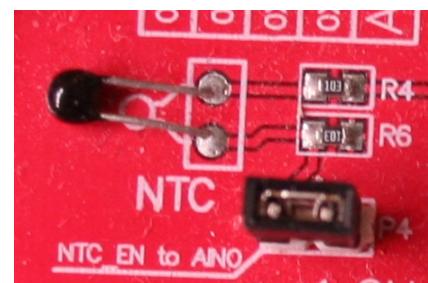


Figura 19. NTC

Es decir, que si tenemos un sensor tabulado este ha tenido que pasar antes por una transformación de la lectura como la que se hace para cualquier sensor: primero se convierte la lectura y, después, se pasa por la tabla.

De nuevo, con programación procedural, esto supondría un nuevo “*do case*” o un “*if-then-else*” (que, idealmente, tampoco deberían aparecer en la OOP).

(Aunque queda claramente fuera del alcance de este blog entrar en más profundidades, digamos que la razón por la que la no existencia de cláusulas “switch” es muestra de buen análisis OOP porque es síntoma de poco *acoplamiento* entre clases, asunto deseable en lo referente a la reusabilidad del software).

3.2.3.- Clases *agentes de cálculo*.

Resolveremos el asunto creando una familia de clases “agentes de cálculo” (para lo que, además, utilizaremos la técnica de herencia -“*inheritance*”- para crear unos a partir de otros) que implementen los diferentes algoritmos de cálculo en función del formato (binario ó ASCII), de los datos del sensor y de que tenga -el sensor- o no tabla asociada (Ver 5.3, “Algoritmos de cálculo de la lectura. Clases agent””).

El *quid de la cuestión* no será tanto el cómo se crea esta familia de clases -lo de la *herencia*-, si no la técnica que emplearemos para vincularlas al sensor: será el objeto de éste -clase sensor- el que a su vez instancie la clase que necesite de los agentes de cálculo: conseguiremos de esta forma que el método *readout* de la clase sensor, que es el encargado de calcular la lectura, lo haga delegando en el objeto *agente de cálculo*: el código del método será exactamente el mismo sean los que sean sensor y bus, y que, por otra parte, no será más que una línea de código:

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

(como se verá en su momento).

Tenemos ya planteada la estructura básica de la librería:

- Clases para cada uno de los buses que pueden conectarse al Raspberry: oneWire, ADS1115 y Arduino. Como se verá en su momento se creará también una clase abstracta -ADCBus- de la que heredarán estas tres; es -lo de la clase abstracta- mitad cuestión de estilo y mitad de utilidad: hay un par de métodos básicos (imprescindibles) que comparten todas las clases bus.
- Una clase sensor cuya utilidad básica será la de almacenar las características de cada sensor *real*, incluyendo el bus al que está conectado; un método, *readout*, de la clase calculará, mediante un objeto de una de las clases agentes de cálculo -que habrá instanciado oportunamente- la lectura del sensor.
- Una familia de clases *agentes de cálculo* según lo descrito más arriba.

En total no llega a una docena de clases: el problema a resolver no es muy complicado.

3.3.- Esquema de clases de la librería.

Utilizando una notación basada en el estándar OMT (con alguna licencia propia), el siguiente esquema resume lo anterior.

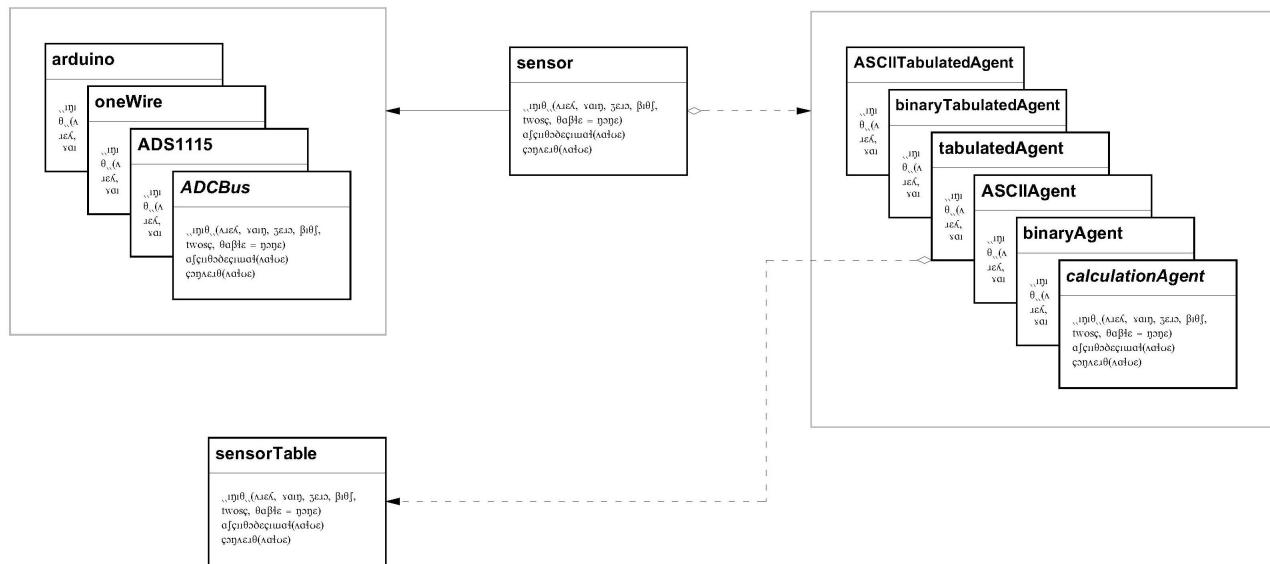


Figura 3-2. Esquema preliminar de clases

Es muy posible que el lector haya apreciado la aparición de una nueva clase de la que no hemos dicho nada de momento: “`sensorTable`”. Se trata de una clase que, a efectos del análisis global de la librería, tiene un interés secundario. Adelantemos que se implementa para manejar las tablas de puntos de los sensores tabulados, que pueden ser voluminosas (ver punto 7, “Tablas de sensores [‘`sensorTable`’]”).

La notación empleada esta derivada de la del -excelente- libro “Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides”, que, a su vez, está derivada del estándar OMT (https://en.wikipedia.org/wiki/Object-modeling_technique). Queda más allá del alcance de este blog entrar en detalles de esta notación, aunque se ha procurado respetar la simbología; parte de ella se puede deducir, a lo largo del blog, por las sucesivas explicaciones.

4.- Clase ADS1115

Recapitulemos brevemente antes de comenzar con el diseño de las clases para control de los dispositivos a los que se conectan los sensores: "ADS1115", "oneWire" y "Arduino".

Nuestro objetivo es diseñar una librería en Python (de la que estas clases formarán parte) con la que resulte sencillo utilizar varias opciones de hardware existentes para la "captura de datos" utilizando un Raspberry.

El diseño de las clases (de su interfaz: básicamente los "métodos" que permitirán interactuar con los objetos de la clase) que controlen directamente las diferentes opciones de hardware -los ADC- tiene que hacerse de tal forma que permita cambiar de -hardware- conversor analógico digital con un mínimo de modificaciones en el programa de aplicación, idealmente solo la instanciación del bus: lo lograremos. Estas clases tienen como misión la comunicación con el bus nada más (Ver a continuación la acepción que se hace de estos términos -"bus" y "ADC").



Figura 4-1. Placa con ADS1115



"ADC" y "bus": Estos dos términos ni son intercambiables ni se refieren a la misma cosa: mientras que el primero hace referencia a un "chip" (o placa electrónica, o módulo), el segundo se refiere a los sistemas de comunicación entre cualquier tipo de chips, ya sea en un pequeño sistema como el de este proyecto o en equipos de mucha más complejidad. En este proyecto -EIDEAnalog- se usan tanto chips ADC convencionales -el ADS1115- como los pequeños conversores ADC que van integrados en los dispositivos "sueltos" que se conectan, en nuestro caso, al Raspberry por medio de un "bus" oneWire (los termómetros DS18B20). Este bus -el oneWire- resuelve así los dos problemas simultáneamente: por un lado los sensores dan ya la conversión hecha y el bus -esa es su misión- lo transmite al Raspberry; como se ve, el conjunto es una mezcla de sensores, conversor(es) analógico digitales y bus de comunicación al que, desde el punto de vista de la librería que estamos diseñando, controlará una sola clase -"oneWire"-.

En el caso del ADS1115, este se conecta al Raspberry mediante el "bus" i2c y para el del Arduino se usa el bus "serie" convencional -RS232- (buses que, por otra parte, permiten conectar al Raspberry todo tipo de cosas: displays, cámaras de vídeo, teclados, impresoras, ...).

Al ser precisamente objeto de este proyecto el diseño de unas clases que hagan homogéneo el acceso a cualquiera de las soluciones mencionadas, y estar tan entrelazados -desde el punto de vista lógico- el acceso al bus y la propia captura de los datos de los sensores, es por lo que, al referirnos a las clases que controlan los conjuntos bus + ADC, usaremos ambos términos casi de forma indistinta (e, incluso, en ocasiones el *palabro* "ADCBus", que carece de cualquier rigor técnico).

Las clases de control de los ADC no tendrán en cuenta para nada qué sensores están conectados a estos; para eso habrá que ir pensando en otras clases cuyos objetos manejen la información relativa a aquellos -los sensores: tipo, ganancia, etcétera- y, eventualmente, hagan los cálculos pertinentes para convertir los datos que entrega el ADC en los valores que, posteriormente, grabaremos, mostraremos por pantalla o lo que sea que queramos hacer con ellos. Así pues, las clases de control de los ADC acaban su función dando el valor *mondo y lirondo* que entregan estos, que en el caso más general será una información en binario (valor directo o en "complemento a 2"), aunque los termómetros DS18B20 del bus oneWire entreguen, casi, la lectura final, como se verá en su momento.

Finalmente, esto es una perogrullada, se instanciarán las clases *ADCBus* tantas veces como sea necesario. Es decir que si, por ejemplo, al Raspberry se conectan varios ADS1115 (cuatro es el máximo), otras tantas instancias de la clase correspondiente permitirán controlarlos simultáneamente y por separado; otro tanto ocurrirá, si el hardware lo permite, con las otras dos estructuras -oneWire y Arduino-. Más aun, y de nuevo si el hardware lo permite, podrán funcionar buses (ADC) de diferente tipo al tiempo.

Para hacer el desarrollo más comprensible lo haremos en dos fases: en la primera nos fijaremos como objetivo leer el primer canal del conversor con los parámetros “por omisión” del ADS1115 (ganancia, muestras por segundo -SPS-. A esto lo llamaremos “versión mínima”); esto ya nos permitirá leer un sensor, lo que, al fin y al cabo, ya se va aproximando a la finalidad básica del proyecto. Seguidamente, añadiremos los métodos necesarios para sofisticar el control del ADS1115 y poder cambiar el canal, la ganancia y las SPS y hacer la lectura de todos los canales.

En aras de no complicar demasiado este primer desarrollo no se incluye, de entrada, el código de captura de errores, que hace el código final bastante más confuso y puede desanimar al principiante.

En lo que no se simplificará el código, ni siquiera para hacerlo algo más comprensible de entrada, es en cualquier aspecto que nos desvíe de los siguientes objetivos:

1. que el interfaz sea homogéneo para todas las clases de los diferentes dispositivos “*ADCBus*”.
2. que un método se tenga que modificar posteriormente como consecuencia de una nueva estructura de métodos o atributos -datos- de la clase que se adapte mejor a especificaciones que ya se conozcan de antemano.

puede ser que la aplicación de estos dos principios en algún momento cause extrañeza por una aparente complicación en la estructura de una clase, pero estará justificado para lograr los objetivos mencionados. Si se da esta circunstancia se comentará debidamente.

4.1.- Desarrollo.

Empecemos: en esencia, necesitaremos poder hacer lo siguiente para leer un canal del ADS1115:

1. Dar la orden al ADS1115 de que inicie la conversión (por omisión está en modo “single shot”: hay que decirle que convierta).
2. Verificar si ha terminado de hacer la conversión.
3. Pedirle que la entregue -la conversión-.

Cada una de estas tres acciones corresponderá con un método de la clase (*singleShot*, *ready* y *readConversion* respectivamente); digamos que ya tenemos un esbozo del *interfaz* de esta clase: estos tres métodos. Como veremos inmediatamente estos métodos necesitarán de otros -métodos- más especializados que iremos analizando sobre la marcha.

4.1.1.- Orden al ADS de inicio de conversión.

Para que el ADS1115 convierta hay que escribir un “1” en la posición 15 de su registro de configuración. Y esto hay que hacerlo sin cambiar los otros 15 bits del registro por lo que, teniendo en cuenta que para cambiar algo en el registro de configuración hay que cambiar los 16 (el de disparo -comienzo conversión-, en este caso, y los otros 15), hay que leer antes lo que tiene -el registro de configuración-, cambiar sólo el bit en cuestión y volver a mandárselo al ADS1115.

Esta técnica, leer el registro de configuración, cambiar lo que se deseé solamente y volver a enviarlo, es la forma habitual de programar el ADS1115, por lo que conviene ir pensando, en principio, en dos métodos “especializados” (ver párrafo anterior), uno que lea el registro de configuración y otro que lo modifique (grabe).

Por otra parte, el ADS1115 tiene otro registro -conversión- para cuya lectura se empleará exactamente el mismo código que para -leer- el de configuración, por lo que, a efectos de no duplicarlo -el código-, conviene que sea un solo método el que lo hace y al que se le pasa el parámetro para indicarle cuál de los dos registros se quiere leer. Iremos poniendo nombre a los métodos correspondientes: “*readWord(register)*” y “*sendWordToConfReg*”; el “*(register)*” del primero quiere decir que para invocar este método tendremos que decirle qué registro queremos leer.



Uso del bus i2c: Dado el alcance del blog, no podemos detenernos en las interioridades del bus *i2c*. Es un protocolo para gestionar mediante dos hilos la comunicación entre chips; la librería “*smbus*” dota a Python de las instrucciones -métodos- necesarios para interactuar a través de ella. Dado que, en realidad, el protocolo “*smbus*” es un subconjunto del protocolo “*i2c*” para bajas velocidades, sorprende que la librería no se llame así -“*i2c*”- y no “*smbus*”. Para acabar de liarlo -o no-, en este blog llamamos “*i2c*” a la instancia de “*smbus*” que usamos para interactuar con el ADS1115: ustedes perdonen -o no-.

Hay otro asunto a tener en cuenta: la orden de lectura de los dos registros del ADS1115 se hace a través de la orden “*read_word_data*” de la librería “*smbus*” que, en efecto, devuelve -retorna- un “word” (16 bits) en 2 bytes. El problema es que, tal y como los devuelve esta instrucción, los 2 bytes vienen “bailados”: el último delante y el primero detrás. Es cierto que la orden de escritura -“*write_word_data*”- también los baila cuando los envía, por lo que podríamos olvidarnos de ello, localizar el bit -o bits- en lo que se ha leído, cambiarlo y enviarlo tal cual: funcionaría. El problema es que esto sería muy confuso con el registro de configuración (habría que dar la vuelta al propio manual -*datasheet*- del chip para aclararse). Necesitamos, pues, otro método “utilitario” al que le pasemos un “word” y nos lo devuelva bailado: lo llamaremos “*swap*”.

Finalmente: conviene que el *interfaz* -de la clase- sea lo más legible posible. Si el cliente de la clase escribe *readWord(0)* para leer el registro de configuración y *readWord(1)* para leer la conversión, la aplicación funcionará perfectamente, pero su lectura -y probablemente su uso- serán mucho más confusas que si “envolvemos” estas dos variantes con dos métodos que se llamen *readConfig* y *readConversion*. Son unas pocas líneas más de código y una ventaja innegable para no equivocarse y leer el código con más facilidad.

Así pues, ya hemos deducido que, de momento, necesitamos 7 métodos que, resumiendo, son los siguientes:

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

Tabla I. Clase ADS1115 (*mínima*). Los métodos del *interfaz* en negrita.

Con la explicación previa a la tabla debería ser suficiente para comprender el contenido de cada uno de los métodos. Insistiremos un poco más a medida que los vamos presentando para que las ideas queden completamente claras (no se explican con detalle las instrucciones concretas, salvo en el caso de que tengan alguna complicación que vaya más allá del conocimiento básico de Python):

swap(self, word)

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

- devuelve (*return*) el *word* que se le pasa con las dos mitades de 8 bits permutadas.
- El *pseudoparámetro self* es una exigencia sintáctica de Python para todos los métodos de una clase. También se debe usar como prefijo siempre que una clase haga referencia a atributos y métodos de la propia clase (por ejemplo: “*self.swap*”, “*self.addr*”).

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
```

- Retorna el contenido del registro “*register*” del ADS1115 con los dos bytes ordenados (“*valor = self.swap(valor)*”)
- La información -el “*word*”- se obtiene mediante el método “*read_word_data*” del “objeto” *i2c*, el cual es una instancia de la clase “*smbus*”. La librería “*smbus*” viene incluida en Python y es la que controla el bus i2c en su nivel más bajo: más adelante, al comentar el código del módulo completo, se volverá sobre este asunto.
- “*self.addr*” es la dirección en el bus i2c del ADS que corresponde con la instancia de la clase que se esté usando (recuérdese que puede haber hasta cuatro ADS1115 conectados simultáneamente, cada uno con una dirección diferente). El valor de “*self.addr*” se fija en el método “*__init__*” de la clase (“constructor” en la jerga de la OOP; ver más adelante).

readConfig(self)

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

- Hace uso del método anterior para leer el registro de configuración. El número de éste -del registro de configuración- es el contenido del “atributo de clase” “*ADS1115.confReg*”; es una cuestión de estilo -usar atributos en lugar de valores *monos y lirondos* (“0”, en este caso)- que mejora la legibilidad y mantenimiento del código.

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)
```

- Envía el “word” que se pasa como argumento al registro de configuración del ADS1115.

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)
```

- Este método está comentado con mucho detalle antes de la tabla de métodos previa.

Bien, las cinco trozos de código precedentes forman casi todo el núcleo de la *versión mínima* de la clase “ADS1115”; recuerde que se trata, de momento, de poder leer el canal “por defecto” (el “0”). Conviene comentar un poco lo precedente ya que, al fin y al cabo, es la primera implementación de código en Python del blog.

- Si nos fijamos en el método “*singleShot*” (que, en definitiva, es el objetivo de esta primera tacada de código; se trata de poder decirle al ADS1115 que convierta para, después, leer la conversión), observaremos que está formado nada más que por tres instrucciones: “valor = self.readConfig()”, “valor = valor | ADS1115.startConv” y “self.sendWordToConfReg(valor)”. De éstas, sólo hay una ,“valor = valor | ADS1115.startConv”, que es *nativa* de Python; las otras dos son llamadas a métodos de la propia clase. Esta es una de las piedras angulares de la OOP: se trata de generar un lenguaje propio de forma que el flujo del programa se convierta en una serie de llamadas entre métodos de los diferentes objetos (o, como en este caso, de algunos métodos del objeto a sus propios métodos). Esta es una idea recurrente que ya se ha comentado (ver 3.2, “Las clases como proveedoras de servicios.”) sobre la que se insistirá en el blog.
- La clase (y su *interfaz*) están diseñados sin una especificación “final” concreta. En nuestro caso ésta -la especificación- es bastante genérica: se trata de leer variables analógicas con un Raspberry; uno de los dispositivos empleados es el ADS1115. No se especifica si los valores que se obtengan van a usarse para ser mostrados, o volcados a una página WEB; no se sabe si darán lugar a avisos de alarma. Ni siquiera sabemos si van a ser transformados a una lectura “visible” (por ejemplo, 23,2 °C; 7,0 kg/cm²). Y, sin embargo, todo el trabajo que hemos hecho sabemos desde ya que va a ser utilizado tal y como lo hemos finalizado. Este es otro de los pilares de la OOP: las clases -sus objetos- ofrecen al cliente un *interfaz* que, de alguna manera, es una extensión del lenguaje de programación. A partir de la instanciación de la clase, “*singleShot()*” pasa a ser una instrucción más. No importa si la clase cambia -el código de la clase-: basta con mantener el *interfaz* para que el cliente que la usa sea ajeno a esas modificaciones; su código -de usted; el programa de aplicación del *cliente*- podrá seguir funcionando exactamente igual, mientras la clase lo haga. También volveremos sobre esta idea una y otra vez.

- Todos los métodos serán accesibles al cliente de la clase. En otras palabras, se podrá usar, por ejemplo, “`readWord(0)`” en lugar de “`readConfig()`”. Es ésta, la de poner o no ciertos métodos de la clase a disposición del cliente, una cuestión más de metodología que práctica, toda vez que en Python no es posible hurtar al cliente (si se empeña) el acceso directo a los métodos y atributos de una clase. Lo que haremos en este proyecto es indicar en la tabla (resumen) de la clase qué métodos se consideran “públicos” (los que se recomienda que se usen como *interfaz*) y cuales no.
- Se usa un estilo de programación que evita constantes explícitas en el código (por ejemplo: “`ADS1115.confReg`” en lugar de “`0`”; “`ADS1115.startConv`” en lugar de “`0x80`”). Esto obliga a añadir unas líneas de código en el “encabezamiento” (más adelante se matiza el término “encabezamiento”: “Atributos de clase” y “método `__init__()`” de la clase, pero el esfuerzo se paga con creces en la mejora de legibilidad y facilidad de mantenimiento del código. (El avisado lector quizá detecte a continuación -“Atributos de clase”- que los valores de “`startConv`” y “`done`” son idénticos. Solamente por lo dicho en éste párrafo merece la pena tener dos atributos idénticos; además, si se usa la estructura de este método para otro ADC cuya programación siga los mismos esquemas que el ADS1115, la existencia de los dos atributos con diferente finalidad será una -otra- ventaja).

4.1.2.- Comprobación de fin de conversión.

El -arduo- trabajo hecho para poder dar la orden de comienzo de conversión (“`singleShot`” más otros cuatro métodos de utilidad) comenzará a rendir frutos inmediatamente. El método para comprobar si el ADS1115 ha terminado la conversión es:

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

que retorna “1” (*True*) si ha acabado o “0” (*False*) en caso contrario. Su funcionamiento no necesita de ninguna explicación (o, en todo caso, no más de las que ya se han dado en el capítulo anterior “Orden al ADS de inicio de conversión”).

4.1.3.- Lectura de la conversión

De nuevo el trabajo previo simplifica este método:

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

que retorna el valor en el -único- formato del ADS1115: “complemento a 2”

Ya tenemos, casi, el código completo -el básico, al menos- de la clase en su *versión mínima*: faltan los “atributos de clase” y el método “`__init__()`”. Veamos.

4.1.4.- Atributos de clase.

Python tiene un procedimiento para definir atributos (“variables”) comunes a todos los objetos de una clase concreta. Esta particularidad tiene sus inconvenientes, ya que si se cambia el valor de uno de estos atributos éste varía para todos los objetos de la clase, lo que de alguna forma es contrario a uno de los paradigmas de la OOP, el “*encapsulado*” de los datos (como muchos inconvenientes que se plantean con las herramientas informáticas en general, a quien no le parezca conveniente le basta con no usar esta posibilidad y todos tan contentos).

En nuestro proyecto usaremos los atributos de clase para almacenar las “constantes” del ADS1115, es decir, todos los parámetros del chip que nos harán falta para su uso (y que, evidentemente, usarán todas las instancias de la clase, excepción hecha de la dirección - address-, como se menciona en el siguiente punto); de momento hemos detectado los siguientes, cuya primera aparición en el código se hará de la siguiente forma:

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

Para usar estos atributos el código, dentro y fuera de la clase, los direcciona como “ADS1115.XXXX”. También son invocables como atributos de todas las instancias de la clase.

4.1.5.- Método `__init__()`. Atributos de la instancia.

Cada instancia de la clase ADS1115 tiene algunos atributos propios, no compartidos con el resto de instancias (ver punto anterior). De momento hemos usado uno “`self.addr`”, que es la dirección de cada ADS1115 concreto en el bus *i2c*. Su valor se inicializa en el método “`__init__`” y se implementa así:

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

El método “`init`” es el llamado “constructor” en la jerga de la OOP. Aparte de tener un aspecto amedrantador es, simplemente, un método que se ejecuta automáticamente cuando se instancia una clase. Otra cosa es que su uso pueda (ver 5.4, “Código de la clase *sensor*. Instanciación”) incluir una lógica propia que haga que la mera instanciación del objeto le dote de ciertas sofisticaciones que sean necesarias para que su uso posterior dé lo que se espera de él -del objeto-.

No es el caso aquí -ADS1115-: ya se ha comentado que las clases para uso de los buses son simplonas y, básicamente, implementan lo necesario para su funcionamiento. En el caso del ADS1115, lo usaremos, de momento -recuerde que estamos escribiendo una *versión mínima*- para inicializar la dirección de la placa (ver figura 22) y, si el cliente lo decide, numerar la instancia -*ordinal*- y darle un nombre -*name*-; estos dos últimos tienen asignado por omisión “0” y “None” para el caso de que el cliente no asigne ninguno en particular.

En el caso de la *placa china*, la dirección se fija mediante un *jumper* en la placa que permite seleccionar entre cuatro direcciones distintas; la instanciación del ADS1115 tiene que hacerse con la dirección seleccionada (0x48 en la foto), de forma que la instancia creada queda asignada a la placa que tiene dicha dirección. La instanciación, pues, quedará con esta forma:

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

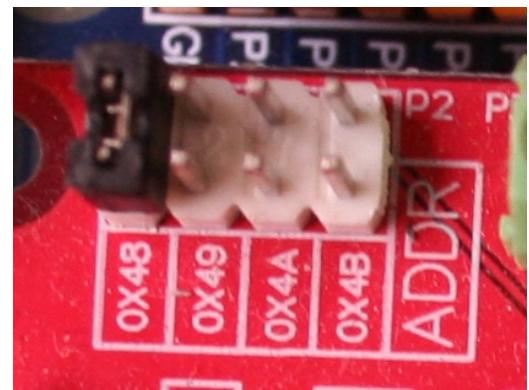


Figura 22. ADS1115 seleccionado en 0x48

queda creado un objeto “myBus” que es un objeto de la clase ADS1115 con la dirección 0x48; corresponde -gobernará- a una placa física que contiene un ADS1115 y que tiene asignada la dirección 0x48.

4.1.6.- Código completo de la versión mínima de la clase ADS1115.

Con lo anterior ya tenemos completo el código de una versión mínima (solamente lectura del primer canal -"0"- del ADS1115 con los valores de configuración por defecto) de la clase. Es:

```
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.- Instanciación (uso) de la clase ADS1115.

Para usar la clase -para poder usar los recursos (métodos) que contiene- hay que crear un objeto de ella -de la clase-; en nuestro caso la instrucción es (escribala a continuación del código de la propia clase):

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

que merece un par de comentarios:

- Argumento “0x48”: Es el que utilizará el método “`__init__`”, en este caso como “address”. La línea
`self.addr = address`

del propio método usa este valor -“0x48”- para iniciar el atributo de la instancia “`self.addr`”

Como en la instanciación no se especifican más valores los atributos “ordinal” y “name” de esta instancia quedan con los valores por omisión “0” y “None” (ver más arriba 4.1.5, “Método `__init__()`. Atributos de la instancia”).

- Como consecuencia de esta instrucción todos los métodos de la clase están ya accesibles. La forma de invocarlos es “`myBus.XXXX()`”. (También lo están todos los atributos -por ejemplo “`myBus.addr`”, 0x48 en nuestro caso-. Ver advertencia más arriba -al final de 4.1.1, “ Orden al ADS de inicio de conversión”- en cuanto a la posibilidad y la -in-conveniencia de acceder a métodos y atributos de un objeto).

De manera que, si queremos leer la conversión, tendremos que escribir el siguiente código (a continuación de “`myBus = ADS1115(0x48)`”):

```

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

```



El código anterior es perfectamente correcto y operativo. No obstante, si intenta ejecutarlo tal cual aparecerá el error “global name ‘i2c’ is not defined”: no está definido aun el objeto ‘i2c’ de la clase “smbus”. Hay que añadir las siguientes líneas de código al comienzo del módulo (hay una explicación algo más detallada en 4.1.1, “Orden al ADS de inicio de conversión”, ver también el anexo II, “Configuración de Raspbian”):

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

cuando ejecute este código, aparecerá la lectura de la conversión del primer canal del ADS1115. Será un valor entre 0 y 65535 (ambos inclusive; 65535 es $2^{16} - 1$).

Aunque de momento no toca, quizá el seguidor del blog quiera darle alguna interpretación al valor que obtiene. Sin entrar en muchos detalles: el valor “0” (“0000 0000 0000 0000” corresponde con 0 Voltios en la entrada del canal; el valor 32767 (un “0” seguido de 15 “1”: “0111 1111 1111 1111”) es el mayor valor positivo de la tensión que el ADS1115 puede leer con relación a la tensión de referencia a la que está programado. Como, de momento, damos como bueno el valor por defecto -aun no sabemos cambiarlo- y este es 2,048 V., este valor – “32767”- corresponde con este valor (o superior) de la tensión presente en el canal “0”. Todos los valores intermedios son valores entre 0 V. y 2,048 V.: hay que hacer una regla de tres. El resto de valores (de 32767 a 65535) son eventuales entradas negativas, que, midiendo sólo un canal, no pueden ser convertidas (ni conectadas directamente al ADS1115, por cierto).

(Es tentador incluir aquí las dos líneas de código necesarias para convertir este valor al decimal correspondiente, pero no lo haremos. Cuando analicemos las clases de los sensores veremos cómo se relacionan estas con otras clases -“agentes de cálculo”- de forma que entre ambas hagan esta transformación; no se trata de ningún algoritmo diferente de las dos líneas mencionadas, pero mejor incluirlo donde toca. Paciencia).

4.1.8.- ADS1115. Métodos para su configuración.

Hasta ahora nos hemos limitado a usar el ADS1115 con los valores por defecto con los que funciona (con los que se “autoprograma” cuando arranca -cuando se le da corriente-). Es evidente que es un uso (muy) limitado; recordemos que se le puede programar para lo siguiente:

1. Seleccionar el canal que se quiere medir (de 1 a 4). No se incluyen en la librería las 4 opciones de “entrada diferencial” (se incluye como ejercicio más adelante).
2. Seleccionar la ganancia (la tensión de referencia; no confundir con la ganancia del sensor): En función de la tensión que podemos esperar en la entrada (sensores), es posible modificar el valor -la referencia- con la que la compara el ADS1115. Así, por ejemplo, si estamos midiendo un voltaje que no esperamos que supere algunos mV (un termopar, por ejemplo) lo más razonable es usar la más pequeña de las referencias: 0,256 V, ya que es muy difícil que un termopar dé una tensión mayor (incluso esta tensión es demasiado grande: el tipo “K”, por ejemplo, da 41 mV para 1000 °C; los demás andan por ordenes de magnitud similares). Un sensor de temperatura tipo LM35 da 1,0 V para 100 °C, con lo que -si lo vamos a mantener en ese rango; los encapsulados comerciales a prueba de agua no aguantan mucho más- lo suyo sería usar la referencia de 1,024 V.

3. Seleccionar la “velocidad de muestreo”: aunque, en rigor, el ADS1115 “muestrea” - convierte- siempre a la misma velocidad, este valor -la “velocidad de muestreo”- hace referencia al número de muestras -conversiones- que promedia para darnos un valor: cuanto más bajo más conversiones y, por tanto, más fiable -y preciso. Y más lento-.
4. Fijar un umbral (o ventana) de alarma que el ADS1115 usará para generar un mensaje de aviso (Esta opción es bastante complicada -prolifa- de usar, por lo que, al menos de momento, la dejaremos fuera del contenido del blog).
5. Cambiar el modo de disparo de *single* a *continuo*: como el modo “continuo” supone un funcionamiento del ADS1115 que se aleja de la estandarización que pretendemos en el blog, también lo dejaremos fuera del mismo.

Salvo por lo minucioso de la programación el asunto no tiene otra complicación que la de tener en cuenta que cada una de las posibilidades -canal, ganancia, velocidad de muestreo- afecta a diferentes posiciones del registro de configuración; véase el código a continuación.

Además, lo mismo que en el método “*singleShot*” ya estudiado, hay que tener la precaución de no modificar los restantes bits del registro de configuración; hay que cambiar solamente los que toca. En este caso, y a diferencia de la orden de *start* mencionada, que siempre es un “1” en una posición determinada del registro (y, por tanto, es suficiente con leer el registro, hacerle un “or” y volver a enviarlo; ver 4.1.1, “Orden al ADS de inicio de conversión”), aquí se necesitan cuatro pasos:

1. Leer el registro de configuración.
2. Poner a “0” los bits correspondientes a lo que se quiere configurar (con un “*and*”).
3. Poner a “1” los que tienen que tomar ese valor de los que se pusieron a “0” en “2” (con un “*or*”).
4. Enviar el resultado al registro de configuración.

Si se hace con un cierto orden, la cosa es, se insiste, prolífica pero no difícil de entender. De entrada, es necesario este código en la clase ADS1115 (lo que ya se ha explicado, ver 4.1.4, “Atributos de clase”; se incluye de nuevo con el mismo formato que el resto):

	#0 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 = 0b000011111100000	#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      = 0b0111111000000000 #0|111|111|1|000|00000
SPS_8          = 0b0000000000000000 #0|000|000|0|000|00000
SPS_16         = 0b00000000000100000 #0|000|000|0|001|00000
SPS_32         = 0b00000000001000000 #0|000|000|0|010|00000
SPS_64         = 0b00000000001100000 #0|000|000|0|011|00000
SPS_128        = 0b00000000010000000 #0|000|000|0|100|00000
SPS_250        = 0b00000000010100000 #0|000|000|0|101|00000
SPS_475        = 0b00000000011000000 #0|000|000|0|110|00000
SPS_860        = 0b00000000011100000 #0|000|000|0|111|00000

```

este código no es más que parte del *datasheet* del ADS1115 pasado a variables de forma que se pueda usar con facilidad. De una forma u otra cada uno de los valores que contiene tiene que aparecer en el código explícitamente; en este blog hemos optado por agruparlos y presentarlos con un comentario de forma que la correspondencia con los bits del registro de configuración del ADS1115 sea lo más clara posible. El único añadido son los tres words de reset – *channelReset*, *gainReset* y *SPSReset*- que, como se verá más adelante, facilitan el código de programación.

(El word ‘reset’ que encabeza el asunto es una *boutade*, toda vez que no se usará: dejaría al ADS1115 en la misma situación que cuando arranca, lo que no se necesita hacer nunca).

De esta forma si se quiere, por ejemplo, seleccionar el canal 3, hay que hacer lo siguiente:

1. Leer el registro de configuración.
2. Poner a “0” los bits correspondientes a lo que se quiere configurar (con un “and” de lo leído y el word ‘*channelReset*’).
3. Poner a “1” los que tienen que tomar ese valor con un “or” del resultado de “2” y el word ‘*setChannel3*’.
4. Enviar el resultado al registro de configuración.

Y así con todo.

Parece que, tal y como estamos organizando las cosas, necesitaremos un método por cada una de las características -canal, SPS, ganancia- que queremos programar, ya que el “reset” -paso “2” anterior- es diferente para cada opción. Así es. Además, si se añade lo que viene a continuación, los tres métodos -cambio de canal, ganancia y velocidad de muestreo- consistirán en una única línea de código:

```

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),
)

```

Lo que se hace es agrupar en tres *tuples de tuples* -de dos words cada uno- las parejas de *words* que se necesitan para cada una de las programaciones posibles: cuatro canales, ocho velocidades de muestreo y seis ganancias. Ahora solo hay que decirle a cada uno de los tres métodos qué opción queremos que use, por ejemplo:

```
myBus.setChannel(3)      # seleccionar el canal "3"
myBus.setSPS(7)          # seleccionar 860 muestras por segundo
```

(recuerde que el primer elemento de un *tuple* es el "0")

Veamos ahora el código de los tres métodos:

```

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])

```

Al haber organizado las cosas así -los words de programación a pares y dentro de tuples a su vez-, se ha podido unificar la forma en la que se programa el ADS1115: con un método único *programConfReg* que, a su vez, usa los -ya explicados- métodos *readConfig* y *sendWordToConfReg*. Su código es:

```

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

```

Ya tenemos el núcleo de la clase ADS1115 completo: podemos cambiar de canal y de velocidad de muestreo y, esto ya lo teníamos, decirle que convierta y leer la conversión.



El ADS1115 tiene la posibilidad de trabajar con entradas diferenciales. Al no admitir las entradas individuales valores negativos de la tensión, se trata de poder conectar sensores que entregan tensiones que sí pueden tomar valores negativos. En concreto, los bits que se programan y la conexión del sensor responden a la siguiente tabla:

Bits "MUX"	Conexión
000	Positivo: AIN0; Negativo: AIN1
001	Positivo: AIN0; Negativo: AIN3
010	Positivo: AIN1; Negativo: AIN3
011	Positivo: AIN2; Negativo: AIN3

1. Añadir los atributos “de clase” que sean necesarios para poder usar esta opción.
2. Estudiar un nuevo método (*¿setDiffChannels?*) para activar las entradas diferenciales.

Queda, todavía, un asunto importante que conviene analizar.

4.1.9.- método *chooseVref* (tensión de referencia). Método *setChannelGain*.

En 4.1.8, “ADS1115. Métodos para su configuración” se ha explicado que cambiar la ganancia del ADC sirve para adaptarlo al rango -esperado- del sensor: no es lo mismo medir un termopar, que da milivoltios, que un sensor del que podemos esperar varios voltios; es conveniente darle la tensión de referencia al ADS1115 para que la conversión tenga una referencia mayor o menor, según convenga.

Es, por tanto, el propio sensor -y el uso previsto del mismo- el que fija la tensión de referencia. Por ejemplo, un LM35 convencional con encapsulado a prueba de agua (el *chino* que se encuentra a patadas en Internet, vamos) soporta 100 °C: 1,0 voltios como máximo. Como una de las tensiones de referencia del ADS1115 es 1,024 voltios, parece que esta es la ideal para esta situación. No hay ningún inconveniente en fijarla con la orden *myBus.setGain(2)* y, después, dar la orden de conversión.

Pero, si podemos automatizar el proceso -se debe poder, ya que el sensor tiene una tensión máxima prevista y el ADC tiene las que tiene de referencia- aparece el inconveniente de que el sensor no *sabe* esto (que el ADS1115 tiene una tensión de referencia de 1,024). En realidad el sensor -las características que el código tiene almacenadas de él; ver 5.1, “Sensores” *standardSensorsData*- ni siquiera *sabe* a qué bus está conectado. Lógico. Dicho de otra forma, tenemos un tipo de sensor ‘LM35’ que no pasará de 100 °C -el “1.0” de la lista ‘LM35’: [‘LM35’, 100, 0, 1.0, None] en el diccionario mencionado recién-, y nada más. El que *sabe* -tiene que saber- qué tensiones de referencia tiene es el bus, su clase.

Así que, de momento, necesitaremos la lista de tensiones de referencia del bus en los *atributos de clase*

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

e incluiremos un nuevo método en la clase ADS1115 que, pasándole la tensión del sensor como argumento, devuelva la tensión de referencia correspondiente (y su posición en la lista). Esto:

```
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)
```

método que quedará añadido a la clase a partir de ahora.

Ahora ya solo queda añadir a la clase un método que haga uso de lo anterior. (Aunque adelantamos acontecimientos, digamos que al método se le pasa como argumento el objeto sensor; este tiene -entre otros- como atributos el número de canal al que está conectado y la tensión de referencia que le corresponde entre las que tiene disponibles el ADC al que está conectado). Este es el método en cuestión:

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



Pasar un objeto como argumento en la llamada a un método tiene ciertas implicaciones de tipo más teórico que otra cosa en OOP; digamos, simplificando, que el método puede modificar el objeto en cuestión, lo que va contra uno de los paradigmas de la OOP. Como ya se ha advertido en más de una ocasión en Python es imposible poner el *interior* de los objetos a salvo del *programador bobo* (otro día explicamos lo de este tipo de programadores), por lo que se entiende que con este pequeño aviso está todo dicho.

4.1.10.- Clase ADS1115 completa. Ejemplo de uso.

En <https://github.com/Clave-EIDEAnalog/EIDEAnalog>

(EA_4111_ADS1115_COMPLETE_USAGE.py), tiene el código completo de acuerdo con lo desarrollado hasta aquí. Si, como es lo más probable, está usted aprendiendo, lo más recomendable es *corta-copiarlo* para poder enredar a gusto. Recuerde que tiene que importar el "smbus" e instanciarlo (nota-aviso en 4.1.7, "Instanciación [uso] de la clase ADS1115"). En ese punto tiene, también, un ejemplo básico de uso del bus: vuelva a usarlo para comprobar que lo tiene todo a punto.

Veamos algo más sofisticado que ese trocito de software: lo más probable es que usted quiera comprobar las lecturas de los cuatro canales. Conecte a ellos lo que le parezca: sensores, potenciómetros; conéctelos a "0" o a "3.3 V"; recuerde que al canal "1" (el "0" en la placa") se le puede conectar el NTC (Ver figura 19). El siguiente código leerá los cuatro canales:

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

```

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

```

Aparecerá por pantalla algo así:

```

canal 1: 17957
canal 2: 2533
canal 3: 8665
canal 4: 0

```

ni que decir tiene que los números dependerán de lo que tenga conectado: 0 V -un puente a “0”- debería dar una lectura próxima a “0”; el NTC alrededor de 18000.



El módulo “time” tiene un método -“time.time()”- que da, con una precisión lo suficientemente alta, el instante en el que es ejecutada (ahora = time.time() carga en la variable “ahora” el instante actual en segundos; no se preocupe *desde cuándo* son esos segundos: es irrelevante para este problema).

Usando este método y un código mínimo -no haga “prints”-, calcule la velocidad efectiva de muestreo que puede obtener del ADS1115 (El canal por omisión; cambie el “SPS” a 860. Sugerencia: programe un lazo que haga la conversión un número elevado de veces; 1000, por ejemplo).

Resultado: en torno a 400 muestras por segundo. La velocidad viene limitada por el trasiego de órdenes - *read.word*, *write.word*- por el bus i2c, que, si no cambia usted la velocidad -100.000 bits/s-, limita el proceso. En el anexo II, “Configuración de Raspbian” se indica cómo cambiar la velocidad a 400.000 bits/s; si lo cambia observará un aumento en la velocidad efectiva hasta, aproximadamente, 580 muestras/s.

4.1.11.- Clase ADS1115. Conclusión.

En los puntos anteriores se ha justificado la estructura de una clase para control de un chip ADC ADS1115; también se ha desarrollado completamente el software y se han planteado varios ejemplos de uso de la clase y ejercicios que, en alguna medida, serían los ejemplos que habría que haber desarrollado inmediatamente a continuación de aquellos. El desarrollo se ha hecho acompañado de todas las explicaciones que se han considerado necesarias; además, una primera parte de la clase se ha desarrollado de forma independiente y muy comentada, esperando así que el lector haya podido seguir las explicaciones con un esfuerzo razonable.

Más allá de la complejidad inherente a los algoritmos que implementa cada método de la clase (y de los que se han explicado las particularidades si hay alguna cuestión intrincada), hay varios detalles relativos a la arquitectura que conviene comentar a modo de conclusión:

- Para no complicar demasiado el desarrollo no se ha incluido el código de captura de errores (ver 1.2.4, “[Conocimientos previos necesarios] Python. OOP”).
- Se ha insistido en -e intentado fijarlo con el de esta clase- el concepto de *interfaz* de clase. También se ha insistido en el concepto de las clases como *proveedoras de servicios*. Además, se ha diseñado el *interfaz* de la clase de forma que sea

aplicable a los otros *ADCBus* de la librería (Ver 6.1, “Clase oneWire” y 6.2, “Clase arduino”).

- Se usan los *atributos de clase* para almacenar los que son comunes a todos los objetos de la misma. Se ha advertido de la posible *heterodoxia* de esta particularidad de Python.
- Se han planteado varios ejemplos de uso de la clase desde el punto de vista del *cliente* de la librería.
- Subsidiariamente se ha explicado con mucho detalle el funcionamiento del ADS1115, lo que no tiene nada que ver, *a priori*, con clases ni objetos (ni *a posteriori* tampoco).

5.- Clases para sensores. agentes de cálculo.

Una vez analizada -y programada- las clase para controlar el bus ADS1115, toca ahora idéntica labor para con los sensores.

En el punto “4” hemos ido deliberadamente paso a paso: era el primer contacto con el análisis y la programación OOP y no era cuestión de acabar de entrada con una clase ininteligible. Y la materia era relativamente fácil; un ADS1115 funciona como funciona y, por tanto, el software para controlarlo está relativamente acotado; además hicimos el análisis de la estructura de la clase en dos fases, lo que alivió bastante la cuestión.

En este punto analizaremos y programaremos la clase *sensor*. También analizaremos y programaremos unas clases *de apoyo* -a la clase sensor- correspondientes a entes completamente abstractos -los *agentes de cálculo*- . Más aún, crearemos una jerarquía con estos agentes para introducir el concepto de *herencia* entre clases, otra de las piedras angulares de la OOP (de paso, habrá también que introducir el concepto de *polimorfismo*, asimismo nuclear en la OOP; como se ve, este punto es importantísimo).



Objetos lógicos vs. Objetos reales: No conviene deducir del párrafo inmediatamente precedente que los -otros- objetos -sensores, ADCBus, que sí corresponden a objetos tangibles- sean por ello menos “abstractos”. En realidad todas las clases son – y corresponden con- una “abstracción”, ya que, al fin y al cabo, su contenido es puramente lógico; el que haya una correspondencia más o menos evidente entre la clase y algo tangible es, de hecho, lo menos habitual.

Más aun: el término *objeto(s)* de la técnica -OOP- es confuso, aunque inevitable en la práctica, toda vez que, en efecto, el programa se divide en porciones de código -las clases- cuya finalidad es implementar las abstracciones de la aplicación que, se insiste, en unos casos corresponderán con algo tangible, las menos, y en otras con *abstracciones-abstracciones*. Véase a continuación las clases *agente de cálculo* y comentarios sobre este mismo asunto en el punto 3, “Análisis preliminar de la arquitectura de la librería”.

Finalmente, se analizará también como la solución adoptada genera objetos *compuestos*, asunto igualmente nuclear en al análisis OOP.

5.1.- Sensores

Veamos en primer lugar de qué tipos de sensores nos vamos a ocupar en este blog.

1. **Sensores con lectura analógica.** Por definición una lectura analógica es la que entrega un sensor que transforma un parámetro físico (temperatura, humedad, presión) convertido a otro parámetro físico proporcionalmente (conforme a una “regla de tres”, de ahí lo de *analógicos*. No es broma). Son los más comunes. El LM35, por ejemplo, entrega 1,0 voltios por cada 100 °C que detecta: 0,0 V. son 0 °C; 0,5 V. son 50 °C y así sucesivamente.

El parámetro físico entregado suele ser una tensión -voltios-, pero no siempre es así. Es muy frecuente el uso de intensidad -en lugar de la tensión- como parámetro de salida, ya que es mucho más resistente a las interferencias. (En este caso no se podrían usar la mayor parte de los ADC directamente: habría que emplear lo que se llama una resistencia de bucle -o cierre-).

2. **Sensores “cuasi-proporcionales”:** Son muy similares a los descritos en el párrafo anterior, aunque en este caso la relación entre la salida y la entrada es lineal -en lugar de completamente proporcional-; dicho de otra manera, la salida -lo que sea, voltios, amperios- no es “0” cuando la entrada -temperatura, humedad- es “0”. El LM35 es como el LM35 pero con la salida desplazada 0.5 V.: 0 Volt. Son -50 °C; 0,5 Volt. 0 °C; 1,0 Volt 50 °C, etcétera.
3. **Sensores sin linealidad (tabulados):** Hay sensores cuya salida no es ni proporcional ni siquiera lineal con el parámetro medido. Aunque cada vez se usan menos (los de los apartados “1” y “2” son cada vez más baratos), aun se encuentran por el mundo; de hecho, una placa muy popular de ADC a base del ADS1115 tiene un termistor -resistencia que varía con la temperatura- montado que se puede conectar al primer canal del ADC (Ver figura 19). Lo usaremos como ejemplo de este tipo en el blog.

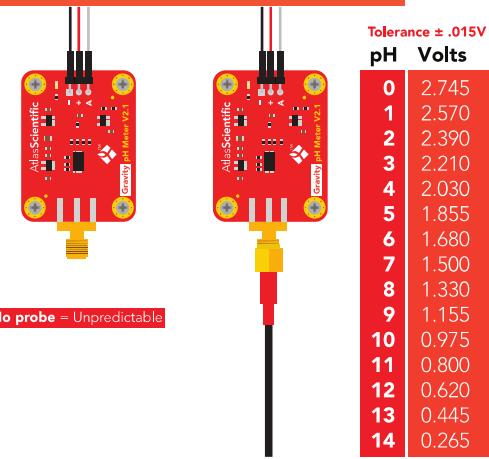
(No solamente hay sensores no lineales: hay multitud de ocasiones en las que se dispone tensiones que corresponden con alguna información -no tiene porque ser estrictamente un sensor- y en las que la relación entre esta -la información- y la tensión está tabulada: un ejemplo podría ser la relación entre la posición del vástagos de una válvula y la cantidad de agua que deja pasar. Otros ejemplos son la luminosidad de una bombilla respecto de la tensión que se le aplica, la lectura del *pH* o la *presión de vapor* de una sustancia en función de la temperatura; en general se trata de correspondencias que se podrían expresar analíticamente -con una “formula”-, pero es más cómodo -eso, o que aún no se conoce la “formula”- manejarlos con una tabla. Por cierto, saque pecho con los amigos si sabe lo que es la “presión de vapor”)

Converting the analog signal into pH

The Atlas Scientific Gravity™ Analog pH Sensor / Meter will output a voltage from 3.00V to 0.265V.

Equation to convert voltage to pH

$$pH = (-5.6548 * \text{voltage}) + 15.509$$



4 Copyright © Atlas Scientific LLC

AtlasScientific

Environmental Robotics

Figura 23. Tabla de sensor de pH (Atlas Scientific)

4. **Sensores que entregan la información en formato (casi) legible:** Corresponden, en general, con dispositivos miniaturizados que incluyen un ADC. En este blog usamos el DS18B20, que entrega directamente la temperatura en formato ASCII. (En realidad la lectura se obtiene mediante el concurso del sensor -con su ADC incorporado- y el software que controla el bus *oneWire* y deposita la lectura en un fichero. Ver 2.3.1, “Bus *oneWire*”).

5.2.- Clase sensor. Preliminar.

Comencemos con el análisis de la clase *sensor*. Quizá la primera acotación que hay que hacer, y volvemos al asunto de la correspondencia entre clases y objetos -de la vida real-, es que, aunque la clase se instancie una vez para cada sensor a utilizar, estas instancias se ocupan, básicamente, de convertir lo que entrega el conversor ADC al formato legible (*lectura*). Es una cuestión sutil, pero el código -de la clase *sensor*- no tiene nada que ver con el objeto físico *sensor*, sino con la información que, en definitiva, entrega el ADC y su conversión a formato legible (por otra parte, es difícil imaginar qué código cabría desarrollar asociado a un sensor que, en general, no es más que un poco de silicio o algo parecido sometido a temperatura, presión, etcétera).



Sobre el uso del término *sensor* para esta clase : No es el más adecuado. Como se verá casi de inmediato, en realidad se debería llamar “*sensorCanal*” (o algo así), en referencia a que cada instancia corresponde con un sensor -materialmente- diferente que, a su vez, puede estar conectado a diferentes tipos de ADCBus. Un LM35 conectado a un ADS1115 estará representado por un objeto de esta clase -sensor-, cuyos atributos serán la suma de los de un sensor de este tipo -de sensor *físico*- junto con los del chip ADS1115. Un LM35 conectado a un arduino necesitará un objeto -instancia- con atributos diferentes del que se instancia para el LM35 conectado a un ADS1115. Dos sensores tipo LM35 conectados a un mismo ADS1115 también estarán representados por dos objetos diferentes (en este caso el único atributo distinto será el número de canal al que están conectados -*self.channel*-).

Como se verá enseguida, la clase *sensor* tendrá pocos métodos, y, de ellos, solo un par imprescindibles: 1) el que calcula (o, mejor, se ocupa de que le calculen) la lectura a partir de lo entregado por el ADC y 2) el que, previamente, ha asignado al objeto -sensor- A) el *agente de cálculo* (el algoritmo con el que se transformará lo entregado por el ADC para el sensor en cuestión en la lectura), B) la tensión de referencia del ADCBus y C) la tabla de conversión -en su caso-. Más aún: con excepción de estos dos métodos el resto de la clase *sensor* tiene un cometido más que nada *administrativo*. Es, casi en su totalidad, un contenedor de datos -tipo de sensor, ganancia, ADCBus al que está conectado, tablas de conversión para sensores tabulados, etcétera-; el método *readout*, que calcula la lectura partiendo de lo entregado por el ADC, no tiene una sola línea de código que calcule nada: le pasa la papeleta al *agente de cálculo* que se haya elegido -instanciado- dependiendo del tipo de sensor y del ADCBus al que está conectado. Ver algo más adelante.

En el caso de la clase *sensor* tiene, pues, algo más de complicación el análisis de los atributos de la clase que el de los métodos. Hay que analizar qué datos son necesarios para cada sensor; para cada objeto de la clase *sensor*.

Son los siguientes:

1. El tipo del sensor: por ejemplo LM35.
2. El ADC al que está conectado (puede haber varios ADC en funcionamiento simultáneamente; incluso de diferentes tipos).
3. El canal del ADC al que está conectado (este atributo no es necesario para el cálculo del *readout*, pero sí para configurar el ADC. Ver 4.1.8, “ADS1115. Métodos para su configuración”).
4. La tensión de referencia (el *campo de entrada*) que se usará para medir.
5. La identificación, si se dispone de ella, del sensor.

El “ADC al que está conectado” es, en realidad, el objeto del *ADCBus* previamente instanciado, y que se pasa como un *argumento* al instanciar el sensor. Así -el objeto de- la clase *sensor* sabe en qué formato le llegan los datos de la conversión (en realidad, al pasar como argumento el *objeto* del *ADCBus*, lo sabe *todo* de él; más sobre este asunto dos párrafos más abajo). La necesidad de saber a qué canal está conectado el ADC es obvia, sabiendo que hay un conmutador de canales que conecta la salida del sensor conectada a un determinado canal al ADC.

La “identificación” del sensor es su *matrícula*. No todos los sensores tienen matrícula, de hecho esta información es, al menos a efectos de este blog, la que el propio bus *oneWire* proporciona para cada uno de los sensores que tiene conectados -y que el propio bus detecta al arrancar-. En este blog solamente estos sensores tendrán esta identificación.

Vayamos anticipando que hay una “tabla” (un *diccionario* de Python) en la que se almacenen las características de los sensores indexadas por el tipo (por ejemplo “LM35”). Esta tabla contiene las características comunes a todos los sensores de este tipo. Así, por ejemplo, al tipo LM35 se le hará corresponder en la tabla con los parámetros “100, 0, 1.0” que corresponden con la ganancia (100 °C/voltio), el *cero* (0 voltios -> 0 °C) y el valor máximo que cabe esperar que dé uno de estos sensores (1.0 voltios -> 100 °C); la información para un sensor tabulado contendrá, además, el nombre del fichero de texto que contiene la tabla propiamente dicha.

Es esta -la tabla con formato de diccionario de Python que contiene la información de los tipos de sensor-:

```
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'],
}
```

y que, como veremos, forma parte de la clase *sensor*.

Finalmente, el objeto -de la clase `sensor`- calculará y guardará como atributo la tensión de referencia que, de acuerdo con sus propias características y las del bus al que está conectado, usará este último cuando convierta este sensor (en realidad guarda la tensión y la posición de esta en la lista del bus; ver 4.1.9, “método `chooseVref` [tensión de referencia]; método ‘`setChannelGain`’; `refVoltageList = [0, 0.256, 1.024, 2.048, 4.096, 6.144]”).`

Para que la explicación del código de la clase `sensor` resulte más clara es mejor diferirla hasta el punto 5.4. “Instanciación de la clase ‘`sensor`’. Instanciación de las clases ‘`agente`’”. Puede ir a este punto si quiere echar un vistazo, pero le recomendamos que estudie primero todo lo que viene a continuación.

5.3.- Algoritmos de cálculo de la lectura. Clases “`agent`”.

Todos los sensores LM35 son iguales entre sí (y todos los LM50 entre sí, y todos los DS18B20 entre sí). Quiere decirse que, con independencia del conversor ADC al que estén conectados, todos -los LM35- dan 1 Volt. por cada 100 °C (ó 10 mV por °C, como prefieran). En realidad se pueden usar sin ordenador ni conversor ADC ni nada: si dispone de una fuente de tensión entre 3 y 5 voltios -corriente continua- y un tester basta con conectar la fuente entre los cables negro (-) y rojo (+) y tendrá la temperatura en el cable amarillo; basta con medir la tensión con el tester y multiplicar por 100.



No todos los sensores del mismo modelo son *completamente* iguales entre si, y aunque es cierto que suelen ser muy parecidos, no es menos cierto que para que la indicación sea correcta pueden necesitar un ajuste fino del factor de conversión (ganancia), o una calibración, si se quiere que sea intercambiable.

Por otra parte, no todos los ADC funcionan igual: el ADS1115 entrega la lectura en complemento a 2 (16 bits), mientras que un arduino “ProMini” da la lectura en 10 bits “directa” (0,0 V. -> 00 0000 0000; 5,0 V. 11 1111 1111). Lo anterior está simplificado; suponemos que la tensión a la que está alimentado el arduino son 5,0 V: en este caso tampoco 11 1111 1111 correspondería exactamente con 5,0 V, sino con $5.0 * [1023/1024]$.

A donde queremos llegar es a que, estando claro que la clase -los *objetos* de la clase- `sensor` tiene que tener los atributos ya mencionados -ganancia y cero del sensor; 100 y “0” para el LM35 por ejemplo- para poder calcular la lectura, el algoritmo de cálculo no depende solo del sensor, sino también del ADC.

Volvemos al concepto de *encapsulamiento*: a los objetos de la clase `sensor` les vamos a pedir que calculen la lectura (pasaremos como argumento lo que nos da el `ADCBus`), de forma que tienen que tener el recurso que les permita hacerlo. Por otra parte, el algoritmo de cálculo tampoco pertenece a un sensor concreto: la parte *difícil* -convertir la información binaria a decimal; incluyamos también en *difícil* cambiar la coma a la lectura de los sensores del `oneWire`- viene determinada por el ADC -por el formato en el que da la lectura-, pero otro sensor puede que tenga otra ganancia, o diferente cero (el LM50, por

ejemplo; ver 5 “Clases para sensores. ‘Agentes’ de cálculo.’, Sensores ‘cuasi-proporcionales””).

Conviene, por tanto, ir pensando en una(s) clase(s) -agente(s) de *cálculo*- cuyos objetos sean capaces de gestionar todo lo anterior: por ejemplo, una clase *binaryAgent* nos proporcionará objetos que serán capaces de calcular la lectura entregada, por ejemplo, por el ADS1115 para un canal de este tipo de ADC; ni que decir tiene que, de alguna forma, el agente sabrá cuales son el resto de parámetros necesarios, los del sensor.

5.3.1.- Agente de cálculo “binario” (*binaryAgent*). Herencia.

El ADS1115 y el arduino dan la conversión en formato binario, pero diferente entre ellos (16 bits en complemento a 2 el primero y en 10 bits sin signo el segundo). Hay, por otra parte, decenas de conversores ADC que se pueden conectar a un raspberry, unos darán la conversión en 10 bits, otros con 12, 13, 14, 15 o 16 (y los hay hasta de 24: cualquier persona con unos mínimos conocimientos en el campo de adquisición de datos arruga el cejo para cualquier cosa por encima de 16 bits; 20 ya son un lujo. 24 una exageración); la mayoría entregarán los datos en complemento a dos, aunque el formato directo -sin signo- es también frecuente.

Vemos, por tanto, la necesidad de un primer algoritmo que, en función de un puñado de parámetros -ver a continuación-, calcule el valor legible -la *lectura; readout*- correspondiente a la información binaria que nos proporciona un conversor ADC. Dicho algoritmo -el lector ya se está imaginando que el “algoritmo” es (parte de) lo que venimos llamando *agente de cálculo*- contiene -atributos- los siguientes parámetros:

1. La cantidad de bits (16, por ejemplo, en el caso del ADS1115).
2. Si admite o no signo (Sí, en el caso del ADS1115; no, en el caso del arduino)
3. La ganancia, el cero y la tensión de referencia empleada para cada sensor concreto (100, “0” y 1,024 V en el caso del LM35 “encapsulado” conectado a un ADS1115, si no entiende lo del 1.024 vaya a 4.1.9, “método chooseVref [escoger tensión de referencia]”).

Se verá más adelante que el código de esta clase -*binaryAgent*- es independiente del bus concreto al que esté conectado el sensor que la invoque -instancie- y también del sensor propiamente dicho. Es, en ese sentido, “universal”; al menos lo es para todos los sensores *lineales* -la tensión que entregan varía linealmente con lo que miden- cuya información venga en formato binario. Vayamos adelantando que cada vez que se instancie un sensor para incluirlo en el sistema de medición -*aplicación*- que esté usando esta librería, la propia instancia del sensor instanciará a su vez el agente de cálculo que necesita para calcular su lectura.

Se incluye a continuación el código de esta clase, cuya interpretación -a excepción del método *__init__*, que es harina de otro costal-, si el lector tiene unos conocimientos mínimos de álgebra binaria, no debería presentar ningún problema. Sea como sea, le recomendamos que lo lea como mucho -y de momento- *en diagonal*. Es mejor que antes de entrar en más profundidades se lea la parte correspondiente al análisis de las otras

clases *agente* -“Agente de calculo “ASCII” (ASCIIAgent)” y “Agente de calculo “tabulado” (tabulatedAgent)”, lo que le llevará con más facilidad al siguiente punto (5.3.3., “Clase ‘calculationAgent’”); ya habrá tiempo de volver atrás para consultar el código a continuación con más detalle. Seguro.

```
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)
```

El método `binaryToDecimal` no tiene más misterio que el algoritmo propiamente dicho.

El método `__init__`, sin embargo, sí -tiene misterio-. Es la primera vez en el blog que nos topamos con una clase que hereda de otra; en este caso la clase `calculationAgent`; la creación de la clase lo explica: “class `binaryAgent(calculationAgent)`”.

Hagamos una primera aproximación a la cuestión de la *herencia*. Habrá -hay- una clase (`calculationAgent`) que tiene atributos -datos y métodos- que atan a la clase heredera, `binaryAgent` en nuestro caso. Echemos un vistazo preliminar -también en *diagonal*- al código de la clase *ancestro -parent-*:

```
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
```

es, por así decirlo, *la otra mitad* de `binaryAgent`: ésta no tiene, aparentemente, atributos; de eso se encarga -de definirlos e inicializarlos- `calculationAgent`.

(El código de la clase *calculationAgent* se comentará con detalle en el apartado 5.3.3, “Clase *calculationAgent*”; aquí simplemente se anticipa para complementar la explicación del *binaryAgent*).

El asunto -de la herencia- es, en parte, el siguiente:

- La clase heredera -*binaryAgent*- hace suyos todos los métodos y definiciones de atributos de la clase ancestro -*calculationAgent*-.
- Los métodos que también aparecen en la clase heredera sobreescriben -suplantan; *override*- los de la clase ancestro, en nuestro caso `__init__` y `convert`.
- Es frecuente que la clase ancestro sea una clase “abstracta”: estas clases solamente contendrían métodos vacíos (no es este caso). Sus funciones son múltiples, algunas muy sútiles y cuya explicación queda claramente fuera del alcance de este blog. Una de ellas es definir métodos que alguna de las clases herederas pueden no usar pero que, debido a la arquitectura de la aplicación, pueden ser invocadas -aunque no hagan nada-.
- En este caso, el método `__init__` del heredero necesita que se ejecute el del ancestro: lo invoca tal cual (“`calculationAgent.__init__[self, Vref, gain, zero, bits, twosC, None]`”); como el agente binario -*binaryAgent*- no usa tabla, sustituye la que eventualmente se le haya podido pasar como argumento por “None”. Una instanciación cuidadosa de la clase debería no usar este agente con una tabla con contenido como argumento: se propondrá en el anexo correspondiente la detección de este error como ejercicio; en nuestro caso, tal y como está escrito el código este error -haber pasado una *tabla-tabla* como argumento al instanciar *binaryAgent*- no tendría ningún efecto. Haga caso omiso, de momento, de la línea `self.name = 'binary'`.



(a) Escriba una aplicación de *cliente* para usar la clase *binaryAgent*: el objetivo es obtener la lectura correspondiente a un valor -en código binario- positivo de 10 bits (no en complemento a dos) cuyo fondo de escala es 5.0 ($V_{ref} = 5$; $0 \rightarrow 0$, $1024 \rightarrow 5.0$). Escriba un programa que calcule la lectura para los valores de entrada siguientes: 2, 5, 10, 20, 50, 100, 200 y 1000.



(b) Al final del punto 4.1.7, “Instanciación (uso) de la clase ADS1115”, se planteó la posibilidad de añadir el código correspondiente para convertir la lectura del ADC al formato legible. Añada al código propuesto en ese punto para la instanciación del bus y la lectura del canal el código necesario para la conversión de la lectura (Sugerencia: cree una instancia del *binaryAgent* y úsela -“`convert(value)`”- para generar la lectura)

5.3.2.- Agente de calculo “ASCII” (ASCIIAgent).

Ya hemos adelantado más arriba (ver 2.3.1, “Bus oneWire”) que el bus oneWire es un caso (muy) especial de conversor analógico digital. En realidad no es un ADC, sino tantos ADC como sensores tiene conectados; por un procedimiento bastante alambicado (y muy lento), el *driver* de este bus deposita las lecturas ya convertidas de los sensores que están conectados al bus en ficheros de texto: el texto que contienen los ficheros es, además de mucha otra información, la lectura del sensor (por ejemplo la temperatura en el caso del DS18B20) aunque con la coma fuera de sitio (en el caso del DS18B20 lo da en milésimas de grado centígrado, es decir, 23456 para 23,456 °C).

Se necesita, por tanto, un *agente de cálculo* para este supuesto; se tratará de un algoritmo trivial, toda vez que lo único que tiene que hacer es cambiar la coma de sitio.

```
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)
```

A poco que haya estudiado el punto anterior, el código de esta clase le empezará a resultar familiar: el `__init__` -salvo por el `self.name = 'ASCII'`; seguro que ya le va *pillando el tranquillo* a la cosa- es idéntico, y el método `convert` casi. Cambia el núcleo del asunto: el método del algoritmo; por lo demás, todo lo dicho en el punto anterior es válido para este.

5.3.3.- Clase *calculationAgent*.

Hemos visto en los dos puntos anteriores como los dos agentes de cálculo *binaryAgent* y *ASCIIAgent* contienen, por así decirlo, lo estrictamente necesario para ejecutar la parte de la tarea que los diferencia entre sí; tómese la molestia de compararlos. Si la librería que estamos desarrollando no tuviese necesidad más que de uno de ellos (si solamente tuviésemos que *lidiar* con el ADS1115 bastaría el *binaryAgent*, si el proyecto contemplase el bus *oneWire* nada más el *ASCIIAgent* habría bastado) probablemente no existirían como tales, y su función se implementaría como un método más de la clase *sensor*.

Pero, aparte de que sí tenemos, de momento, dos algoritmos de cálculo de la lectura diferentes, hemos decidido diseñar la arquitectura de la librería con *crecederas*.

En esta tesisura, necesidad de varios algoritmos de cálculo de la lectura en función de sensores y conversores ADC, hemos decidido diseñar una familia de *agentes de cálculo*, ver figura 24. Una clase *calculationAgent* será el *ancestro* común a todas (excepto *tabulatedAgent*, pero no se detenga en este detalle de momento). *calculationAgent* contendrá todos los atributos y métodos que comparten sus *herederas* aunque, como se explicará un poco más adelante, hay un sólo método -común-, *convert*, del que en esta clase se hace poco más que enunciarlo.

El código de esta clase es el siguiente (y ahora sí que se necesita que el lector lo estudie con detenimiento; después del código hay una explicación detallada).

```
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():
```

La clase se instancia como una clase sin *ancestros*: “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
```

Es el método *constructor* de la clase. No hace otra cosa que asignar a los atributos de la clase los argumentos que se le pasen (si no se pasa ninguno para “table” le da el valor *None* -*ninguno/a-*; “ ... table=None”). Al ser -casi: ver más adelante explicación al respecto para la clase *tabulatedAgent*- todas las clases *agente de cálculo* herederas directas o en “segunda generación” de esta, estos atributos están definidos para todas ellas (en realidad, se definen al invocar el método *__init__* de *calculationAgent* desde el método *__init__* de cada una de las herederas; ahora ya puede ir volviendo sobre el código de los *agente(s) de cálculo* ya explicados e irá entendiendo mejor alguna instrucción que, hasta el momento, parecía incomprensible).

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

El método *convert* -su existencia, no su contenido- lo comparten todos los *agente(s) de cálculo*. Cómo es diferente en todos ellos, en la clase *ancestro* nos limitamos a enunciarlo (no hace nada: -*pass*-); cada clase definirá su propio método *convert* que sustituirá -*override*- al de la clase *ancestro*.

Podemos analizar (ahora ya sí: es este asunto de la *herencia* entre clases tan rebuscado - como imprescindible, eso sí- que no hay otra que ir adelante y atrás un par de veces para entenderlo bien) el código de los constructores (métodos *__init__*) de los agentes “de primera generación” (los de las clases *binaryAgent* y *ASCIIAgent*):

- Los métodos *__init__* de los agentes *binaryAgent* y *ASCIIAgent* son similares: ambos invocan -*calculationAgent.__init__(...)*- el *__init__* de la clase *calculationAgent* pasándole los argumentos que, a su vez, han recibido del *cliente* que los instancia (la clase *sensor*, en esta librería).
- El mero hecho de incluir como argumento en la definición de una clase “*calculationAgent*” (“class *binaryAgent[calculationAgent]*:” en 5.3.1) hace que la clase así definida -*binaryAgent*, en este caso- herede lo definido en la clase *ancestro -calculationAgent-*. No es mucho lo que heredan, de momento, las clases *binaryAgent* y *ASCIIAgent* del *ancestro* común, pero todo se andará.

Cambiando de *tercio*, que no de clase, parece oportuno resumir los atributos comunes a las clases *agente(s) de cálculo*. Aunque aún no se ha hablado de ellas, metemos de *matute* dos clases más -*binaryTabulatedAgent* y *ASCIITabulatedAgent*-: es posible que el lector se malicie de qué van, aunque de momento sólo se trata de resaltar cómo los atributos se repiten entre clases:

Atributo	Descripción	binaryAgent	ASCIIAgent	binaryTabulatedAgent	ASCIITabulatedAgent
Vref	<i>Voltage reference</i> (valor máximo esperado para este sensor): es la referencia de tensión que ha usado el ADC para medir este sensor. Es necesario para calcular el <i>readout</i> , ya que la lectura del ADC (el valor binario que éste entrega) está referido a este valor.	X	-	X	-
gain	Ganancia: Es la constante por la que hay que multiplicar (o dividir) el valor que entrega el ADC para calcular la lectura del sensor (por ejemplo, "100" para el LM35)	X	X	X	X
zero	Cero: es la tensión que corresponde con el cero de la lectura (por ejemplo, el LM50 da 0,5 V para 0 °C: el valor de zero para este sensor es 0,5)	X	-	X	-
Bits	Cantidad de bits del ADC (16 en el caso del ADS1115)	X	-	X	-
twosC	Valores negativos: indica si el ADC puede entregar valores negativos (habitualmente en "complemento a 2").	X	-	X	-
table	Tabla de conversión (objeto de la clase sensorTable. Ver 7, "Tablas de sensores [sensorTable]").	-	-	X	X

Las clases *agente de cálculo* comparten -no todas todos- los atributos. Si no existiese la posibilidad de hacerlas -a las clases *agente de cálculo*- herederas de una clase común *calculationAgent*, habría que repetir en cada una de ellas la definición de estos atributos.

(No solamente esto, si se decide hacer algún tipo de verificación sobre los atributos -la ganancia no puede ser cero, tampoco el número de bits- es muy conveniente que aquellas -las comprobaciones- estén "centralizadas", es decir, como métodos de la clase *calculationAgent*. Esto también se andará).

5.3.4.- Agente de cálculo “tabulado” (*tabulatedAgent*).

Hay sensores cuya salida no es ni proporcional ni lineal con el parámetro medido (ver 5.1, “Sensores. Sensores sin linealidad [tabulados]”). En este caso necesitamos un algoritmo - cuya lógica no tendrá ninguna similitud con ninguno de los agentes previos analizados- al que se le pase un valor y nos devuelva el valor correspondiente a partir de una tabla que describa su comportamiento. Lo llamaremos *tabulatedAgent*.

La tabla no se pasará como tal (lista o *tuple* de Python) en la creación -instanciación- del agente. A tal efecto se definirá una nueva clase (ver 7, “Tablas de sensores [*sensorTable*]”) que tendrá como objetivo justamente la gestión (lectura desde archivo de texto, verificación, consulta) de la tabla; lo que se pasará al agente será la instancia de esta clase que corresponda.

Las tablas contienen valores sueltos -“discretos”- del comportamiento del sensor, por lo que para calcular los valores intermedios hay que hacerlo con algún criterio (haga la prueba con excel: meta una tabla, conviértala en gráfico y añada una “línea de tendencia”; hay una docena de posibilidades). En este blog optaremos por la más modesta de todas, que es la *interpolación lineal*: en *sensorTable* se implementa esta solución, que necesita de procedimientos puramente geométricos.

La clase *tabulatedAgent* entra a formar parte de la familia de *agente de cálculo* de forma algo espúrea: la podríamos haberla hecho, también, heredera de la clase *calculationAgent*, pero esto habría tenido el inconveniente de hacerla cargar con demasiados atributos (¿*apellidos*?) que no le valen para nada: si se la quisiera usar independientemente del resto de la librería su instanciación rozaría lo incomprendible (“agente = *tabulatedAgent*(0, 0, 0, 0, 0, ‘NTC’)”), por ejemplo). Decididamente es mejor que se incorpore a la familia como una especie de *tío político*, sin tanto atributo.

El código de la clase es:

```
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():
```

La clase *no* se instancia como heredera de nadie.

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

se define un atributo `self.table` que es el *objeto* de la clase `sensorTable` y que ha sido pasado como argumento en la instanciación del agente de cálculo.

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

para extraer el valor de la tabla que corresponde con `value` se usa el método `lookup` del propio objeto `self.table`.

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

por homogeneidad con el resto de los *agentes de cálculo* se conserva el método `convert` que, simplemente, hace uso del método `self.lookupTable` para calcular la lectura.



Escriba la instanciación de la clase `tabulatedAgent` tal y como se ha definido la clase (o sea, no “agente = tabulatedAgent(0, 0, 0, 0, 0, ‘NTC’)”). Suponga que ya se dispone de un *objeto* de la clase `sensorTable` llamado “NTC”.

5.3.5.- Herencia múltiple. Clases “binaryTabulatedAgent” y “ASCIITabulatedAgent”.

En un proyecto como el que tenemos entre manos -y casi siempre- lo que ocurre si hay un sensor tabulado (por ejemplo el NTC de la placa hecha con el ADS1115, ver figura 19), es que éste está conectado a un canal del ADCBus, entregando por tanto una lectura en un determinado formato (binario, ASCII, ...) que hay que convertir a decimal -como la de cualquier otro sensor- para, después, pasársela al `tabulatedAgent` y que éste la pase por la tabla para calcular la lectura definitiva.

Si no estuviéramos diseñando la librería para uso general -y/o no nos hubiésemos empeñado en complicarlo un poco: ver arranque del punto 5, “Clases para sensores. Agentes de cálculo”- es muy probable que nos hubiésemos limitado a diseñar una clase *ad hoc* (“NTC_ADS1115”, por ejemplo); es lo más fácil.

No es el caso: queremos hacerlo de acuerdo con la esencia de la OOP, por lo que crearemos una clase que *herede* de dos de las que ya disponemos. No es difícil adelantar que serán las clases `binaryAgent` y `tabulatedAgent`, a partir de las que crearemos la clase `binaryTabulatedAgent`. Este es el código:

```
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))
```

Bien, parece que tomarse la molestia de hacer -de entrada- las cosas algo más complicadas empieza a rendir beneficios. No se puede negar que con muy poquitas líneas tenemos una clase que sirve para hacer tareas de cierta complejidad. Veamos:

Clase *binaryTabulatedAgent*. Análisis del código.

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

Es la definición de la clase. Como argumentos figuran las clases *ancestro*, que pueden ser tantas como se necesiten. En este caso la clase hereda de dos que ya están definidas, *binaryAgent* y *tabulatedAgent*, y esta vez ya si que la herencia tiene más *chicha*: tiene definidos, además de los necesarios atributos (*Vref*, *gain*, etcétera), todos los métodos de sus dos *ancestros*. *Sabe*, por tanto, convertir una lectura en formato binario a decimal, y *sabe* pasársela por la tabla para convertirla a su valor final.

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

El *constructor* de esta clase usa los de los *ancestros* a los que pasa los parámetros que, a su vez, se le han pasado a ella cuando se la ha instanciado; se crean así, en nuestro caso, todos los atributos necesarios. Observe que mientras que en la llamada al *__init__* de *binaryAgent* *self.table* tomará el valor *None*, en la llamada al *__init__* de *tabulatedAgent* ese valor pasará a ser el argumento pasado como *table* en la instancia de *binaryTabulatedAgent*. (Ale, coja aire y vuelva a leerlo un par de veces: si sigue sin entenderlo del todo no se preocupe, quédese con la música, siga y vuelva con más calma cuando haya acabado, por ejemplo, todo lo referente a los *agentes de cálculo*; poco a poco se va comprendiendo)

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

Este -nuevo- método *convert* es el que mejor sintetiza la ventaja de la herencia, múltiple en este caso. La clase ha heredado todos los métodos de sus *ancestros*: *sabe*, por tanto, convertir una lectura en formato binario a decimal (*self.binaryToDecimal(value)*) y *sabe* calcular la lectura final usando la tabla (*self.lookupTable(...)*).

Este método *convert* sustituye a los posibles métodos *convert* que puedan tener sus clases *ancestro* (por cierto, en este caso las dos. Si no se hubiese definido un método *convert* específico para esta clase, habría prevalecido el del *ancestro* más “a la izquierda” en la definición de la clase: el de la clase *binaryAgent*, en este caso).

El uso del mismo nombre *-convert-* para todos los métodos *agente de cálculo* que convierte la información “cruda” *-value-* a la lectura final no es casual. Aunque un poco traído por los pelos, esta elección corresponde con otro de los paradigmas de la OOP, el *polimorfismo*. Se trata de que el mismo nombre de *-de un método, de un atributo-* designe contenidos diferentes (o, incluso, algoritmos diferentes, como es el caso).

Bien. Lo anterior no es todo lo que hay que saber sobre herencia *-múltiple.-*, pero es mucho. El código de la clase “ ASCIITabulatedAgent” es el siguiente:

```
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))
```

cuya interpretación no difiere en nada de la de la clase anterior.

5.3.6.- Familia de clases agente de cálculo. Resumen. Conclusiones.

El siguiente diagrama resume todo lo anterior:

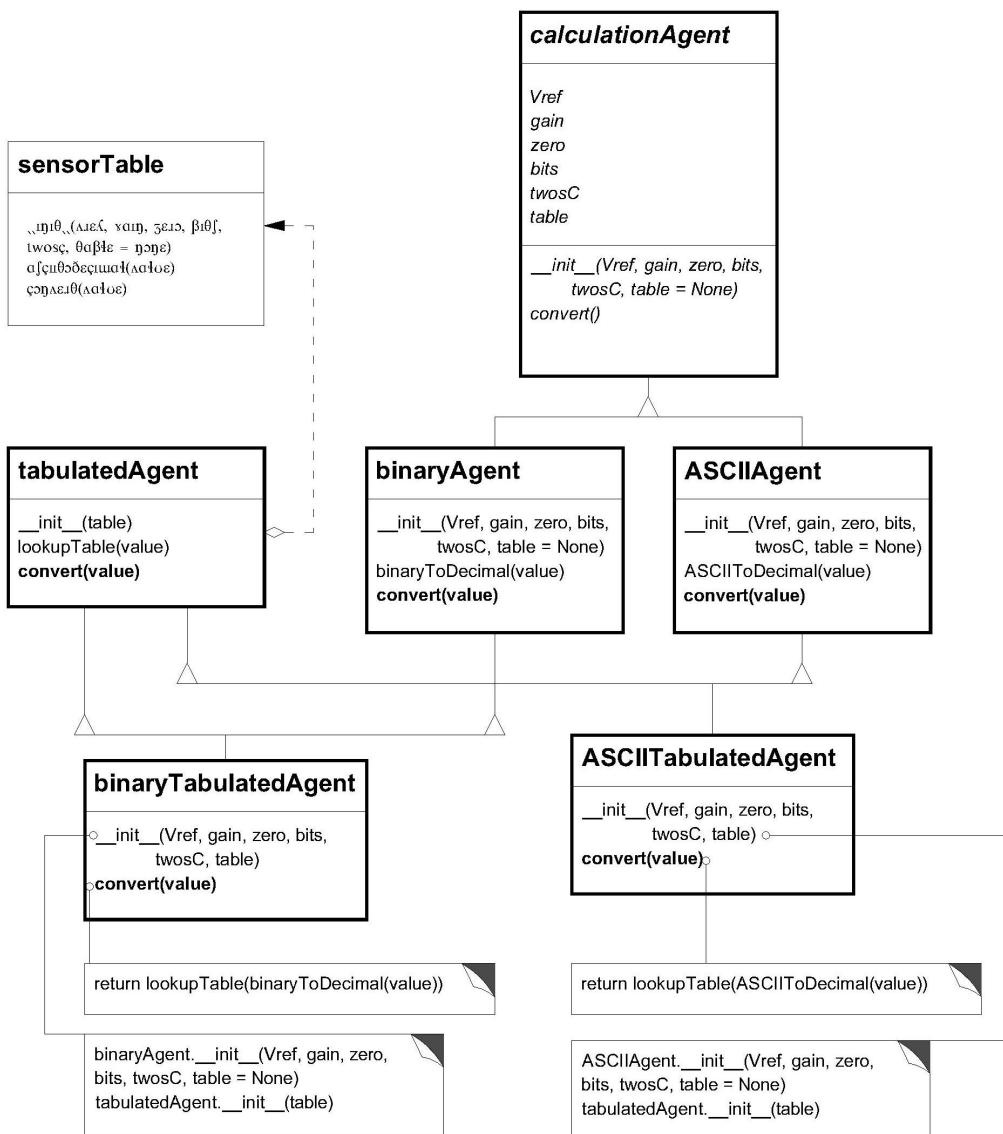


Figura 24. Diagrama de la familia agentes de cálculo

- La librería que se está desarrollando resuelve el problema de cálculo de la *lectura* correspondiente a un determinado sensor conectado a un determinado ADC por medio de una familia de clases *agente de cálculo* que implementan los algoritmos necesarios según el formato del conversor ADC y las características propias de cada sensor. Para implementar las clases que alojan los algoritmos se usa la técnica de herencia -*inheritance*- propia de la OOP.

- Una clase *calculationAgent* oficia de ancestro común para todas las clases de la familia (excepto *tabulatedAgent* que, sin embargo, si es ancestro de las dos clases de la *última generación*). Esta clase no contiene ninguno de los algoritmos de cálculo; simplemente aloja los atributos comunes a todas las clases (y que son necesarios, sin embargo, para el cálculo propiamente dicho de las lecturas). En una versión ampliada de la librería centralizará, al menos, las verificaciones de los argumentos pasados para la instanciación de los agentes a efectos de control de errores.
- Dos clases, ***binaryAgent*** y ***ASCIIAgent***, implementan los algoritmos básicos (información en formato binario y ASCII, respectivamente).
- La clase ***tabulatedAgent*** implementa un algoritmo (realmente el algoritmo está delegado en un objeto de la clase *sensorTable* con el que *colabora*) simple de consulta en una tabla que contiene valores discretos (pares de puntos). Se trata de poder manejar con la librería bien sensores que no son lineales (NTC), bien sensores cuya lectura interesa que se transforme en algún parámetro que, estando directamente relacionado con la lectura que entrega el sensor -y convierte el ADC-, no tiene -el parámetro- una relación lineal con la lectura del sensor. Esta clase es instanciable *per se*, y serviría para implementar una consulta en una tabla (en el contexto de la conversión ADC o en cualquier otro).
- Dos clases herederas por partida doble, ***binaryTabulatedAgent*** y ***ASCIITabulatedAgent***, implementan los algoritmos de las tres clases descritas en los dos puntos anteriores, lo que, en la práctica será lo más habitual (supuesto que la lectura del sensor necesite de una tabla para el cálculo de la lectura).
- La técnica empleada permite que algoritmos complejos queden definidos con clases muy simples.
- La clase *calculationAgent* es una **clase abstracta**: no se puede instanciar (o, mejor, sí se podría, pero no sirve para nada). Una utilidad de este tipo de clases es enunciar los métodos comunes a todas las herederas (el *interface* de la clase); en este caso solamente el método *convert*.
- ***convert***: todas las clases *agente de cálculo* tienen un método *convert* cuya utilidad es exactamente la misma: convertir el valor pasado a la lectura. Esta técnica - propia de la OOP- se conoce como *polimorfismo*: la misma orden usa diferentes algoritmos que, a su vez, están *encapsulados* en diferentes objetos cuyas clases han definido métodos distintos con el mismo nombre.
- **Explosión de clases**. La técnica empleada para el desarrollo de esta familia de clases -herencia; inheritance- puede dar lugar en ocasiones a lo que se denomina “explosión de clases”. Queda fuera del alcance del blog una explicación detallada del término -y del *fenómeno*-; digamos que tiene que ver con las -muchas- combinaciones que podrían ir apareciendo si hay más algoritmos que implementar además de los que ya lo están. Supongamos que ademas del formato -binario, ASCII- y de la posibilidad de que haya -o no- que emplear una tabla, existiese una tercera característica en la conversión, por ejemplo que el resultado haya que multiplicarlo por dos -es una *bobada* improbable, pero sirve para ilustrar lo que

pasaría-. En este caso, siguiendo la estructura que hemos empleado, habría una clase (*doubleAgent*, o algo así) que multiplicaría por dos la lectura, y necesitaríamos las clases *binaryDoubleAgent*, *ASCIIDoubleAgent*, *binaryTabulatedDoubleAgent*, *tabulatedDoubleAgent*, *ASCIITabulatedDoubleAgent*, ... para implementar los correspondientes algoritmos. Es evidente que no es funcional, de modo que no se recomienda esta técnica si no hay una seguridad razonable de que no pasará algo así (porque no hay tantos algoritmos de cálculo a la vista, en el caso que nos ocupa: librería para adquisición de datos).



La intensidad del ruido ambiente se mide en decibelios (db). Se trata de una unidad de medida cuya definición es, a muchos efectos, convencional, por lo que para su medida son necesarios aparatos de cierta complejidad. Se dispone de un aparato antiguo que entrega una señal en voltios que varía en función del ruido ambiente, pero ésta no se puede convertir -la medida en voltios- según una función lineal a db.

Un laboratorio ha calibrado el dispositivo, llegando a una relación aproximada entre la tensión -voltios- que entrega y el ruido ambiente medido en db. Esta es:

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

Estudie qué modificaciones (o adiciones) son necesarias para adaptar la librería “EIDEAnalog” a este dispositivo.

(Sugerencias. Hay varias soluciones; se propone crear un *agente de cálculo* al efecto (*?decibelsAgent?*). En este caso se propone un uso “espúreo” de los parámetros generales de las clases; por ejemplo, *Vref* para el valor “0.9”, *gain* para “4.2”).

5.4.- Código de la clase sensor. Instanciación.

Una vez planteadas las clases para los buses (punto 4) y resuelto el rompecabezas de las clases *agente de cálculo* (punto 5.3) toca enmarcarlo todo con la clase sensor para que el conjunto sea funcional.

La clase *sensor* es el ombligo de la librería: en la instanciación de cada sensor se pasa su tipo y el *bus* al que está conectado, éste como un *objeto* de la clase *ADCBus* que, obviamente, tiene que haber sido previamente instanciado. Con los atributos del bus y los datos del sensor -con el *tipo* de éste se extraen de un diccionario (ver un poco más abajo) - se *rellenan* los atributos del objeto -de la clase *sensor*-, que se usan a su vez para crear sobre la marcha sendas instancias de las clases *sensorTable* -si procede; sensores tabulados- y del *agente de cálculo* que corresponda -esta *siempre* procede- y que quedan vinculadas al objeto de la clase *sensor* como un atributo más. La siguiente figura es un esquema de lo anterior.

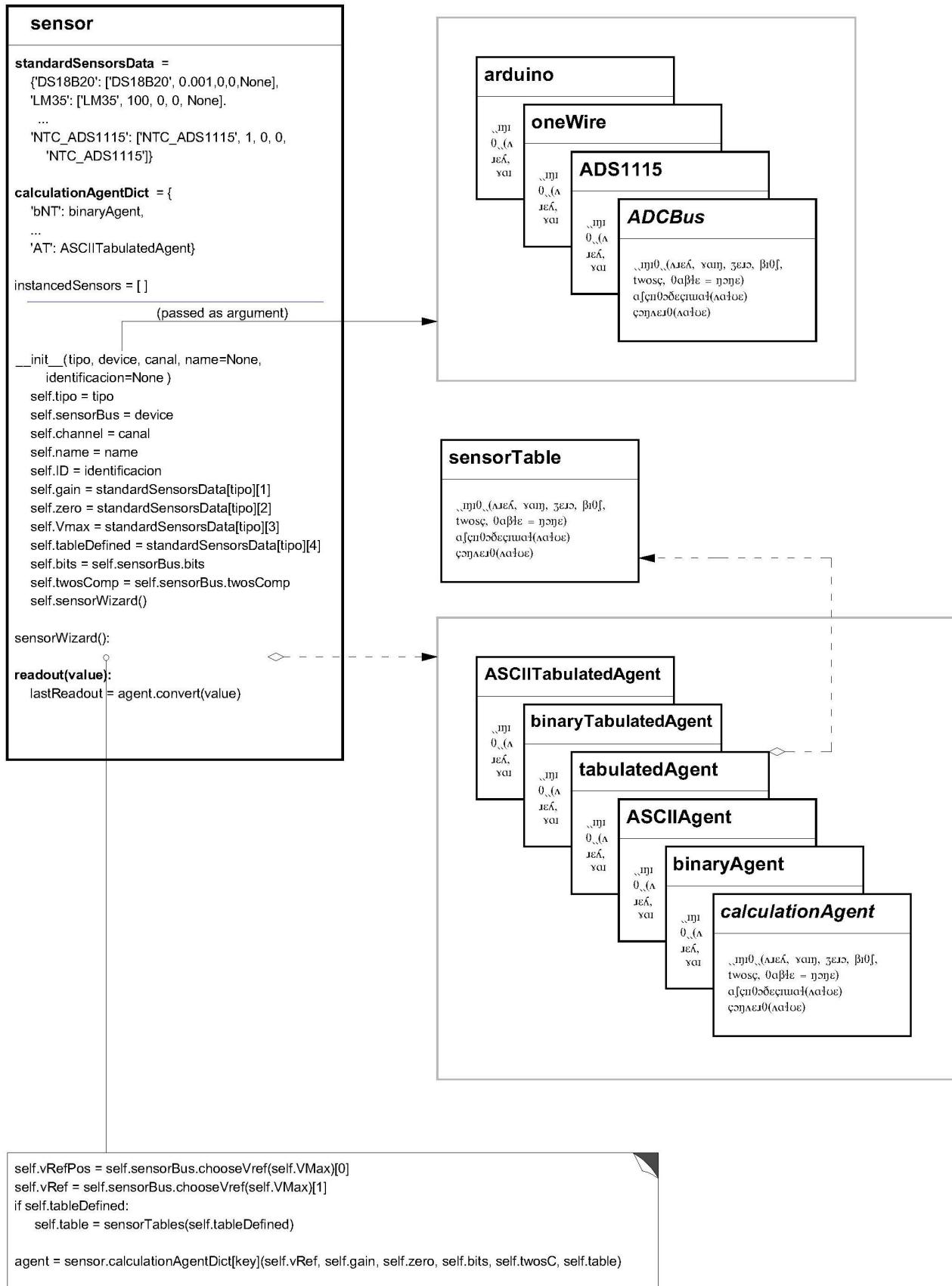


Figura 25. Diagrama de la clase sensor

No es fácil de asimilar a la primera, así que, aún con riesgo de repetir cosas, vamos a comentarlo un poco más:

- Tenga en cuenta, se insiste, que la clase `sensor` no representa un tipo de sensor determinado, sino un *tipo de sensor determinado* conectado a un *ADCBus determinado* (ver nota en 5.1, “Sensores. Preliminar” a este respecto). No hay una clase `sensor` que, instanciada, represente un tipo de sensor determinado; en esta librería esa función está recogida en el diccionario `standardSensorsData` (ver un poco más abajo), que, eso sí, es un atributo de la clase `sensor`. Parece un trabalenguas (y, en cierto modo lo es), pero hasta que no lo entienda es mejor que no siga.
- Para instanciar un sensor hay que haber instanciado previamente al menos un bus, y pasar esa instancia -se repite, *instancia-* del *ADCBus* como argumento al instanciar el sensor; son necesarias ciertas características del bus -formato de datos- al que está conectado el sensor para que éste pueda instanciar a su vez el *agente de cálculo* que necesita.
- En la instanciación del sensor el objeto que se está creando crea a su vez una instancia de la clase `sensorTable` si al *tipo de sensor* hay asociada una tabla. Por ejemplo, como se ve en el diccionario `standardSensorsData`, el sensor ‘ADS1115_NTC’ tiene asociada la tabla del mismo nombre; el ‘LM35’, también por ejemplo, no tiene tabla asociada. La instancia de `sensorTable` pasa a ser un atributo más del objeto que se está instanciando (otro trabajuelguas). A esta clase de colaboración entre objetos se le llama *agregación -aggregation-*. Puede olvidarse de que se llama así para continuar. Incluso del propio concepto puede quedarse, nada más, con la idea intuitiva de que un objeto de una clase puede ayudarse de las habilidades de los objetos de otra clase.
- De forma algo más alambicada, el objeto de la clase `sensor` también instancia un objeto de uno de los *agentes de cálculo* del punto 5.3, el que necesite en función de su tipo -de sensor- y del bus al que está conectado. Para esto necesita la información contenida en el diccionario “`calculationAgentDict`” que es un atributo de la clase `sensor` (lo mismo que el diccionario `standardSensorsData` de más arriba). Esto también es una *agregación* (lo de instanciar una *clase agente de cálculo*).
- La colaboración entre el objeto de la clase *ADCBus* que se ha pasado como argumento para instanciar el sensor y la instancia de éste -del sensor- es, funcionalmente, idéntica a las mencionadas para la tabla y el agente de cálculo, aunque su implementación -se pasa como argumento para instanciar el sensor- es un poco menos ortodoxa. Dado que los métodos y atributos del objeto *ADCBus* quedan a disposición del objeto `sensor`, es, también, una *agregación*, aunque, a diferencia de los dos anteriores, el objeto *ADCBus* no es exclusivo del objeto `sensor` al que se la ha pasado como argumento, sino que será compartido por todos los sensores que tenga conectados (el *ADCBus*).

Veamos, en definitiva, el código de la clase sensor:

```
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

```

Clase `sensor`. Análisis del código.

```

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

```

La clase `sensor` no tiene *ancestros*

```

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'],
}

```

Usamos los *atributos de clase* de la clase `sensor` para almacenar un diccionario con los datos esenciales de los sensores que tiene -o puede tener- conectado el proyecto. Este será uno de los puntos de la librería que será necesario modificar para añadir nuevos sensores; las sucesivas entradas del diccionario contienen la siguiente información: nombre del sensor, ganancia, cero , tensión de referencia y tabla. Estas, las tablas, son objetos de la clase `sensorTable` (Ver 7., “Tablas de sensores [`sensorTable`]”).


Es posible que dos sensores idénticos se empleen con diferentes características: un ejemplo sería dos sensores tabulados materialmente iguales pero con diferente tabla; basta con definirlos con dos nombres diferentes en `sensor.standardSensorsData`. Pasa lo mismo si, por ejemplo, la tensión de referencia es diferente: es necesario definir dos sensores distintos en la lista.

```

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

```

```

'ANT': ASCIIAgent,          # ASCII; not table
'AT': ASCIITabulatedAgent,  # ASCII; table
}

```

Este diccionario se incluye también como atributo de clase; hace referencia a las clases *agente de cálculo* (ver 5.3, Algoritmos de cálculo de la lectura. Clases ‘agent’”). Sirve, como se explicará más adelante -sensorWizard-, para la instanciación del agente de cálculo que necesite el sensor.

```
InstancedSensors = []
```

Es una lista en la que se almacenarán los objetos de los sensores instanciados. Se usará en las clases de uso de la librería. Puede ignorarla de momento.

```

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

```

Es este método `__init__` uno de los más importantes, y complejos, de la librería. Además de 1) inicializar los atributos de la clase necesarios para identificar el sensor, el ADCBus al que está conectado y a qué canal de éste, (“`self.name = ... “self.tableDefined = ”`), 2) extrae de la información del bus parámetros necesarios (`bits`, `twosC`) para instanciar el agente de cálculo que necesita la instancia de la clase `sensor` que se está iniciando.

El *agente* así “vinculado” al objeto de la clase `sensor` que se está inicializando queda, digamos, como “propiedad” del sensor (recuérdese que no estamos hablando de un -*objeto*- tipo de sensor, sino de un -*objeto*- sensor determinado del tipo `self.tipo` conectado al canal `self.channel` del ADCBus cuyo objeto se ha pasado como argumento para instanciar el sensor).

Volvamos a los conceptos una vez más (y, de nuevo, repitiéndolos):

1. Cada objeto de la clase `sensor` (ya se advirtió en su momento que el término no era el más apropiado; quizás “canalSensor” habría sido mejor) contiene -tiene que- contener- los valores adecuados para todos sus atributos, de forma que pueda instanciar el *agente de cálculo* que necesita y con los argumentos (parámetros) correctos. Así, como veremos un poco más adelante, la mera llamada al método

convert del agente de cálculo asociado (con el argumento de la lectura del ADC) nos devuelva la *lectura* del sensor.

2. El objeto de la clase *sensor* delegará en el objeto de la clase *agente de cálculo* el cálculo de la lectura. De ahí que los objetos de la clase *agente de cálculo* sean tan prolíficos -extensos- mientras que el *código-código* de la clase *sensor* es relativamente reducido (aunque denso).
3. La lógica de la librería exige que antes de instanciar los objetos de la clase *sensor* se haya hecho lo propio con, al menos, un *ADCBus* (lo que, si se analiza con detenimiento, es completamente lógico: el sensor no se conecta directamente al raspberry, sino a través de un bus -aunque sea lógico, como en el caso del *oneWire*-).
4. En el momento de la instancia el *cliente* pasa la información estrictamente necesaria del sensor (tipo, bus, canal, nombre, identificación; el nombre es opcional y la identificación solo es necesaria en el bus *oneWire*); el resto de la información necesaria para el cálculo de la lectura (que, se insiste, es casi lo único que hace la clase *sensor*) sale de 1) el *diccionario* de sensores (*sensor.standardSensorsData*) por medio del tipo (*self.tipo*) y 2) del objeto *ADCBus* que se pasa como argumento.

Veamos, para fijar ideas, un ejemplo de instancia de un sensor:

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

que instancia un objeto de la clase *sensor* al que llama "hw" (el objeto; el sensor se llama 'Hot water'). El sensor es del tipo "LM35" y está conectado al canal número "4" del bus *myBus*; no tiene identificación -el sensor-.

El método *__init__* que estamos analizando extrae del objeto de la clase *ADCBus* -*myBus*- el formato de los datos y la tensión de referencia y añade a estos los propios del sensor ("LM35"). Con todo ello instancia el agente -*self.sensorWizard* a continuación- que necesita para el cálculo (y del que, cada vez que se le pida calcular la lectura -*readout*-, echará mano).

```
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)

```

Este método es, en realidad, la continuación del `__init__`; se ha separado para dar mayor legibilidad a este, pero podría perfectamente haberse ubicado allí:

1. Indexa (localiza) en la tabla de *tensiones de referencia* del bus (ver 4.1.9, “Método `chooseVref` [tensión de referencia]. Método `setChannelGain`”) aquella inmediatamente superior a la que tiene como máxima el sensor, `self.VMax`. Obtiene tanto su valor -`self.vRef`- como su posición -`self.vRefPos`-.
2. Si el sensor tiene tabla asigna al atributo `self.table` el objeto de la clase `sensorTable` que la contiene.
3. Finalmente decide qué clase de la familia *agente de cálculo* le corresponde -al sensor- de acuerdo con sus características y las del bus al que está conectado. Lo instancia y asigna al atributo `self.agent`.

```

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

```

Finalmente, este método `-readout-` es la síntesis de todo lo anterior. Gracias a la complejidad introducida en la elaboración de las clases *agente de cálculo* (y en el método `__init__` de la clase `sensor`), ésta resuelve con elegancia el problema de calcular la lectura (que corresponde con el sensor del tipo *tal* conectado al ADC *cual*) con una sola instrucción.

5.4.- Clase sensor. Conclusiones.

- A diferencia de las clases de control de los buses, que encapsulan -implementan- el código necesario para su uso (y, por tanto, son instanciables -y perfectamente operativas- en sí mismas), la clase *sensor* -sus objetos- coordina el funcionamiento de aquellas y las de los agentes de cálculo. Esta clase no es, por tanto, útil -ni siquiera instanciable- sin la previa instanciación de alguno de los buses.
- El método `__init__` de la clase *sensor* (incluyendo en él lo contenido en el método `sensorWizard`) articula el funcionamiento conjunto del resto de clases de la librería. Es un buen ejemplo de como un método *constructor* puede contener mucha *miga* -lógica- que ahorre el código que con una programación *procedural* habría sido inevitable.
- Para la asignación del *agente de cálculo* necesario para un determinado sensor y un determinado bus se emplea un diccionario -`calculationAgentDict`- y un código -`sensorWizard`- que usa la secuencia “*if then else*” un par de veces. No es esta la parte de la librería que más se ajusta a los estándares de la OOP. La lógica contenida en `sensorWizard` no debería estar implementada así, sino como un método de una -otra- clase que contuviese, además, el diccionario `calculationAgentDict` como *atributo de clase*. Este método, por otra parte, se instanciaría una sola vez, ya que solamente implementa el algoritmo de asignación; aunque es lo ortodoxo, da lugar a un código poco legible que se ha procurado evitar.

6.- Clases para buses oneWire y arduino. Clase ADCBus.

En el punto 4 se ha justificado y diseñado la estructura y el código de la clase ADS1115. Al tratarse de un ADC con un manejo -la forma de programarlo, el *timing*, la forma de leer las conversiones- relativamente estándar, hemos decidido usar su *interface* como modelo para los otros dos buses que se estudian en este punto.

(Recuérdese que el *interface* es, básicamente, el conjunto de los métodos que la clase pone a disposición de sus *clientes* para el uso de sus objetos)

Tanto el bus *oneWire* como la forma de interactuar con el arduino -el bus *serial* más la propia programación de éste, del Promini- tienen un funcionamiento que se parece poco a la del ADS1115; no obstante, mantendremos, como queda dicho, el *interface* del ADS1115 como modelo para estas clases. Esto nos permitirá, como se explicará cuando toque (punto 8.2, “Uso con clases intermedias. Clase ‘ADCBus_manager’”), usar exactamente el mismo código para leer las lecturas de cualquiera de los buses mencionados.

6.1.- Clase *oneWire*.

El funcionamiento básico del bus *oneWire* está resumido en 2.3.1, “Bus *oneWire*” (es muy conveniente que relea este punto con detenimiento). A efectos de lo que sigue, recordar:

- El Raspberry sondea continuamente el bus para detectar cuántos dispositivos tiene conectados. El sistema operativo crea tantas carpetas como dispositivos encuentra en la carpeta del sistema *sys/bus/w1/devices*.
- Las carpetas correspondientes a termómetros tipo DS18B20 se nombran como 28-NNNNNNNNNNNN, dentro de cada una de estas carpetas hay un fichero de texto llamado *w1_slave* dentro del cual se deposita la lectura cada vez que -a través del sistema operativo- la aplicación lee el fichero -las aplicaciones escritas en Python leen los ficheros a través del sistema operativo-.
- A efectos del *interface* de la clase (de cómo nos vamos a entender con ella, con sus objetos), el bus *oneWire* siempre está “*ready*”: la lectura se hace en el mismo momento que se pide (*readConversion*).

De acuerdo con lo anterior, lo primero que tenemos que hacer es analizar las diferencias de funcionamiento entre ambos buses:

ADS1115	oneWire
Singleshot(): método para solicitar al ADC que comience una conversión.	El bus <i>oneWire</i> no tiene orden de disparo. El método correspondiente no hará nada (<i>pass</i> ; el método se hereda de la clase “ADCBus”)
Ready(): Comprobar si el ADC ha terminado la conversión.	El bus <i>oneWire</i> siempre está “ready”: la lectura se hace en el mismo momento que se pide (<i>readConversion</i>). El método en esta clase siempre devuelve <i>True</i> , de modo que el resto del código de cliente no se detenga.
readConversion(): Lee, y retorna, la última lectura del ADC.	Da la orden de lectura al canal que se haya seleccionado previamente (ver <i>setChannel</i> a continuación)
setChannel(channel): Programa el multiplexor (número de canal a convertir) del ADS1115.	Modifica los atributos <i>channel</i> y <i>currentChannelID</i> del objeto de modo que la próxima <i>readConversion</i> sea del canal correspondiente a estos atributos.
setChannelGain(sensor): Con la información -valor de los atributos- del objeto pasado como argumento (sensor: <i>channel</i> y <i>vRefPos</i>) programa el canal a convertir y la ganancia del ADS1115 (para el canal seleccionado).	El bus <i>oneWire</i> no tiene posibilidad de fijar ganancias diferentes para los sensores. Este método simplemente invoca <i>setChannel</i> con el número del sensor pasado como argumento.
__init__(): Inicia los valores de <i>address</i> y <i>ordinal</i> .	Inicia el valor de <i>ordinal</i> y del pin Vcc del bus en el conector GPIO del Raspberry. Sondea el bus para detectar los sensores conectados a él. Inicia los valores de <i>channel</i> y <i>currentChannelID</i>

Atributos de clase:

- | | |
|---|--|
| <ul style="list-style-type: none">• Formato de datos• Parámetros del ADS1115 | <ul style="list-style-type: none">• Formato datos.• Carpetas y ficheros sensores. |
|---|--|

Como ya se advirtió, puede que el empeño para que el interface de ambas clases - *ADS1115* y *oneWire* (y, después, *arduino*)- sea idéntico parezca gratuito, una *machada*. No lo es: se hace así para que el software de *cliente* sea lo más independiente posible del bus que quiere manejar. En el desarrollo del bus *ADS1115* se propuso una versión mínima de este (ver 4.1.6, “Código completo de la versión mínima de la clase *ADS1115*”), en ella se proponía una muestra de código de cliente:

```
myBus = ADS1115(0x48)

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

con la que es posible leer el primer canal del *ADS1115*. Pues bien, basta sustituir la instanciación del bus por:

```
myBus = oneWire(19)
```

y, *voilà*, tenemos la lectura del primer canal del bus *oneWire*: parece, pues, que es ventajoso conservar el *interface* (por cierto, si está usted viendo la temperatura ambiente multiplicada por mil, enhorabuena). Sobre este asunto del *interface* se vuelve varias veces en el blog; cada vez quedarán más claras las ventajas.

Finalmente: aparte de los métodos que forman el *interface* (la tabla anterior), la clase *oneWire* tiene alguna clase propia para los algoritmos propios (lectura del fichero *w1_slave*; elaboración de la lista de matrículas de los sensores). El código, completo en este caso, de la clase *oneWire* es:

```
class oneWire(ADCBus):
    tipo = 'oneWire'
    refVoltageList = [0, 9999]
    bits = 0
    twosC = False

    baseDirectory = "/sys/bus/w1/devices/"
    folderPrefix = '28-'
    sensorInfoFile = 'w1_slave'

    def __init__(self, pin, ordinal=0, name=None):
        self.ordinal = ordinal
        self.name = name
```

```

    self.initPin(pin)
    # Get sensors info.
    self.sensorsData = self.sensorsID()

    # 1st channel selected by default.
    self.setChannel(1)

def initPin(self, pin):
    """ Set passed pin as Vcc """
    GPIO.setmode(GPIO.BCM)
    GPIO.setup(pin, GPIO.OUT)
    GPIO.output(pin, True)
    time.sleep(1)

def sensorsID(self):
    """ Construct and return a list with the sensors info. """
    # Read base folder contents.
    folders = os.listdir(oneWire.baseDirectory)
    sensorsData = []
    for folder in folders:
        if not(oneWire.folderPrefix in folder):
            pass
        else:
            sensor = folder[len(oneWire.folderPrefix)-len(folder):]
            sensorsData.append(sensor)
    return sensorsData

def setChannel(self, channel):
    """ Sets the channel for next 'readConversion()' """
    self.channel = channel
    self.currentChannelID = self.sensorsData[channel-1]

def setChannelGain(self, sensor):
    """ Sets a channel """
    self.setChannel(sensor.channel)

def ready(self):
    """ Return true if new data available"""
    # oneWire ALWAYS has data available (see class comment)
    return True

def readConversion(self):
    """ Returns the conversion for current channel """
    sensorFile = os.path.join(oneWire.baseDirectory,
        (oneWire.folderPrefix+str(self.currentChannelID)))
    sensorFile = os.path.join(sensorFile, oneWire.sensorInfoFile)
    with open(sensorFile, "r") as f:
        data = f.readlines()
    return float(data[1].split("=")[1])

```

Clase *oneWire*. Análisis del código.

```
class oneWire(ADCBus):
```

- La clase hereda de la ADCBus, ver 6.4 “Clase ADCBus”.

```
    tipo = 'oneWire'  
    refVoltageList = [0, 9999]  
    bits = 0  
    twosC = False
```

- El bus *oneWire* no tiene tensiones de referencia, ni *bits* ni, por supuesto, la lectura está -ni deja de estar- en formato de complemento a dos.

```
baseDirectory = "/sys/bus/w1/devices/"  
folderPrefix = '28-'  
sensorInfoFile = 'w1_slave'
```

- Es la información para indicar dónde están las carpetas de los sensores, qué prefijos están previstos y en qué fichero está la información de la medida del sensor (Ver comentarios al comienzo de este punto).

```
def __init__(self, pin, ordinal=0, name=None):  
    self.ordinal = ordinal  
    self.name = name  
    self.initPin(pin)  
    # Get sensors info.  
    self.sensorsData = self.sensorsID()  
  
    # 1st channel selected by default.  
    self.setChannel(1)
```

- Por razones puramente prácticas (es necesario disponer de 3.3 V en algún sitio y el conector del raspberry es francamente *tacaño* con esta posibilidad: sólo lo entrega en un pin que queda, vaya *por dios*, debajo de los conectores de casi todas las pantallas LCD que se emplean en los kits de medida de temperatura comerciales - figura 9-. Ver también “Anexo II.3.2.- Bus *oneWire*”) se deja a elección del *cliente* de la librería la activación de un pin genérico para suministrar 3.3 V: *pin*, ver a continuación *initPin*.
- La lista *self.sensorsData* pasa a contener las matrículas de los sensores que están conectados al bus. Ver *self.sensorsID* a continuación.
- Se inicia el objeto con el sensor “1” activado -seleccionado-.

```

def initPin(self, pin):
    """ Set passed pin as Vcc """
    GPIO.setmode(GPIO.BCM)
    GPIO.setup(pin, GPIO.OUT)
    GPIO.output(pin, True)
    time.sleep(14)

```

- Ver comentario en `__init__`.
- Se dejan 14 segundos `-time.sleep(14)-` para tener la completa seguridad de que el bus se activa correctamente.

```

def sensorsID(self):
    """ Construct and return a list with the sensors info. """
    # Read base folder contents.
    folders = os.listdir(oneWire.baseDirectory)
    sensorsData = []
    for folder in folders:
        if not(oneWire.folderPrefix in folder):
            pass
        else:
            sensor = folder[len(oneWire.folderPrefix)-len(folder):]
            sensorsData.append(sensor)
    return sensorsData

```

- A partir de las carpetas en `baseDirectory` que comiencen por `folderPrefix` se crea la lista `sensorsData` con -exclusivamente- las matrículas de los sensores.

```

def getIDs(self):
    """ Return a list with the sensors ID's """
    return self.sensorsData

```

- Retornar una lista con las matriculas de los sensores.

```

def setChannel(self, channel):
    """ Sets the channel for next 'readConversion()' """
    self.channel = channel
    self.currentChannelID = self.sensorsData[channel-1]

```

- A petición del `cliente` se activa (selecciona) un canal. No hay tal `canal`, en realidad: el bus `oneWire` se organiza con las matrículas de los sensores que tiene conectados, de forma que a qué canal corresponde qué sensor es, casi, completamente arbitrario. Puede usted arreglarse como la mayoría, calentar los

sensores -un poco- con un mechero para ver qué número hace cada uno, o puede conectarlos de uno en uno y, mediante el método *getIDs* (o consultando las carpetas) ir apuntando las matrículas (y luego marcándolos con un rotulador permanente ...).

- Recuerde (“channel – 1”) que las listas se numeran empezando por el elemento “0”

```
def setChannelGain(self, sensor):  
    """ Sets a channel """  
    self.setChannel(sensor.channel)
```

- Este método se conserva solamente a efectos de compatibilidad de interface. Invoca *setChannel*.

```
def ready(self):  
    """ Return true if new data available """  
    # oneWire ALWAYS has data available (see class comment)  
    return True
```

- Como ya se ha explicado profusamente, el bus oneWire está siempre “ready”. En realidad este bus ni está *ready* ni deja de estarlo; lo que ocurre es que, de nuevo por compatibilidad con el interface del bus ADS1115 (y con el de la mayoría de ADC, que funcionan parecido), al *cliente* hay que engañarle para que, informado de que el bus sí está *ready*, pida la conversión. Si se quiere mantener el *interface* - cosa que se aconseja- no hay otra forma de hacerlo. (Es, por otra parte, perfectamente factible usar la clase sin necesidad de usar el método *ready*).

```
def readConversion(self):  
    """ Returns the conversion for current channel """  
    sensorFile = os.path.join(oneWire.baseDirectory,  
        (oneWire.folderPrefix+str(self.currentChannelID)))  
    sensorFile = os.path.join(sensorFile, oneWire.sensorInfoFile)  
    with open(sensorFile, "r") as f:  
        data = f.readlines()  
    return float(data[1].split("=")[1])
```

- Este método retorna la lectura correspondiente al sensor que esté seleccionado - *self.channel*- . Ver comentarios en el método *setChannel*.



El código anterior es perfectamente correcto y operativo. Hay, no obstante, que activar ciertos módulos del sistema para que funcione; en concreto:

```
import os                      # Utilidades del sistema operativo
import time                     # Módulo para medida de tiempo
import Rpi.GPIO as GPIO         # Módulo para manejo de los pines del Raspberry
```

escriba este código antes del de la clase (está incorporado en los ejemplos del repositorio).



(a) Escriba un código de cliente que, usando una instancia de la clase `oneWire`, muestre las lecturas de todos los sensores conectados al bus. No haga uso de ningún método que no sea estrictamente necesario (`ready`, por ejemplo) y déjese, en esta ocasión, de *agentes de cálculo* y de la clase `sensor`.

Pista: Use el método `sensorsData.index("identificación del sensor")` -recuerde que las listas son objetos de la clase `list`; `index` es un método de la clase- para saber qué posición ocupa en la lista.

Este código -el que usted escriba para resolver este ejercicio) no será válido nada más que para el bus `oneWire`.



(b) Compruebe que, en efecto, el siguiente código escrito para el `ADS1115`:

```
myBus = ADS1115(0x48)

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

funciona con el bus `oneWire` simplemente cambiando la instrucción de instanciación del bus (Tiene, además y de momento, que añadir al código de la clase el método `singleShot` -con la instrucción `pass`-; más adelante, 6.3, “Clase `ADCBus`” se explica por qué). Recuerde que la instancia de `oneWire` tiene un argumento: el número de pin por el que se alimentará el bus; según como tenga conectado el bus, este valor puede ser arbitrario (o no; ver Anexo II, “3.2 Bus `oneWire`”).



(c) Escriba un método (`readChannelByID?`) que lea el sensor en base a su identificación (código hexadecimal de 12 caracteres). Pista: aunque no sirve directamente para este propósito, la lista `sensorsData` puede ser utilizada. Use el método `sensorsData.index("identificación del sensor")` -recuerde que las listas son objetos de la clase *list*; `index` es un método de la clase- para saber qué posición ocupa en la tabla.



(d) Utilizando un código similar al del ejercicio del punto 4.1.11, compruebe la velocidad de conversión del bus oneWire. Hágalo con un lazo pero ni se le ocurra hacerlo más de 10 veces -el lazo-. El bus oneWire, como ya se ha advertido en varias ocasiones, es muy, pero que muy lento.

En el punto 8, “Uso de la librería” encontrará varios ejemplos más de uso de la clase `oneWire`.

6.2.- Clase arduino.

Veamos cuales son las especificaciones de la placa en la que está montada el arduino Promini; qué instrucciones y en qué orden hay que darle para que convierta y como entrega los datos: Esta es la especificación que da el fabricante:

1. Para asegurarse del correcto arranque del arduino Promini hay que darle un *pulso de reset*. El pin del conector GPIO del Raspberry al que está conectado el *reset* del Promini es el 22; este se tiene que poner a “0” (LoW) durante un segundo. Una vez enviado este *pulso* hay que esperar 4 segundos para empezar a pedir conversiones.
2. El arduino Promini se comunica por su interface serial; este está conectado a los pines correspondientes del conector GPIO del Raspberry: 14 para el Tx -*transmit*- del Raspberry y 15 para el Rx -*receive*- . El protocolo es: 57600 *baudios*, tamaño *byte* 8, sin paridad, 1 *bit de stop*.
3. Una vez reseteado, el arduino Promini se pone a la espera de recibir 16 bloques de 16 *bytes* cuyo contenido es arbitrario. Una vez recibidos, el arduino Promini envía -devuelve- a su vez otros 16 bloques de 16 *bytes*; finalizado este envío queda a la espera de la orden de conversión. Este intercambio de información sirve para confirmar la correcta conexión de ambos dispositivos.
4. La orden de conversión es el envío de un *byte* de contenido arbitrario (ver nota a continuación) desde el Raspberry al arduino Promini. Sin solución de continuidad -más allá de los aproximadamente 30 us que tarda en convertir- el arduino Promini

contesta con un bloque de 32 *bytes* en el que va la información de la conversión de los ocho canales A/D. Una vez enviada la información el arduino Promini queda a la espera de la siguiente orden de conversión. No es necesaria ninguna precaución especial para desconectarlo (apagarlo).

5. Las conversiones A/D vienen en los 16 primeros *bytes* del paquete de 32 que envía el arduino Promini, es decir, el contenido es:
“1122334455667788XXXXXXXXXXXXXX”, la información de cada canal viene con el *byte* más significativo delante.

 **byte de disparo:** el *byte* de disparo de la conversión tiene, además, otra finalidad de la que no hacemos uso en esta librería: el software en el arduino Promini utiliza la información contenida en el *byte* para controlar 8 de sus salidas digitales. El *data logger* que monta la placa puede controlar umbrales de alarma de los parámetros que mide y, eventualmente, controlar las salidas para señalización (luminosa, acústica, ...)

De acuerdo con lo anterior, procedemos a analizar las diferencias entre la clase ADS1115 y la arduino:

ADS1115	arduino
Singleshot(): método para solicitar al ADC que comience una conversión.	Envío de un <i>byte</i> arbitrario al arduino Promini. Se produce, siempre, la conversión de los 8 canales.
Ready(): Comprobar si el ADC ha terminado la conversión.	Se emplea para esta función la comprobación de si hay 32 <i>bytes</i> de información en el <i>buffer</i> de entrada del canal de comunicación serie del Raspberry.
readConversion(): Lee, y retorna, la última lectura del ADC.	El arduino Promini envía siempre la conversión de los 8 canales A/D. Para que este método sea compatible con el del ADS1115, retornará la conversión del canal que esté seleccionado en el objeto (ver <i>setChannel</i> a continuación)
setChannel(channel): Programa el multiplexor (número de canal a convertir) del ADS1115.	Modifica el atributo <i>self.channel</i> del objeto de modo que la próxima <i>readConversion</i> devuelva la conversión del canal correspondiente a este atributo.
setChannelGain(sensor): Con la información -valor de los atributos- del objeto pasado como argumento (sensor: <i>channel</i> y <i>vRefPos</i>) programa el canal a convertir y la ganancia del ADS1115 (para el canal seleccionado).	El arduino Promini no tiene posibilidad de fijar ganancias diferentes para los sensores. Este método simplemente invoca <i>setChannel</i> con el número del sensor pasado como argumento.
__init__(): Inicia los valores de "address" y <i>ordinal</i> .	Inicia el valor de <i>ordinal</i> . Resetea al arduino Promini enviando un pulso de <i>reset</i> e instancia la clase "serial" para disponer de la comunicación serie. Comprueba la correcta comunicación (ver punto 3 más arriba).

Atributos de clase:

- Formato de datos
- Parámetros del ADS1115
- Formato datos.
- Formato comunicación serie.

La clase arduino también tiene métodos propios: *reset* del arduino Promini, conversión del *buffer* a valores binarios, ... A continuación puede encontrar el código completo de la clase.

```
class arduino(ADCBus):
    """ Class to implement an arduino Promini ADC converter """

    tipo = 'ProMini'
    bits = 10
    twosC = False

    refVoltageList = [0, 5.1]

    def __init__(self, ordinal):

        self.ordinal = ordinal

        self.channel = 1           # Default channel

        self.reset()
        self.initCommunication()

    def reset(self):
        """ Reset arduino """
        # Configure pin
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(22, GPIO.OUT)

        GPIO.output(22, False)      # Send a "0"
        time.sleep(1)               # Hold it for a second
        GPIO.output(22, True)       # Send a "1"
        GPIO.cleanup()              # Close module
        time.sleep(4)

    def initCommunication(self):
        """ Start an analog to digital conversion """
        dummy = "0123456789012345"
        for i in range(16):
            self.enviar(dummy)

            for i in range(1,17):
```

```

        while not(RS232.inWaiting())>15):
            pass

    def enviar(self, texto):
        RS232.write(texto)

    def readBuffer(self):
        """ Get arduino buffer -32 bytes- information """
        return RS232.read(32)

    def setChannel(self, channel):
        """ Sets the channel for next 'readConversion()' """
        self.channel = channel.channel

    def setChannelGain(self, sensor):
        """ Sets the current channel # """
        self.setChannel(sensor.channel)

    def singleShot(self):
        """ Start an analog to digital conversion """
        self.enviar('0')

    def ready(self):
        """ Return true if data has arrived from arduino"""
        hay = RS232.inWaiting()
        if hay > 31:
            return True
        else:
            return False

    def readConversion(self):
        """ Returns the conversion for current channel """
        cadena = self.readBuffer()
        conversion = ord(cadena[2*(self.channel-1)])\
            +256*ord(cadena[2*(self.channel-1)+1])
        return conversion

```

6.3.- Clase ADCBus.

La clase ADCBus es una clase *ancestro* (y, casi, una clase “abstracta”).

Las clases abstractas solamente contienen métodos vacíos (no es este caso). Sus funciones son múltiples, algunas muy sutiles y su explicación queda claramente fuera del alcance de este blog. Una de ellas es definir métodos que alguna de las clases herederas pueden no usar pero que, debido a la arquitectura de la aplicación, pueden ser invocadas -aunque no hagan nada-. Por lo general los métodos que se incluyen solamente para definir el *interface* -de las clases herederas- en la clase ancestro suelen no tener ningún código (*pass*, en el caso de Python). A veces alguna de las clases herederas necesitan de alguno de estos métodos justamente eso, que *pasen*, en cuyo caso se puede no definir el

método en la clase heredera ya que sirve el del ancestro (es el caso del método *singleShot* para la clase *oneWire*: vale con *pass*).

En nuestro caso hay, además, un método común a todas las clases *ADCBus* (*chooseVref*) que sirve para encontrar la tensión de referencia empleada para cada sensor (y que, por cierto, tiene una implementación completamente *fake* -la lista de tensiones- en los buses *oneWire* y *arduino*).

El código de *ADCBus* es:

```
class ADCBus():
    """ Base class for ADC-bus devices """

    instancedBuses = []

    def chooseVref(self, sensorMax):
        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)

    def setChannel(self, channel):
        pass

    def setChannelGain(self, sensor):
        pass

    def singleShot(self):
        pass

    def ready(self):
        pass

    def readConversion(self):
        pass
```

Clase *ADCBus*. Análisis del código.

```
class ADCBus():
    """ Base class for ADC-bus devices """

    instancedBuses = []

    • La clase no hereda de ninguna
```

- *instancedBuses* es una lista que se puede usar, opcionalmente, para alojar los buses instanciados (el cliente puede crear su propia lista en su código que serviría exactamente igual).

```
def chooseVref(self, sensorMax):
    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)
```

- Este método busca en la lista de tensiones de referencia del bus la inmediatamente superior a la pasada. Ésta -la pasada como argumento- es la máxima esperable del sensor (ver 5.2, “Clase sensor. Preliminar”).

```
def setChannel(self, channel):
    pass

def setChannelGain(self, sensor):
    pass

def singleShot(self):
    pass

def ready(self):
    pass

def readConversion(self):
    pass
```

- Clases que forman el *interface* de las clases ADCBus. Excepto el método *singleShot* en la clase oneWire todos los métodos son sustituidos en cada una de las clases herederas por sus propios métodos.

6.4.- Clases para buses. Diagrama. Conclusión.

El siguiente diagrama muestra la jerarquía y diferencias entre todas las clases ADCBus:

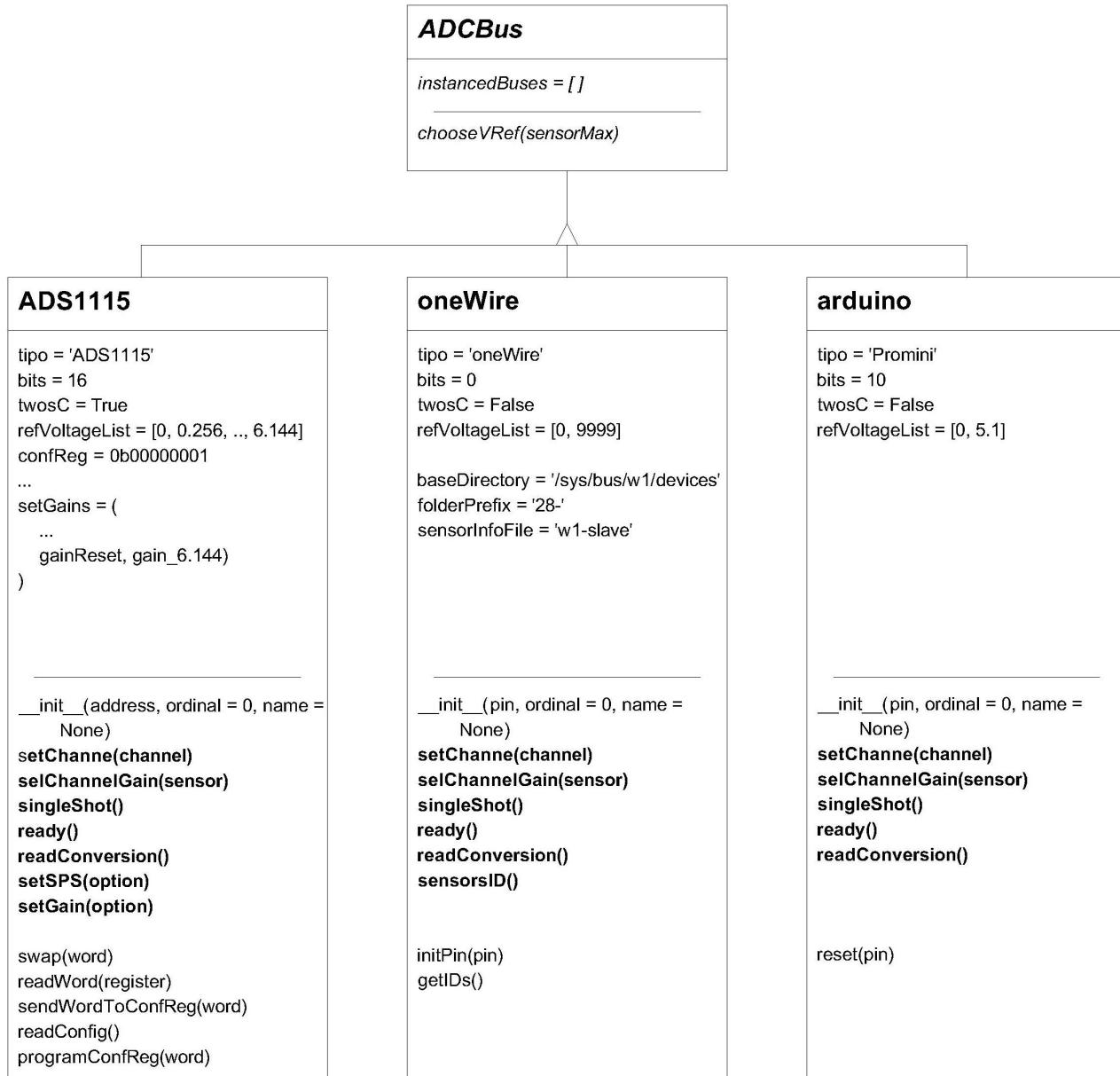


Figura 26. Diagrama de las clases ADCBus

(Las clases en negrita son las únicas que se deberían poder invocar desde el código del cliente. Las comunes a todas las clases son el *interface* común).

- El esfuerzo para que las clases tengan el mismo ***interface*** se paga con creces al ofrecer todas las clases un comportamiento idéntico. El código de *cliente* es -más allá de la instancia del bus que, naturalmente, se tiene que hacer con el nombre de la clase correspondiente- independiente del bus del que se trate. Esto

permite que el código desarrollado para un bus determinado sea válido para otro; también es posible el uso homogéneo y conjunto de las clases de la librería a partir solamente de los sensores instanciados (previsto que se inicien los buses con anticipación; ver 8.1.2, “Uso con *lista de sensores*”). Por tratarse del más estándar de los tres estudiados se ha tomado como modelo el *-interface-* del ADS1115.

- La inclusión de un nuevo tipo de ADC (hay varios muy populares: la familia *MCP3X08* del fabricante *microChip*, por ejemplo) exige la elaboración de una clase para el nuevo ADC. Si se sigue el estándar para el *interface*, el software de *cliente* ya desarrollado para los existentes (ver 8, “Uso de la librería”) valdrá casi con toda seguridad para el nuevo.
- A pesar de la disparidad entre ellos -entre el código de los métodos que las implementan-, se ha hecho a las tres clases de los tres buses estudiados herederas de una clase común *ADCBus* de forma que comparten un método *-chooseVref-* y la definición del *interface*. En todo caso, la complejidad de esta familia de clases es menor que la de los *agentes de cálculo*: no hay herencia múltiple.

7.- Tablas de sensores (*sensorTable*).

Una *tabla* no es otra cosa que una “lista de listas” (de pares de *coordenadas*), pero la gestión de éstas tiene la suficiente complejidad como para tratarla -la gestión- como objeto de su propia clase. Aunque esta descripción -la de la clase *sensorTable*- viene después de la de las clases de los *buses*, los *agentes de cálculo* y de la clase *sensor*, no es, ni mucho menos, complicada; simplemente no tocaba hasta aquí.

Puede, si lo desea, tomarse el análisis y la elaboración de esta clase como un repaso de lo anterior. Si ya se ha metido usted por el cuerpo todo lo que precede con el debido aprovechamiento, lo de las *tablas* va a ser, creáselo, un *paseo militar*.

Especificación.

Veamos cuál es el problema. Se trata de elaborar una clase que resuelva lo siguiente:

1. Un contenido numérico -*salida*- debe ser calculado a partir de otro número - *entrada*- por medio de una tabla de consulta -*lookup table*-; la tabla consiste en una colección de al menos tres pares de puntos en la que no habrá dos *abcisas* -el primer número de cada par- repetidas. De acuerdo con lo habitual llamaremos al primer número de un punto *abcisa* y al segundo *ordenada*; ambas son las *coordenadas* del punto.
2. Las tablas estarán contenidas inicialmente en ficheros de texto *llano* -ASCII-. El formato interno del fichero será el más simple posible: un punto de la tabla por línea con las dos *coordenadas* separadas por una coma. Se usará el punto decimal como separador de decimales (formato anglosajón). No habrá encabezamiento (la primera línea contendrá el primer punto, la segunda el segundo, etcétera). En la instanciación de la clase *sensorTables* se leerá la tabla del fichero y se convertirá a un formato adecuado para su consulta con Python (*lista de listas*. Éstas de dos números reales -las *coordenadas* de cada punto-).
3. La clase -sus objetos- comprobarán la *consistencia* de los datos a partir de los que se genera la tabla. El contenido del fichero se comprobará a medida que se lee: cada línea tiene que contener dos números reales separados por una coma. Además, la tabla no tendrá dos puntos con la misma *abcisa* y deberá contener al menos tres puntos. Estas dos últimas comprobaciones generarán, si el resultado es incorrecto, un error del tipo -clase- “*EIDEErrror*”.
4. Se utilizará un método de interpolación lineal (ver figura 27 a continuación). En esencia el método consiste en lo siguiente (no se interprete *alegremente* que el *método de interpolación* lineal se convertirá en un *método* de una clase: igual sí o igual no; se verá):
 1. Si el punto está dentro de la tabla (su *abcisa* es igual o mayor que la del primer punto e igual o menor que la del último), se calculará la *salida* como la *ordenada* del punto sobre la línea que une los puntos cuyas *abcisas* son la

inmediatamente anterior y posterior a la -*abcisa*- pasada. Un pequeño trabajuelo que se entiende sin dificultad echando un vistazo a la figura 27.

- Si el punto está fuera de la tabla se calculará de la misma forma con la línea que une los dos primeros(últimos) puntos de la tabla.

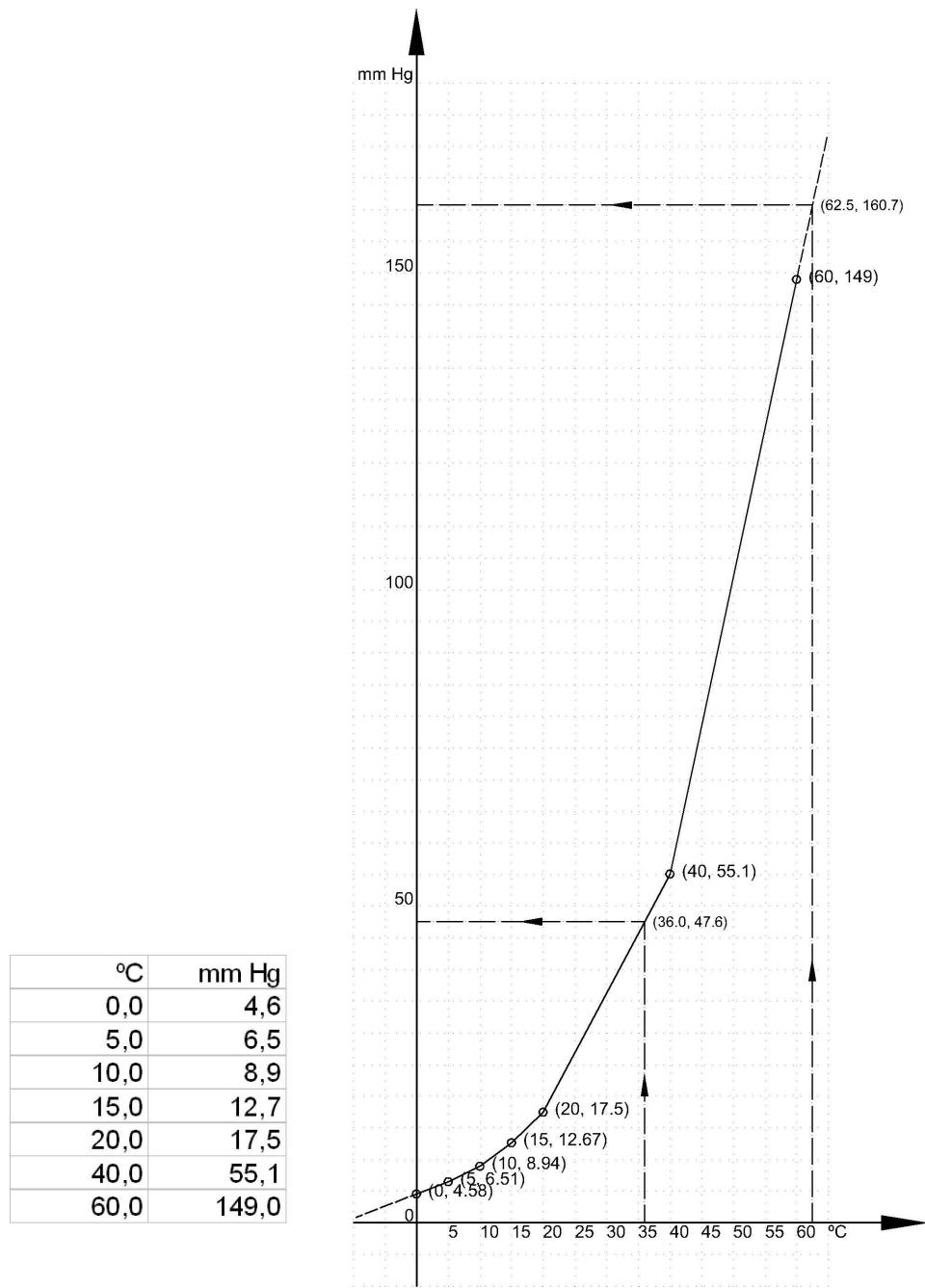


Figura 27. Ejemplo de interpolación lineal. *Presión de vapor* del agua a partir de su temperatura.

Análisis.

Es este, el que tenemos entre manos, un proyecto *modesto*. Si no fuese así, al llegar a este punto habría que hacer un montón de cábalas: aparte de la tabla propiamente dicha - que sí se merece una clase por sí sola: hay mucho que hacer con las tablas, como se verá a continuación. En ello estamos- quizá habría que plantearse las siguientes clases: *abcisa*, *ordenada*, *punto*, *línea*, *tabla*, *fichero*. Vayamos por partes:

Clases “de fábrica”.

En Python todo lo que se usa son objetos -instancias- de alguna clase (hasta las propias clases son objetos, hay que *jorobarse*).

No se despeleje la mente si es la primera vez que se lo plantea -lo de que todo son objetos- y tiene ahora mismo una emergencia neuronal. Véalo así: las *coordenadas* son números reales -*float*- y, como los *float* son, a lo que se ve, objetos de una clase -¿*float*?: sí- no tenemos que crear dicha clase. Los objetos de la clase *float* ya vienen *con todo puesto*: un ejemplo, saben sumarse con otro float: “*float.add*” (y si no se lo cree, arranque el IDLE, teclee “*a = 3.14*”, “*type(a)*”, “*dir(a)*” y “*a.__add__(2.2)*” y verá lo que es bueno. Por cierto, esto -lo la clase *float*- es cierto desde el primer renglón del blog: no se lo hemos contado hasta aquí para no liar demasiado las cosas).

En definitiva, que no hay que crear una clase “*numeroReal*”; viene *de fábrica*. Por tanto, buena noticia, nos olvidamos de “*abcisa*” y “*ordenada*” como objetos de una nueva clase: son números reales -*float*-. Nos ahorraremos, puesto que no tienen más complejidad que esa, la de ser números reales, una nueva clase.

Por otra parte Python tiene, afortunadamente, una clase *file* -fichero- que nos libera de la gestión del fichero de texto en el que están los valores de la tabla. Otra buena noticia.

Harina de otro costal son las eventuales clases *punto* y *línea*: la finalidad de la clase *sensorTables* es el cálculo del valor correspondiente a una determinada *abcisa* empleando una tabla de valores discretos -puntos-, y la especificación que nos hemos autoimpuesto es resolverlo por interpolación lineal (figura 27).

Por primera vez en el desarrollo de la librería, excepción hecha de los *agentes de cálculo*, estamos tratando con asuntos que no tienen que ver ni con sensores ni con buses ADC: los *puntos* y *las líneas* son geometría -*euclíadiana*, por cierto- (lo mismo pasa con los ficheros -que tampoco son sensores ni buses-, pero eso ya lo hemos resuelto). Y el problema de la interpolación no puede ser más geométrico, al menos conceptualmente.

Un proyecto de más *fuste* contemplaría, bien el uso de una librería existente (*sympy*, por ejemplo), bien la creación de una propia para las cuestiones geométricas. La librería *EIDEGraphics*, por ejemplo, también usa conceptos geométricos parecidos para algunas cosas (dibujar la aguja de un display, por ejemplo), y, quizás, convendría pensar en centralizar en una pequeña -o no tan pequeña- librería auxiliar estas cuestiones.

¡Cuán largo me lo fiáis, amigo Sancho!; y también valdría, probablemente, lo de que “lo mejor es enemigo de lo bueno”. Efectivamente podríamos embarcarnos en la elaboración de una librería que, entre otras cosas, haga lo que el método *linearInterpolate* (ver a

continuación) por medio de un objeto de una clase *line* -línea- que, a su vez, instancie dos objetos de la clase *punto*, etcétera (mira, lo propondremos como ejercicio al final de este punto. Ale). Lo que ocurre es que nos saldríamos demasiado de la línea del proyecto.

Aunque el método *linearInterpolate* es el que tiene más chicha (el más épico, vamos) de los de la clase *sensorTables*, lo cierto es que es poca cosa: siete líneas de código nada más. Lo dejaremos de momento así, como un método más de *sensorTables*: se insiste en que no es lo más ortodoxo desde el punto de vista conceptual (OOP), pero es la primera licencia que nos tomamos (y, dicho sea de paso, tampoco es para tanto).

Veamos, pues, cuál es el código de la clase *sensorTables*.

```
class sensorTable():
    """ Class for sensor tables """
    head = os.getcwd()
    tables = os.path.join(head, 'SENSOR_TABLES')

    def __init__(self, archivo):
        self.name = archivo.split(".")[0]
        self.table = []
        self.archivoPath = os.path.join(tables, self.name + '.txt')
        self.read(self.archivoPath)

        self.mapLen = len(self.table)
        self.mapMinimum = self.table[0][0]
        self.mapMaximum = self.table[len(self.table) - 1][0]
        self.sort()
        self.verify()

        self.position = -1

    def read(self, file):
        fichero = open(self.archivoPath, "r+")
        for linea in fichero:
            pointsList = (float(linea.split(",")[0]),
                          float(linea.split(",")[1]))
            self.table.append(pointsList)
        fichero.close()

    def sort(self):
        self.table.sort()

    def verify(self):
        if self.mapLen < 4:
            raise EIDEError("", "Sensor table error: less than three points")
        for counter, i in enumerate(self.table):
            if counter == self.mapLen - 1:
                # Table top reached.
                return
            print (i[0], self.table[counter+1][0])
            if i[0] == self.table[counter+1][0]:
```

```

        raise EIDEEError("", "Sensor table error: double abcissa")

def lookup(self, abcissa):
    """ Return ordinate for abcissa """
    return self.linearInterpolate(self.abcissaPoints(
        self.pointer(abcissa)), abcissa)

def pointer(self, abcissa):
    """ Return position of equal or first smaller number """
    if abcissa < self.table[0][0]:
        # abcissa 'below' table.
        return 0
    for contador,i in enumerate(self.table):
        if i[0] > abcissa:
            return contador - 1
    return contador

def abcissaPoints(self, pointer):
    """ Return a list holding interpolation points """
    if pointer >= (self.mapLen - 1):
        # Point 'above' table.
        return (self.table[self.mapLen - 2],
                self.table[self.mapLen - 1])
    return (self.table[pointer],
            self.table[pointer + 1])

def linearInterpolate(self, points, abcissa):
    """ Return a list holding interpolation points """
    x1 = points[0][0]
    y1 = points[0][1]
    x2 = points[1][0]
    y2 = points[1][1]
    m = (y2-y1)/(x2-x1)
    b = y2 - (x2/(x2-x1))*(y2-y1)
    return (m * abcissa + b)

```

Clase sensorTable. Análisis del código.

```

class sensorTable():
    """ Class for sensor tables """
    head = os.getcwd()
    tables = os.path.join(head, 'SENSOR_TABLES')

```

- El código debajo de una definición de clase se ejecuta una sola vez (cuando Python ve la clase). En este caso sirve para definir la carpeta en la que están los ficheros de texto que contienen las tablas (“..../SENSOR_TABLES”). Ver Anexo I “Descarga y configuración”.)

 Para que la clase sensorTable pueda ser instanciada el código tiene que residir en la carpeta ‘EIDEAnalog’ dentro de la cual tiene que estar también la carpeta ‘SENSOR_TABLES’ que contiene los ficheros con las tablas. Si se ha descargado usted la librería desde <https://github.com/Clave-EIDEAnalog/EIDEAnalog> (ver Anexo I “Descarga y configuración”), debería tener las carpetas indicadas dispuestas de la forma indicada.

```
def __init__(self, archivo):  
  
    self.name = archivo.split(".")[0]  
    self.name = self.name + '.txt'  
    self.archivoPath = os.path.join(sensorTable.tables, self.name)  
    self.table = []  
    self.read(self.archivoPath)  
  
    self.mapLen = len(self.table)  
    self.mapMinimum = self.table[0][0]  
    self.mapMaximum = self.table[len(self.table) - 1][0]  
    self.sort()  
    self.verify()
```

- El nombre del fichero se separa en tantas partes como puntos decimales contenga (menos uno). A la primera de estas partes se le añade el sufijo “.txt” para conformar el nombre definitivo del fichero (así, si se pasa ‘DATOS’, ‘DATOS.txt’ o ‘DATOS.MAS.DATOS’ como nombre del fichero, el nombre efectivo será, en todos los casos, ‘DATOS.txt’).
- *self.table* es la lista que contendrá las coordenadas.
- Lectura del fichero.
- Se almacenan en atributos algunos datos de la tabla (*self.mapLen*, *self.mapMinimum*, *self.mapMaximum*)
- Se clasifica y comprueba la tabla.

```
def read(self, file):  
    fichero = open(self.archivoPath, "r+")  
    for linea in fichero:  
        pointsList = (float(linea.split(",")[0]),  
                      float(linea.split(",")[1]))  
        self.table.append(pointsList)  
    fichero.close()
```

- Lectura del fichero y creación de la tabla primaria (lista de listas de pares de coordenadas). Al utilizar la conversión a número real -float- se realiza indirectamente una verificación del contenido del fichero (se produce un error si cada línea del fichero no contiene dos números reales separados por una coma).

```
def sort(self):
    """ Sort the table by abscissas """
    self.table.sort()
```

- Clasificar la tabla por abcisas.

```
def verify(self):
    if self.mapLen < 3:
        raise EIDEEError("", "Sensor table error: less than three points")
    for counter, i in enumerate(self.table):
        if counter == self.mapLen - 1:
            # Table top reached.
            return
        if i[0] == self.table[counter+1][0]:
            raise EIDEEError("", "Sensor table error: double abscissa")
```

- Verificar la consistencia de los datos: Aquí hacemos nuestros primeros *pinitos* con la detección y tratamiento de errores (aunque limitado: sólo detección, de momento). Se trata de verificar que la tabla tiene al menos tres puntos y ninguna abcisa repetida; las instrucciones pueden sonar raras, pero son, en este caso, bastante intuitivas. Python genera sus propios errores cuando encuentra una condición de error al ejecutar el código (no es el caso que estamos analizando; eso un poco más abajo en este mismo párrafo): suelen ser errores de codificación -por ejemplo, usar un literal en lugar de un número (o viceversa)- o dinámicos, un fichero o carpeta que no existe; aparecen en el IDLE con un texto bastante oscuro (oscuro por alarmante y poco comprensible; el texto suele ser en rojo). Si el error no es detectable por Python, en este caso por que lo que se comprueba es si la tabla es demasiado pequeña o tiene abcisas repetidas, es el propio código el que tiene que verificarlo y generar -raise- el error (que aparecerá como si fuese otro error más: lo del IDLE). Como todo en Python, el error propiamente dicho es también un objeto de una clase, *EIDEEError* en nuestro caso. Esta clase se debería haber definido en algún otro sitio como heredera de la clase genérica *Exception*.

```
def lookup(self, abscissa):
    """ Return ordinate for abscissa """
    return self.linearInterpolate(self.abscissaPoints(
        self.pointer(abscissa)), abscissa)
```

- Método de cálculo de la *salida* a partir de la *entrada -abcissa-*. Hace uso de los métodos *pointer* y *abcissaPoints* para localizar los puntos en la tabla; después se los pasa al método de interpolación *linearInterpolate*.

```
def pointer(self, abcissa):
    """ Return position of equal or first smaller number """
    if abcissa < self.table[0][0]:
        # abcissa 'below' table.
        return 0
    for contador,i in enumerate(self.table):
        if i[0] > abcissa:
            return contador - 1
    return contador
```

- Localiza el punto en la tabla. Retorna la posición del primer punto de la línea (que se usará para la interpolación).

```
def abcissaPoints(self, pointer):
    """ Return a list holding interpolation points """
    if pointer >= (self.mapLen - 1):
        # Point 'above' table.
        return (self.table[self.mapLen - 2],
                self.table[self.mapLen - 1])
    return (self.table[pointer],
            self.table[pointer + 1])
```

- Devuelve los puntos correspondientes a *pointer* (que determinan la línea que se usará para la interpolación).

```
def linearInterpolate(self, points, abcissa):
    """ Return a list holding interpolation points """
    x1 = points[0][0]
    y1 = points[0][1]
    x2 = points[1][0]
    y2 = points[1][1]
    m = (y2-y1)/(x2-x1)
    b = y2 - (x2/(x2-x1))*(y2-y1)
    return (m * abcissa + b)
```

- Cálculo de los parámetros -m, b- de la recta para interpolar y de la ordenada correspondiente a la abcisa pasada.



(a) Instancie la clase `sensorTables` con el fichero 'NTC_ADS1115.txt' como argumento. Calcule por medio de la instancia creada la temperatura correspondiente a las lecturas 1.0, 2.0 y 2.5 Voltios.



(b) Compruebe, con la ayuda de la tabla 'VAPOUR_PRESSURE', los valores de la presión de vapor del agua para 36,0 °C y 62,5 °C (Figura 27).



(c) Cree un fichero de tabla ('FAULTY_TABLE') con sólo dos puntos. Verifique el error al intentar instanciarlo. Compruebe también que la instancia da error si el fichero tiene una abcisa duplicada y, también, si en lugar de valores -números- contiene textos. Estudie la diferencia entre los mensajes de error (los dos primeros deberían estar tratados por un gestor de errores propio -*EIDEError*- que, de momento, no existe. El último es un error *nativo* de Python).



Para que la clase `sensorTable` pueda ser instanciada tal y como se indica en los ejercicios anteriores, el código tiene que residir en la carpeta 'EIDEAnalog' dentro de la cual tiene que estar también la carpeta 'SENSOR_TABLES' que contiene los ficheros con las tablas. Si se ha descargado usted la librería desde <https://github.com/Clave-EIDEAnalog/EIDEAnalog> (ver ver Anexo I "Descarga y configuración"), debería tener las carpetas indicadas dispuestas de la forma indicada.

7.1.- Tablas de sensores. Conclusiones.

- Aunque la tabla que contiene los puntos para el cálculo por interpolación no sea más que una lista de Python, conviene que esta lista sea un atributo más de los objetos de una clase `sensorTable`. Esto permite encapsular los datos -la tabla- junto con los métodos para manejarla.
- De los métodos de la clase hay uno, `linearInterpolate`, que tiene un contenido -algoritmo- impropio del manejo de la tabla: quizás debería formar parte de una clase diferente que tuviera como objeto la manipulación de elementos geométricos. Se ha mantenido en ésta con las debidas advertencias.
- Uno de los métodos de la clase, `verify`, contiene una primera aproximación a la gestión de errores.
- La clase `sensorTable` descrita -el código que contiene- es muy poco eficiente, tanto por el algoritmo de cálculo propiamente dicho (ver problema al final de este punto), como por la organización de la tabla de puntos y/o los propios algoritmos de búsqueda dentro de la tabla: cada vez que se usa -el algoritmo- se realiza una búsqueda secuencial (desde la primera posición de la tabla hasta que lo encuentra -el punto- o se acaba la tabla), lo que, si la tabla es grande, puede llegar a ser

inaceptable. Hay varios modos de hacer el cómputo total -el cálculo del valor correspondiente a la abcisa por medio de la tabla- mucho más rápido: la búsqueda *binaria* en la tabla -en lugar de la descrita, la secuencial-, por ejemplo. Queda fuera del contenido del blog analizar este asunto con más profundidad; baste indicar que, gracias al encapsulamiento que proporciona la OOP, las mejoras en este sentido se pueden hacer sin modificar el código de cliente (Véase como ejemplo, de nuevo, la solución al ejercicio propuesto a continuación: se modifica la clase pero su *interface* -el método *lookup*, en este caso- no varía).



La clase *sensorTables* tal y como está elaborada es poco eficiente en términos de tiempo de ejecución: cada vez que hay que convertir una *entrada* el algoritmo es:

1. Buscar en la tabla los dos puntos que rodean al pasado como argumento: métodos *pointer* y *abcissaPoints*.
2. Calcular ("m" y "b") la línea que los une y la ordenada correspondiente a la abcisa: *linearInterpolate*.

Como ya se ha explicado, esto es ineficiente (lento):

1. Calcule (instanciando *sensorTables* con la tabla NTC_ADS1115.txt y ejecutando el algoritmo que implementa el cálculo de la ordenada en función de la abcisa un número elevado de veces; ver ejercicio al final de 4.1.11) el tiempo medio que tarda el algoritmo programado en calcular una ordenada.
2. Olvídense de todo lo anterior:
 1. Elabore una clase *line* que reciba como argumento dos puntos de diferentes abcisas; en la inicialización -instanciación- de los objetos *line* calcule los parámetros "m" y "b" de la linea, que quedarán como atributos del objeto: *self.m* y *self.b*.
 2. En la instancia -del objeto- de la clase *sensorTables*, instancie tantas líneas como las $n - 1$ parejas de puntos consecutivos de la tabla; guarde las instancias en una lista (*self.lineas*, por ejemplo).
 3. Cuando la clase *sensorTables* tenga que calcular -*lookup*- la salida correspondiente a una entrada -*abcisa*-, tire de la lista del apartado anterior: tendrá que usar *pointer* para saber qué línea necesita, pero, ahora, basta con modificar el método *lookup* a algo así:

```
puntero = self.pointer(abcissa) - 1  
return self.lineas[puntero].m * abcissa + self.lineas[puntero].b
```

para tener una ejecución algo más rápida (se evita el método *linearInterpolate*).

NOTA: Necesita una clase casi igual, pero distinta, de la planteada (*sensorTable*) que implemente lo dicho. Lo suyo sería crear una clase *ancestro* de la clase *lenta* y de la nueva que diferirían en cuatro cosas (el método *lookup* una de ellas). Déjese de *virguerías* y cree otra clase diferente (*?sensorTable7?* -el "7" porque estamos en el capítulo 7-) para resolver el ejercicio.

Solución: Alrededor (en un raspberry Pi 2) del 30% (Variará ligeramente dependiendo de la tabla, los valores, etcétera. Puede variar, incluso, de una prueba a otra).

8.- Uso de la librería.

La librería ya está completa. Se ha hecho un buen trabajo: creando las clases de los buses (ADCBus, ADS1115, oneWire, arduino), las clases *agente de cálculo* y la clase sensor, podemos, simplemente instanciando los buses y sensores que tenemos conectados, manejarlos con mucha facilidad. Si contemplamos el trabajo desde el punto de vista de los métodos -las “instrucciones” - que hemos añadido al sistema (ver 3.2, “Las clases como proveedoras de servicios”), tendríamos:

Buses
singleShot():
método para solicitar al ADC que comience una conversión.
ready():
Comprobar si el ADC ha terminado la conversión.
readConversion():
Lee, y retorna, la última lectura del ADC.
setChannel(channel):
Establecer canal activo.
setChannelGain(sensor):
Establecer canal activo y, si el ADC tiene la opción, la ganancia correspondiente.

En cuanto a los sensores, gracias a la familia de clases de agentes de cálculo que nos hemos sacado de la manga (y a la estructura que le hemos dado a la propia clase sensor; ver 5.4, “código de la clase ‘sensor’. Instanciación.”), basta con que esta -la clase *sensor*- tenga un único método *readout* para que nos dé en forma legible -*human readable*- la lectura correspondiente al sensor en cuestión. No está mal.

Vayamos a un caso práctico.

8.1.- Uso básico.

Partiremos del siguiente supuesto:

1. Placa ADS1115 en la dirección 0x48 del bus i2c (placa *china*. Ver figura 13)

2. Termistor (sensor de temperatura no lineal) de la propia placa conectado al canal “1” del ADS1115 (Ver figura 19). La tabla del sensor -la conversión de la tensión que lee el ADS1115 a la temperatura correspondiente- es la contenida en el fichero ADS1115_NTC.txt; ver 7, “Tablas de sensores [sensorTable]”.
3. Sensor de temperatura LM35 conectado al canal “2” de la placa -del ADS1115-.

Por lo demás, se trata de visualizar las dos temperaturas



Uso de este blog como recetario -“cookbook”-: El código que tenemos que escribir a continuación para ver las lecturas de los sensores es prolífico. No se alarme, cuando acabemos este punto del blog -el 8, “Uso de la librería”- veremos que la librería se puede usar, casi, con una o dos instrucciones. Lo que pasa es que este blog está pensado para aprender.

Se hace esta advertencia para quienes se hayan saltado todo lo anterior (quizá viene usted del punto 1, “Presentación”, donde se sugiere eso, que se salte usted lo anterior. Bienvenido si es así).

No obstante, más adelante (8.3, “Uso mediante clases ‘user’ . Ficheros de configuración.” y 8.4, “Uso conjunto con EIDEGraphics”) la cosa se parece más al citado *cookbook* que, quizás, sea lo que usted anda buscando.

Volvamos al asunto. Usted es, a partir de ahora, *el cliente*.

No se asuste: se trata de que todo el código desarrollado hasta ahora, la librería propiamente dicha, está en el fichero *EIDEAnalog.py*; usted está escribiendo su propio fichero “.py”, un *script*, un programa, vamos. Llámelo como quiera.

8.1.1.- Instanciación directa de buses y sensores.

Así que lo primero que tiene que hacer su flamante *código de cliente*, esto es nuevo -pero completamente lógico-, es importarla -la librería-:

```
import EIDEAnalog
```

Ya tenemos a nuestra disposición todas las clases -y sus métodos- que nos hemos molestado en elaborar previamente.



Clase services: La agrupación de todas las clases descritas hasta aquí en un solo fichero (*‘EIDEAnalog.py’*) no supone ningún cambio en cuanto a su estructura interna -la de las clases-. Por una cuestión de *ortodoxia* se ha creado una clase *services* -servicios- que contiene A) los datos que pueden ser usados por varias clases y B) los servicios comunes a las clases existentes (y futuras). En concreto:

- **Datos de sensores:** en la versión de la librería que se está desarrollando están contenidos en un diccionario ‘*standardSensorsData*’ que se aloja en esta clase. Probablemente en una próxima

revisión de la librería los datos de los sensores estarán en un fichero de texto (txt) que será leído en la definición de la clase -o en la instancia-; como es una clase que tendrá una sola instancia da lo mismo- y cargado -el contenido del fichero- en el diccionario mentado. Esto último es mucho más práctico, ya que el cliente no necesita entrar en el código de la librería si quiere añadir tipos de sensores nuevos.

- **‘Ruta’ de acceso a ficheros:** se incluye una atributo de clase *-tablesFolder-* con esta información para las tablas de sensores.
- **Instanciación del servicio i2c:** no es descartable que aparezca otro ADCBus que lo necesite (no que haya otro ADS1115 -la única clase que, de momento, usa este servicio-, sino que haya otro conversor que se comunique vía i2c con el Raspberry).

La clase se instancia inmediatamente después de definida:

```
srv = services()
```

y todos sus atributos -datos y métodos- son accesibles como `srv.xxxxxx`.

(Esta clase -su instancia- está prevista para que sea usada solamente por las instancias de las clases de la propia librería; no obstante, están igualmente accesibles desde el módulo que importa la librería).

Para activar el bus -la placa ADS1115- hay que instanciarla:

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

hemos creado una instancia del bus ADS1115 que está en la dirección 0x48 del bus i2c. Vaya a 2.3, “Conversores ADC” si no entiende nada.

Con esto ya tenemos todas las órdenes -métodos- de la clase ADS1115 a nuestra disposición; si, por ejemplo, queremos seleccionar el segundo canal del ADC, la orden será:

```
myBus.setChannel(2)
```

(No estamos contando nada nuevo; esto mismo ya se explicó en el punto 4.1.11, “Clase ADS1115 completa. Ejemplo de uso”; vuelva allí si no tiene meridianamente claro lo que se está explicando).

Vamos, ahora, con los sensores:

Para instanciar el NTC la orden será:

```
NTC_china = EIDEAnalog.sensor('ADS1115_NTC', myBus, 1, name='NTC_placa')
```

El nombre del objeto -“NTC_china”- y el del sensor -“NTC_placa”- son completamente arbitrarios.

El tipo de sensor, sin embargo, tiene que ser el que tiene que ser, de lo contrario -si el sensor no es de un tipo *reconocido*- se producirá un error (ver 5.4, “código de la clase ‘sensor’. Instanciación”; ‘standardSensorsData’).

La instancia del LM35 es:

```
T_Exterior = EIDEAnalog.sensor('LM35', myBus, 2, name='T_Exterior')
```

Supongamos que estamos (com)probando lo anterior: usted quiere verificar que, en efecto, el NTC mide la temperatura. Su programa tiene que -además de lo anterior-:

1) Seleccionar en el bus el canal 1

```
myBus.setChannel(1)
```

2) Seleccionar como tensión de referencia para la lectura 4,096 V (porque en la tabla de datos de los sensores *-standardSensorsData-* el NTC tiene como tensión de referencia 3.3 V; el valor inmediatamente superior de la lista de tensiones de referencia del ADS1115 es 4,096 V).

```
myBus.setGain(4)
```

(El lector avezado -y aplicado- quizá esté pensando que los dos pasos anteriores se pueden hacer en uno sólo *-"setChannelGain"*: es cierto, pero cada cosa a su tiempo, siga leyendo).

3) Seleccionados canal y ganancia, le decimos al ADS1115 que convierta:

```
myBus.singleShot()
```

4) Nos ponemos a la espera ...

```
while not(myBus.ready()):  
    pass
```

5) leemos:

```
lectura = myBus.readConversion()
```

6) Y convertimos y visualizamos

```
a_visualizar = NTC_china.readout(lectura)  
print (a_visualizar)
```

Si usted ha hecho exactamente lo que se le ha ido diciendo que haga, en este momento estará viendo en el IDLE de Python un número que es la temperatura que está midiendo el NTC de la placa del ADS1115. Resumamos por si acaso:

1) Usted tiene un fichero que se llama “EIDEAnalog.py” cuyo contenido se ha descargado de Internet - <https://github.com/Clave-EIDEAnalog/EIDEAnalog> -

2) En la misma carpeta que tiene el fichero anterior, usted ha guardado un *script* - programa- como el siguiente:

```
# Import and init EIDEAnalog.
```

```

import EIDEAnalog
EIDEAnalog.init()

# Instance bus and sensor.
myBus = EIDEAnalog.ADS1115(0x48)
NTC_china = EIDEAnalog.sensor('ADS1115_NTC', myBus, 1, name='NTC_placa')

# Configure and trigger ADC. Wait for bus ready.
myBus.setChannel(1)
myBus.setGain(4)
myBus.singleShot()
while not(myBus.ready()):
    pass

# Read conversion. Convert and display it.
lectura = myBus.readConversion()
a_visualizar = NTC_china.readout(lectura)
print (a_visualizar)

```

Y tiene que funcionar (está probado).



Ahora que ya somos personas serias e importamos librerías, viene bien que vayamos adquiriendo buenas costumbres: la librería, como ya se ha explicado muchas veces, pone en marcha automáticamente, al importarla, algunos recursos del Raspberry que conviene que, una vez acabado el trabajo con aquella -la librería-, queden liberados de nuevo. Aunque no sea más que para cerrar la aplicación ordenadamente, conviene que la librería tenga una función que desactive dichos recursos. La orden

EIDEAnalog.quit()

sirve justamente a tal efecto. De momento póngala al final del código.

Si quisieramos hacer lo propio con el LM35, el código sería:

```

# Instance sensor
T_Exterior = EIDEAnalog.sensor('LM35', myBus, 2, name='TE_LM35')

# Configure and trigger ADC. Wait for bus ready.
myBus.setChannel(2)
myBus.setGain(2)
myBus.singleShot()
while not(myBus.ready()):
    pass

# Read conversion. Convert and display it.
lectura = myBus.readConversion()
a_visualizar = T_Exterior.readout(lectura)
print (a_visualizar)

```

que se puede añadir a continuación del anterior y le mostrará, también directamente en °C la lectura del LM35.



Modifique el código precedente para el bus *oneWire*. Suponga que tiene dos sensores DS18B20 conectados.

8.1.2.- Uso con *lista de sensores*.

Lo anterior se puede mejorar, y mucho.

De entrada, tal y como se advirtió, hay un método en la clase ADCBus -"setChannelGain"- que, como su propio nombre indica, configura de una tacada el canal a leer y la ganancia correspondiente. Recuerde que en el diccionario de sensores hay una referencia a la tensión máxima que cabe esperar del sensor (por ejemplo, el LM35 lo tenemos a 1,0 V) y en la clase ADCBus hay una lista con las tensiones de referencia que admite -el ADC-: al instanciar el sensor se consulta ésta y el valor pasa a ser un atributo (dos, en realidad: *vRef* y *vRefPos*) del objeto sensor.

Con este bagaje, basta invocar el método en cuestión con el objeto de la clase sensor como argumento (importantísimo) para que el método en cuestión se encargue de todo.

(Se repite a continuación nota pertinente)



Pasar un objeto como argumento en la llamada a un método tiene ciertas implicaciones de tipo más teórico que otra cosa en OOP; digamos, simplificando, que el método puede modificar el objeto en cuestión, lo que va contra uno de los paradigmas de la OOP. Como ya se ha advertido en más de una ocasión en Python es imposible poner el *interior* de los objetos a salvo del *programador loco* (otro día explicamos lo de este tipo de programadores), por lo que se entiende que con este pequeño aviso está todo dicho.

De esta forma, los dos pares de líneas

```
myBus.setChannel(1)  
myBus.setGain(4)
```

```
myBus.setChannel(2)  
myBus.setGain(2)
```

pasarían a ser

```
myBus.setChannelGain(NTC_china)  
myBus.setChannelGain(T_Exterior)
```

lo que es un pequeño ahorro y una mejora sustancial en la legibilidad del software.

En todo caso, el código sigue siendo demasiado engorroso. Hay una posibilidad de simplificación evidente: si analiza los dos bloques vera que se repiten casi línea por línea,

salvo, precisamente, las dos últimas mencionadas, y la de `sensor.readout`, que son diferentes para cada sensor.

Python es mágico para esto: tiene la herramienta perfecta, las listas. Una lista, usted lo debería saber ya, es una relación de cosas -números, strings, *objetos*- . Las listas se pueden crear *sobre la marcha*, nosotros vamos a crear una lista de *objetos-objetos*, los sensores que vamos instanciando. Algo así:

```
instancedSensors = []
NTC_china = EIDEAnalog.sensor('ADS1115_NTC', myBus, 1, name='NTC_placa')
T_Exterior = EIDEAnalog.sensor('LM35', myBus, 2, name='TE_LM35')
instancedSensors.append(NTC_china)
instancedSensors.append(T_Exterior)
```

y ahora viene lo bueno: las listas de Python son iterables, es decir, se puede hacer lo siguiente:

```
for i in instancedSensors:
    i.sensorBus.setChannelGain(i)
    i.sensorBus.singleShot()
    while not(i.sensorBus.ready()):
        pass
    lectura = i.sensorBus.readConversion()
    a_visualizar = i.readout(lectura)
    print (a_visualizar)
```

de momento no lo complicaremos más (hay una cuestión evidente: añadir el tipo -o el nombre- del sensor para identificar la lectura; lo proponemos como ejercicio)



Añada al código anterior lo necesario para que la visualización de la lectura sea similar a:
. ADS1115_NTC: 24.1 °C
. LM35: 12.5 °C

Los rótulos (ADS1115, LM35) NO se deben incluir como tales en el “print”, sino como atributos del objeto sensor.

8.2.- Uso con clases intermedias. Clase “ADCBus_manager”.

Lo desarrollado al final del punto anterior es un avance, pero aún se puede simplificar más (si no hay problemas de *velocidad*: lea a continuación).

Un *cliente* de la librería puede que necesite la máxima velocidad a la que puede funcionar el ADC, en cuyo caso su código de aplicación podría tener un aspecto así:

```
lecturas = [0,0]
while True:
```

```

for counter, i in enumerate(instancedSensors):
    myBus.setChannelGain(i)
    myBus.singleShot()
    # Aquí el código de aplicación del cliente
    while not(myBus.ready()):
        pass
    lectura = myBus.readConversion()
    lecturas[counter] = i.readout(lectura)

```

En esta variante del código del final del punto anterior se ejecuta un bucle para capturar las lecturas continuamente; más allá de la sofisticación en la creación del bucle interno -*for counter,i in enumerate(instancedSensors)*- para tener el índice de la lista de lecturas -*counter*- siempre a mano, lo que se hace es aprovechar el tiempo que tarda el ADC en convertir para hacer lo que el cliente tenga a bien con las lecturas -"Aquí el código de aplicación del cliente"- que, bucle a bucle, se van actualizando en la lista *lecturas*.

Hay un uso posible que es algo menos exigente (y que, como veremos inmediatamente, da lugar a un código mínimo): supongamos que lo que quiere hacer el *cliente* no corre tanta prisa; en el ejemplo anterior, suponiendo que el código de aplicación -*cliente*- no lleve mucho más de unos μ s (lo que, por otra parte, es bien difícil con solo dos lecturas: a ver qué se puede hacer con dos tristes números que lleve más tiempo), el ritmo de lectura de datos podría llegar, para un ADS1115, a unas 400 (200 x 2) lecturas por segundo (algo más si tiene *tuneado* el raspberry -ver Anexo II, "Configuración de Raspbian" y ejercicio al final de 4.1.11-).

Si lo que se quiere hacer -el código de cliente- no requiere de tanta velocidad es posible que resulte suficiente con que el ADC convierta *todos* los sensores consecutivamente y rellene una lista con los resultados: en un caso como el que estamos analizando -dos sensores y un ADS1115- esto querría decir que el código de *cliente* se ejecutaría, aproximadamente, con la mitad de frecuencia, o sea, unas 200 veces por segundo en lugar de 400; si en lugar de dos sensores tenemos 4, el código de cliente se ejecutaría unas 100 veces por segundo.

Si lo que se quiere hacer es, por ejemplo, visualizar las medidas en displays (ver 8.3, "Uso conjunto con '*EIDEGraphics*'"), una velocidad de 24 veces -actualizaciones de los displays- por segundo es más que suficiente para el ojo humano: a esa velocidad una aguja de un indicador parece moverse de forma continua; si la visualización es numérica basta -y a veces es hasta molesto- con renovar la lectura un par de veces por segundo.

Propóngamonos, pues, resolver este problema de forma que un (*super*)*cliente* de la librería lo tenga todavía más fácil. Se trata de, en resumen:

1. Diseñar una nueva clase, *ADCBus_manager*, que haga de interface (en los textos a este tipo de clases se les suele denominar *wrapper* o *façade*) con las clases ya desarrolladas (*ADS1115* y *sensor*)
2. Que la instanciación de éstas sea automática en función de los argumentos que, a su vez, se pasan en la instanciación de la nueva clase *ADCBus_manager*.
3. Es suficiente con que la demanda de datos al *ADCBus* se pueda acomodar a una frecuencia tal que la -nueva- orden de conversión ponga en marcha la de los cuatro

canales que tiene de forma consecutiva para, al finalizar esta, demandar de la nueva clase -del objeto de la nueva clase- todas las lecturas.

Para fijar ideas, supongamos que tenemos la nueva clase ya elaborada. Podemos plantear que su uso sería algo así:

```
manager = ADCBus_manager(project)

while True:
    lecturas = manager.readConversions()
    # Aquí el código del cliente
```

tómese lo anterior como *pseudo-código* (no el código final, aunque lo puede parecer -y no será muy diferente-).

Un proyecto de *EIDEAnalog* siempre estará formado por uno o varios buses -ADCBus- a los que están conectados uno o varios sensores (por bus). Tanto los buses como los sensores tienen que ser de un tipo determinado (de no ser así, en el caso de los buses no tendríamos una clase para manejarlo y en el de los sensores no habría información en el diccionario de sensores).

Así, cuando se instancia la clase *ADCBus_manager*, se pasa como argumento “*project*”. “*project*”, sea cuál sea la forma final que se use para él -una lista de Python- tiene que contener los siguientes datos:

1. Tipo de bus. Datos operativos (i.e., la dirección de un ADS1115 en el bus i2c).
2. Sensores conectados al bus, con sus datos identificativos (tipo, canal del ADCBus y nombre).

Un boceto de *project* podría ser:

```
project = [
    'ADS1115', 0x48,
    ['NTC_china', 'ADS1115_NTC', 1, 'NTC_placa'],
    ['T_Exterior', 'LM35', 2, 'TE_LM35']
]
```

que, evidentemente, corresponde con el ejemplo del punto 8.1.1.

En definitiva, la lista *project* tiene tres miembros: los dos primeros especifican el bus, y los restantes, listas a su vez, dos en nuestro caso, los sensores que tiene conectados; cada sensor adicional requeriría una lista con idéntica estructura. Es evidente que con el contenido de *project* la nueva clase tiene toda la información que necesita para instanciar bus y sensores.

Al permitirlo la especificación (“*la demanda de datos al ADCBus se puede acomodar a una frecuencia tal que la -nueva- orden de conversión ponga en marcha la de los cuatro canales que tiene de forma consecutiva para, al finalizar esta, demandar de la nueva clase -del objeto de la nueva clase- todas las lecturas*”), el cliente de la clase -usted- sólo tendrá que llamar a un método de ésta, digamos

```
lecturas = manager.readConversions()
```

como ya se ha explicado más arriba.

Hagamos un boceto-análisis del código de la clase ADCBus:

1. **parseProject**: Un método de la clase deberá desmenuzar la lista que se pasa como argumento y que contiene los datos del bus y de los sensores. (En la jerga informática a esta acción se le llama *parse*).
2. **InstanceObjects**: Con la información estructurada por el método anterior, este método instanciará el bus y los sensores.
3. **readConversions**: finalmente, este método ejecuta un bucle sensor a sensor para leer las sucesivas conversiones, almacenarlas en una lista y *retornarlas*.

Veamos el código (comprobado) de la clase

```
class ADCBus_manager():  
    busesDict = {'ADS1115': EIDEAnalog.ADS1115,  
                 'oneWire': EIDEAnalog.oneWire,  
                 }
```

- Este diccionario contiene los nombres de las clases indexadas por los nombres que pasa el *cliente* al instanciar el objeto. Los nombres de las clases vienen precedidas por '*EIDEAnalog*' porque se trata de clases que han sido importadas del módulo *EIDEAnalog*.

```
def __init__(self, project):  
    self.sensorsCName = []  
    self.sensorsType = []  
    self.sensorsChannel = []  
    self.sensorsName = []  
    self.instancedSensors = []  
    self.project = project  
    self.parseProject(self.project)  
    self.instanceObjects()
```

- Se inician las listas que alojarán los datos del proyecto y se ejecutan los métodos *parseProject* e *instanceProject*.

```
def parseProject(self, project):  
    for i in project:  
        if isinstance(i, list):  
            # Sensor list  
            self.sensorsCName.append(i[0])  
            self.sensorsType.append(i[1])  
            self.sensorsChannel.append(i[2])
```

```
    self.sensorsName.append(i[3])
```

- Se extraen los datos de los sensores de la lista *project*. (*if isinstance(i, list)* descarta los datos que *no* van dentro de listas; los del proyecto)

```
def instanceObjects(self):
    self.bus = ADCBus_manager.busesDict[self.project[0]](
        self.project[1])
    for contador, i in enumerate(self.sensorsCName):
        objeto = sensor(self.sensorsType[contador],
                        self.bus,
                        self.sensorsChannel[contador],
                        self.sensorsName[contador])
        self.instancedSensors.append(objeto)
```

- Se instancia el bus (con los dos primeros datos de *project*) y los sensores (con las listas rellenadas en *self.parseProject*).

```
def readConversions(self):
    lecturas = []
    for i in self.instancedSensors:
        self.bus.setChannelGain(i)
        self.bus.singleShot()
        while not(self.bus.ready()):
            pass
        lectura = self.bus.readConversion()
        lecturas.append(i.readout(lectura))
    return lecturas
```

- Este método ejecuta un bucle, sensor a sensor, en el que lee las conversiones de los canales a los que están conectados los sensores, las transforma -*readout*- las almacena en una lista y las retorna.

Y ya está. Una vez definida esta clase, el siguiente código:

```
manager = ADCBus_manager(project)
while True:
    lecturas = manager.readConversions()
    # Aquí el código del cliente
```

leerá continuamente los canales del ADS1115 y los pondrá a disposición del *cliente* para que este haga con ellos lo que tenga por oportuno y conveniente (# *Aquí el código del cliente*). Ni que decir tiene que, faltaría más, después de todos los esfuerzos para que el interface de las clases sea el mismo, bastaría con un *project2*:

```
project2 = [
    'oneWire', 19,
    ['DS18B20_1', 'DS18B20', 1, 'DS18B20_1'],
```

```
[ 'DS18B20_1', 'DS18B20', 2, 'DS18B20_1' ]
```

como este para que exactamente el mismo *ADCBus_Manager* -el mismo- gestione el funcionamiento del bus oneWire. No está mal para un código de apenas tres líneas y una lista con los datos imprescindibles para identificar bus y sensores.



(a) Un raspberry tiene conectado un ADS1115 en la posición 0x48 del bus i2c con dos sensores como los descritos en el ejemplo anterior y activo el bus oneWire alimentado por la patilla 19 del GPIO y con dos sensores DS18B20. Escriba un código cliente que lea todos los sensores y muestre sus lecturas. No hay prisa. (“No hay prisa” para cómo hacer el programa, no en hacerlo -o no-; use la clase *ADCBus_Manager* descrita recién)



(b) Tal y como se explicó en el apartado 8.1.2, “Uso con *lista de sensores*”, se puede usar la librería iterando una *lista de sensores*. Si no ha resuelto así el ejercicio anterior, resuévalo ahora creando una *única* lista de sensores (los dos del ADS1115 y los dos del oneWire) e iterando sobre ella. (Hay varias formas de resolver el problema: creando dos instancias de *ADCBus_manager* -que es lo que se sugería para el ejercicio anterior- y la que se pide en éste: hágalo de las dos formas).

8.3.- Uso de la librería EIDEAnalog. Conclusiones.

- La librería EIDEAnalog se ha desarrollado haciendo un esfuerzo para que los elementos que la componen sean homogéneos -su *interface*-; se ha conseguido con esto simplificar mucho el uso de la misma.
- Se puede hacer uso de la librería haciéndola pivotar sobre la(s) clase(s) *ADCBus* instanciada(s) y/o sobre los sensores instanciados indistintamente (ver punto 8.2). Es posible, incluso, iterar con el mismo código una lista de sensores que estén conectados a diferentes buses (ADS1115 y oneWire, por ejemplo).
- En determinadas condiciones, se pueden desarrollar clases intermedias que faciliten aún más el uso de la librería; una de estas permite capturar los datos con una única línea de código.

9.- Conclusión. Continuidad.

- Se ha desarrollado, y, lo que es más importante, justificado el desarrollo, una librería para facilitar la adquisición -captura- de datos con Raspberry usando A) uno de sus buses nativos -oneWire- y B) otras dos opciones usando hardware complementario: la que, probablemente es la opción más popular para el Raspberry, el ADS1115 y otra con un arduino ProMini.
- Se ha hecho mucho énfasis en la arquitectura de la librería, en su concepción.
- Se presenta el código completo de la librería que, además, está convenientemente probado.
- Aunque visto el trabajo desde la perspectiva del lector (hecho y funcionando) pueda parecerlo, no es éste un proyecto trivial, al menos en cuanto a la arquitectura del mismo. Es cierto que las clases para buses son casi inmediatas, y de hecho muchos de los métodos han sido *corta-copiados* de WEBs que tienen código similar (se reconoce sin ningún rubor: varios de los métodos de la clase ADS1115 son los de los -excelentes- contenidos en github.com del usuario *saper*), pero la arquitectura del resto -agentes de cálculo y la clase sensor- tienen su *miga*, y, además, son completamente originales. Y si lo duda eche un vistazo por Internet a ver si encuentra algo similar.

Sin entrar en mucho detalle, la estructura final de la librería ha sufrido cambios durante su desarrollo (algunos de ellos se sugieren en las explicaciones que se van dando); hay numerosas soluciones posibles, de las que hemos seleccionado la que, a nuestro juicio, es la más flexible de todas ellas.

- La librería tiene cierta complejidad: aunque las clases de los buses se pueden instanciar independientemente y tiene un análisis relativamente simple, las clases *sensor* y las de los *agentes de cálculo* -su arquitectura- son más complejas. La clase *sensorTable* (punto 7), sin embargo, tiene un análisis simple. Si usa el blog como ayuda al aprendizaje de OOP, puede emplear el diseño y la codificación de esta clase como un ejemplo independiente -y fácil- de OOP.

Continuidad.

Se continuará la publicación de la librería con las siguientes tareas:

- **Versión 2.0:** con idéntica funcionalidad pero con tratamiento de errores.
- **Clase ADS1115_threaded:** la clase para el ADS1115 que incorpora esta release - 1.0- no permite que el cliente ejecute una tarea mientras que los canales son convertidos uno a uno (sí si se espera a que se conviertan los cuatro, ver 8.2, “Uso con clases intermedias. Clase ‘ADCBus_manager’”). Esta nueva clase permitirá

que el código de cliente se ejecute simultáneamente al sondeo continuo de los cuatro canales del ADS1115.

- **Módulo EIDELogger:** que genere ficheros de texto (compatibles con excel) con los valores de los canales de medida (data-logger).
- **Módulo EIDELog:** que permitirá elaborar un registro del funcionamiento de EIDEAnalog (configuración de arranque, errores, avisos).
- **Módulo EIDEAlarm:** para fijar condiciones de alarma -umbrales- asociados a los canales de los ADC y activar salidas del Raspberry en caso de que se excedan los umbrales para activar elementos externos (señalizaciones acústicas y/o ópticas).
- **Módulo EIDEIoT:** que permita transmitir vía Internet las medidas en tiempo real.
- **Nueva release de EIDEGraphics:** que admita más información que la puramente numérica (la versión 1.0, la actual, solamente convierte a información gráfica - indicadores- los números que se la pasan como argumento. Ver Anexo III.- Uso conjunto con 'EIDEGraphics'), de forma que se puedan visualizar textos, simular LEDs de alarma y/o actividad, etcétera, de forma que se pueda configurar un sinóptico.

Anexo I.- Descarga de EIDEAnalog.

En la barra de direcciones del navegador copie la siguiente dirección:

<https://github.com/Clave-EIDEAnalog/EIDEAnalog>

Aparecerá la siguiente pantalla.

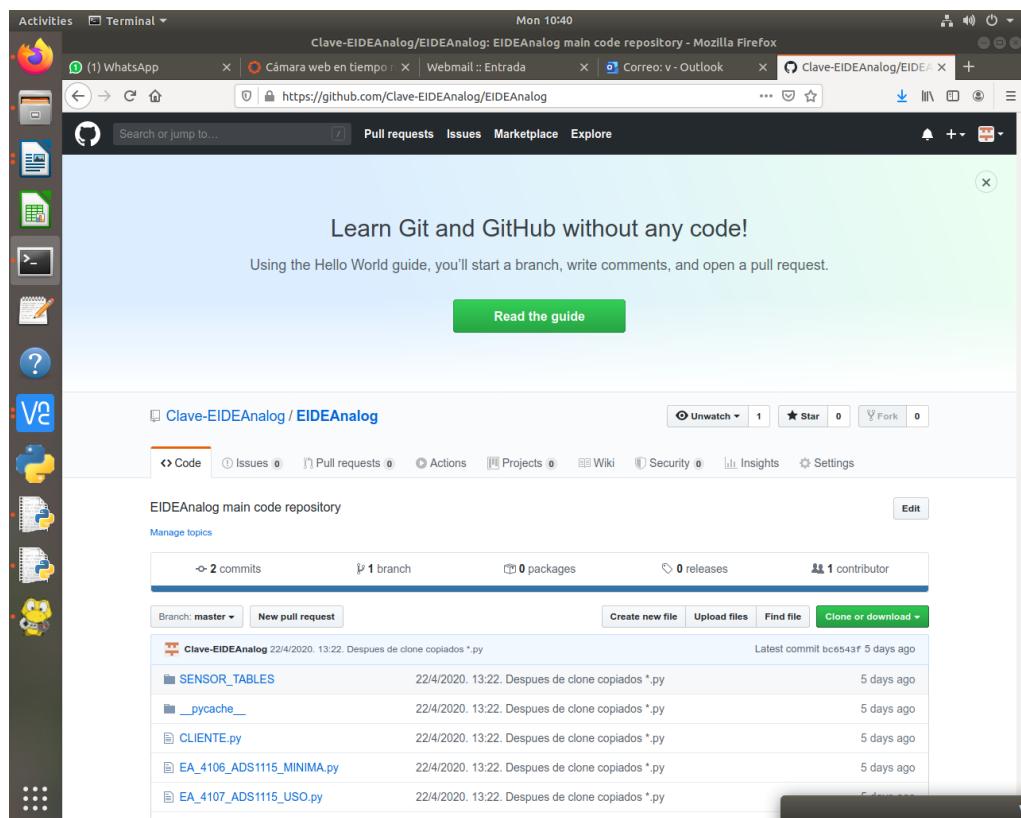


Figura I.1. Página de descarga de EIDEAnalog.

Seleccione “Clone or Download” (a la derecha; botón verde). En su carpeta de descargas aparecerá el fichero “EIDEAnalog-master.zip” que deberá descomprimir (zip, 7zip, ...).

Cuando termine se encontrará con la siguiente estructura de carpetas:

```
.. |EIDEAnalog-master
    |
    +-- SENSOR_TABLES
        +-- ...
    |
    +-- __pycache__
        +-- ...
    |
    +-- EIDEAnalog.py
    +-- EIDEAnalog_4106_ADS1115_MINIMUM.py
    +-- ...
```

La librería no necesita de instalación. Basta con que ejecute cualquiera de los ejemplos que incluye (EIDEAnalog_NNNN_*.py) y, previsto que tiene usted conectado el hardware que necesite el ejemplo (y el Raspbian debidamente configurado, ver Anexo II a continuación), obtendrá la salida correspondiente al ejemplo que haya seleccionado.

El código de la librería está en un *repositorio* (el término es feísimo: quizá *depósito* o *almacén* sería más correcto) de *github*. *github* es un sitio WEB que permite, entre otras muchas cosas (muchísimas: en realidad usar *github* solo para depositar código para que otro se lo descargue es usar un *fórmula 1* para ir al trabajo), almacenar código de programas y que el interesado se lo descargue con facilidad. Puede usted hacerlo como se ha indicado más arriba -*download*- o puede, si lo desea, entrar en el fichero que quiera de los que se muestran y hacer un *corta-pega*.

Anexo II.- Configuración de Raspbian.

II.1.- Instalación básica. *Distribuciones*.

En Linux se le llama *distribución* a lo que podríamos en ciertos contextos llamar *versión*. No se trata, como en windows, de retorcer una vez más la anterior -versión- (ya saben, de la 95 a la 98, de la 98 a la 2000, de la 2000 a la XP, etcétera) para que siga funcionando con los mismos problemas (si no mayores) que la anterior pero que, eso sí, la ciudadanía se dé de bofetadas en las tiendas del ramo para llevársela a casa.

En realidad el *núcleo* de Linux cambia poco con el tiempo: se le van añadiendo algunas cosas -el interface tipo windows, por ejemplo- pero no compulsivamente, sino que, en general, son modificaciones/mejoras necesarias. Además, un *consejo de sabios* vela porque las adiciones sean eso, necesarias y que, además, estén bien hechas y probadas. Añádale al conjunto que es gratis y ya es *la leche*.

Las distribuciones -los *millennials* las llaman *distros*- son, en realidad, las diferentes versiones del S.O., pero no tanto porque sean diferentes en cuanto a prestaciones, sino porque están adaptadas a diferentes máquinas, ordenadores -procesadores-; es por eso que, mientras que windows no funciona en un *apple* o el macOS no funciona nada más que en máquinas *apple*, Linux funcione en casi todas.

En general la instalación de los S.O. basados en Linux se hace siempre de la misma forma: de algún sitio WEB uno se descarga la distribución que corresponda con la máquina que se tiene; ésta -la distribución- suele estar en forma de un *fichero imagen* ("img") que hay que *copiar* al dispositivo de almacenamiento desde el que va a arrancar el ordenador de que se trate; también puede ser que sea -la imagen- un fichero en parte ejecutable, de forma que, montado de alguna manera en el ordenador que se pretende configurar para Linux (por lo general un DVD o, cada vez más frecuente, un lápiz de memoria USB), basta con tener un mínimo de arrojo para interrumpir el *bootstrap* del ordenador (darle a "F2" o "F11" cuando está arrancando), decirle que arranque desde el USB (o DVD) y, a partir de ahí, es coser y cantar. Si tiene algún trasto viejo por casa, pruebe a instalarle Linux (Ubuntu es una de las distribuciones más comunes): verá lo que es bueno.

II.2.- Instalación de *Raspbian*.

Configurar el Raspberry es más fácil: se puede hacer de una forma parecida a lo descrito en el párrafo anterior, aunque lo más frecuente es hacerlo directamente en la µSD de la que va a ir dotado (el Raspberry NO tiene de origen ningún dispositivo de almacenamiento. La imprescindible µSD puede tener hasta 128 Gb. El S.O. ocupa unos 7 Gb). El procedimiento es aproximadamente (recuerde que este blog no es un *tutorial*; siga las instrucciones en la propia WEB de Raspberry):

1. Descargar la imagen desde la WEB de Raspberry:
<https://www.raspberrypi.org/downloads/>. Déjese de NOOBS y de pamplinas y descargue el *Raspbian* directamente.
2. Pasarla -la imagen- a la µSD (digamos, para entenderlos que la imagen se *descomprime*; al proceso también se le llama a veces *flash* -la µSD-): esto, si no tiene otra opción (lo que suele ser bastante común) lo puede hacer en una máquina windows; el programa que necesita es el *Win32DiskImager* (hay otros. También lo puede hacer con un apple o con un ordenador que funcione con Linux).
3. Montar la µSD en el Raspberry y encenderlo; tiene que estar conectado a Internet para que el propio Raspberry termine la instalación: en general, diga que sí (*continuar*) a todo lo que le pregunte.

Por cierto, Raspbian -a veces Debian; es casi lo mismo- es la distribución de Linux para Raspberry. La normalización de los nombres de las cosas no es una de las virtudes del mundo de Linux, nadie es perfecto. Así, puede ocurrir que la misma distribución tenga diferentes nombres (o que las diferencias sean mínimas, casi cosméticas), o tenga alias más o menos confusos: la distribución de Raspbian que se descarga desde la segunda mitad de 2019 es el 'Buster' (así, con 'u', para sorpresa de hispanohablantes).

Por si se complica la instalación, ahí van algunas WEBS de ayuda:

<https://www.raspberrypi.org/downloads/raspbian/>

<https://raspberrytips.com/install-raspbian-raspberry-pi/>

<https://honeWirechoo.com/g/ywmxmza2ndf/raspbian-buster-install-or-upgrade>

II.3.- Personalización de *Raspbian*.

Cuando haya acabado la instalación, su Raspberry parecerá un ordenador hecho y derecho: aparte de un interface que no tiene nada que envidiar al windows de hace tres o cuatro años (vaaale, seis o siete), puede usted navegar por Internet, tiene aplicaciones compatibles con office, reproducción de contenidos multimedia, etcétera.

Para que el contenido de este blog funcione en su Raspberry hay que añadirle un par de cosas a cómo lo ha dejado la instalación básica. Veamos.

II.3.1.- Bus i2c.

Para que funcione tiene que estar activada la correspondiente opción en el Raspberry (i2c "activo": figura II.1, dele sin miedo)

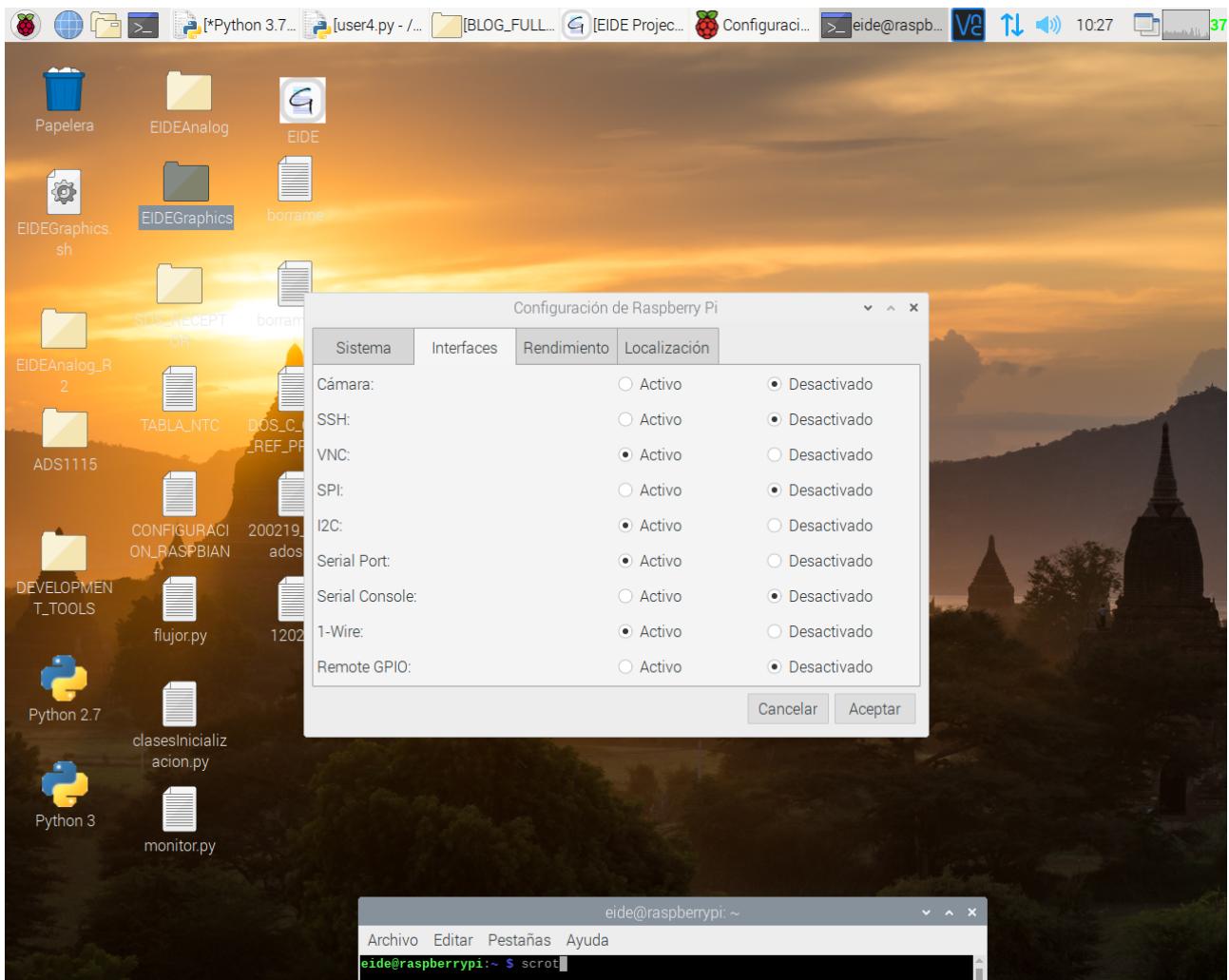


Figura II.1. Configuración del Raspberry para i2c activo.

y tener el “driver” correspondiente descargado. Si ya lo tiene el comando

```
$ i2cdetect -y 1
```

(“\$” es para indicar que está usted en la linea de comandos de Linux; si no se está enterando de *la misa la media* vaya a 1.2.3.- “Linux”)

le producirá un listado más o menos como este:

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
20:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
30:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
40:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	48
50:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
60:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
70:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

cualquier cosa que no sea parecida a lo anterior quiere decir que no tiene el driver instalado. Para ello tiene que hacer (mejor con la versión de Raspbian recién instalada como se indica en el punto anterior).

```
$ sudo apt-get install i2c-tools
```

conteste que sí (Yes) a todo y, cuando termine, apague, encienda y pruebe otra vez con el *i2cdetect*. Debería funcionar.

Funcionamiento del bus i2c a 400.000 bits/s. Tuneado del Raspberry.

Aunque los autores del blog tenemos ya una edad y no estamos para muchas goyerías, lo cierto es que el bus i2c no es precisamente rápido, y, quizás, conviene tomar cartas en el asunto: la velocidad del bus penaliza el funcionamiento (la velocidad aparente) de todo lo que esté conectado al Raspberry a su través. (Tampoco el ADS1115, que es lo que, en este blog, tenemos conectado al Raspberry a través del i2c es un *rayo*, pero lo cierto es que, ver Ejercicio en 4.1.11, “Clase ADS1115 completa. Ejemplo de uso”, se puede mejorar su velocidad -aparente- de conversión en más de un 50% cambiando la velocidad del bus; siga leyendo).

Por defecto el driver del i2c lo configura a 100.000 bits/s: si usted ha leído el ejercicio que se sugiere en el párrafo anterior, está informado de que el trasiego de órdenes entre el Raspberry y el ADS1115 hacen que la velocidad -aparente- de conversión se vea afectada por las órdenes previas y posteriores a la conversión propiamente dicha. También habrá visto que se puede configurar el bus para que vaya cuatro veces más rápido.

Para cambiar la velocidad del bus hay que toquetear un fichero que se llama “*config.txt*” y que está en la carpeta */boot* del Raspbian. Hay dos noticias, una buena y otra mala: la mala es que el procedimiento *ortodoxo* para modificar este fichero en el propio Raspberry es usando el editor de texto *vim* con privilegios de *superuser*. El *vim*, digan lo que digan los forofos, es una antigua de los tiempos de *maricastaña* (y eso que, que conste, los autores del blog lo usan a pesar de todo); lo de los privilegios del *superuser* es otra complicación (quién se haya aventurado por los caminos del Linux se habrá topado antes o después con la orden *sudo*, que a los hispanohablantes nos recuerda, inevitablemente, la dificultad de los primeros pasos en Linux). El asunto es que la orden para modificar *config.txt* desde el propio Raspberry es

```
$ sudo nano /boot/config.txt
```

con la que entra usted, bajo su propia responsabilidad, en el mentado procesador de textos (*vim*; que se le invoque como *nano* es otro misterio). A partir de aquí se conduce usted por su cuenta y riesgo. Ayúdese, si sigue por aquí, con esto:
<https://blog.desdelinux.net/usando-vim-tutorial-basico/>

La buena -noticia- es que el fichero *config.txt* es modificable, también, desde cualquier otro ordenador cuando se le inserta la µSD que está usted usando con el Raspberry (y es, también, un método *ortodoxo*). Para ello, tiene que, naturalmente, apagar el Raspberry, sacar la µSD e insertarla en el -otro- ordenador (necesitará, casi con toda seguridad, un

adaptador). Allí descubrirá el fichero en cuestión al que podrá meter mano sin *sudo* ni nada; en windows use el *notepad*:

Lo haga como lo haga, tiene que localizar una línea que es

`dtparam=i2c_arm=on`

y modificarla para que sea

`dtparam=i2c_arm=on,i2c_arm_baudrate=400000`

¡no deje espacios!.

Una vez hecho esto, vuelva a arrancar el Raspberry. Hecho.

Por si necesita más ayuda:

<https://learn.sparkfun.com/tutorials/raspberry-pi-spi-and-i2c-tutorial/all>

Pines de salida del bus i2c.

En la configuración de fábrica, el bus i2c sale por los pines GPIO02 y GPIO03 del conector del Raspberry. (Si está usando la placa de la figura 3 -el ADS1115 chino- puede ignorar lo anterior; son directamente los que usa la placa).

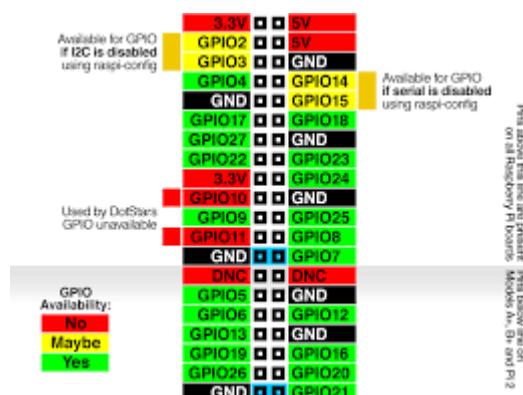


Figura II.2. Identificación de los pines del conector GPIO.

II.3.2.- Bus oneWire.

Para que funcione tiene que estar activada la correspondiente opción en el Raspberry (1-Wire “activo”: figura II.1, dele, aquí también, sin miedo).

Aunque en el caso del bus oneWire el driver siempre está instalado, hay que verificar -de nuevo con todas las instrucciones y avisos del punto anterior- que en el fichero *config.txt* aparezca la línea

```
dtoverlay=w1-gpio
```

(y ojo, de nuevo, con los espacios)

Pines de salida del bus oneWire.

Hay que tener en cuenta un par de cosas:

1. **Alimentación del bus (3.3 V).** Nominalmente el bus oneWire no necesitaría nada más que dos hilos: *tierra* (0 V, *masa*) y el del bus propiamente dicho. En la mayor parte de las ocasiones los sensores necesitan que se les *alimente*, con lo que hay que habilitar un tercer hilo. La configuración estándar -fábrica- usa el pin GPIO04 (ver figuras 11 y II.2) para el bus; para *tierra* cualquier de los 8 que lo son y para 3.3 V el “1”.
2. **Pantallas LCD de pequeño formato:** Para kits y aparatos de medida de pequeño formato son habituales las pantallas LCD de 3.5” (también de 3.2”; ver figuras 9 y II.3) que ocupan los 26 primeros pines del GPIO del raspberry. No solo el pin del bus oneWire (GPIO04), sino también el único pin de 3.3 V de todo el conector (el “1”) quedan, pues, inutilizados.

Hay una solución para poder usar el bus oneWire en estas condiciones. En primer lugar, se puede configurar el driver para que use otro pin para el bus (*señal*). Para ello, la línea de *config.txt* mencionada algo más arriba tiene que ser

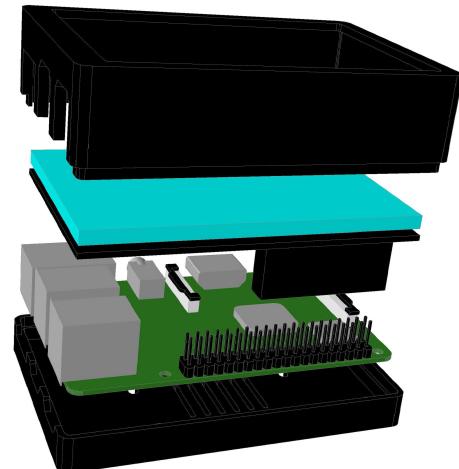


Figura II.3. Kit con pantalla LCD

```
dtoverlay=w1-gpio,gpiopin=20
```

Con lo que se usará el pin GPIO20, en este caso, para el bus. Además, se necesitan 3,3 V en algún sitio: la librería habilita, a tal efecto, el pin que se le indique en la instancia del bus oneWire: ver 6.1, “Clase oneWire” para más detalles.

II.3.3.- Conexión serie.

Tiene una secuencia parecida a las anteriores:

1. Figura II.1: “Serial port” activo.
2. En config.txt la siguiente línea:
 - enable_uart=1
3. Hay que cambiar también el fichero *cmdline.txt*. Hay que poner atención: contiene una sola línea con varios comandos de la que hay que eliminar la parte “console=serial0,115200”. NO DEJE ESPACIOS después de quitar esta parte de la línea. Todas las consideraciones previas acerca de hacerlo directamente en el raspberry o con la ayuda de otro ordenador siguen siendo válidas (ver II.3.1, “Bus i2c”).
4. Por fin, desde la línea de comandos del terminal hay que ejecutar estos dos comandos:
 1. sudo systemctl stop serial-getty@ttyAMA0.service
 2. sudo systemctl disable serial-getty@ttyAMA0.service

Apague y encienda el raspberry y ya está configurado. La conexión serie queda habilitada en los pines GPIO14 (Tx) y GPIO15 (Rx)

II.4.- Python.

En las últimas distribuciones de Raspbian vienen instaladas las versiones 2.17 y 3.X (cambie “X” por la más reciente) de Python. Hasta la versión “buster” (ver II.1, “Instalación básica. *Distribuciones*”) ambas versiones tenían un acceso directo desde el menú principal, “programación”.

El *buster* ha suprimido esta opción: a juicio de los responsables tenemos todos que arreglarnos con un interface de programación que se llama *Thonny* (y que sí aparece en el menú).

El *Thonny* en cuestión no está del todo mal, pero puede que los que llevamos tiempo con el IDLE (el entorno de programación) de toda la vida le hayamos cogido cariño. Si usted también quiere recuperarlo, haga lo siguiente:

1. Instálese el IDLE. Desde la linea de comandos del terminal “sudo apt-get install idle-python3”.
2. Desde la línea de comandos “idle-python3.X” y aparece el IDLE.

II.5.- µSD preconfigurada.

Hay bastantes opciones en Internet de hacerse con tarjetas µSD preconfiguradas para funcionar con el raspberry. Estas son algunas de las opciones:

[https://www.ebay.es/itm/Raspbian-buster-for-Raspberry-Pi-pre-complied-OS-with-OPENCV-4-and-Python-16GB/324046805030?
hash=item4b72b1d826:g:YLoAAOSwMWxel~hC](https://www.ebay.es/itm/Raspbian-buster-for-Raspberry-Pi-pre-complied-OS-with-OPENCV-4-and-Python-16GB/324046805030?hash=item4b72b1d826:g:YLoAAOSwMWxel~hC)

[https://www.ebay.es/itm/32-GB-microSD-card-loaded-with-Raspbian-Buster-desktop-and-recommended-software/324029370700?
hash=item4b71a7d14c:g:tRQAAOSwS~ReDOTi](https://www.ebay.es/itm/32-GB-microSD-card-loaded-with-Raspbian-Buster-desktop-and-recommended-software/324029370700?hash=item4b71a7d14c:g:tRQAAOSwS~ReDOTi)

[https://www.ebay.es/itm/Raspberry-Pi-Micro-SD-Card-32-GB-Preloaded-Raspian-OS-SanDisk-XC-Class-10-UHS-1/293516874409?
hash=item4456f812a9:m:mHSsKZMU0Son_fip5DgBrPw](https://www.ebay.es/itm/Raspberry-Pi-Micro-SD-Card-32-GB-Preloaded-Raspian-OS-SanDisk-XC-Class-10-UHS-1/293516874409?hash=item4456f812a9:m:mHSsKZMU0Son_fip5DgBrPw)

[https://www.ebay.es/itm/Raspberry-Tarjeta-micro-SD-de-16-Gb-con-Raspbian-Buster-preinstalado/163907615931?
hash=item2629a7bcbb:g:IroAAOSwApRdpuze](https://www.ebay.es/itm/Raspberry-Tarjeta-micro-SD-de-16-Gb-con-Raspbian-Buster-preinstalado/163907615931?hash=item2629a7bcbb:g:IroAAOSwApRdpuze)

(La µSD de este último vendedor tiene, además del S.O. propiamente dicho, activadas todas las opciones de los buses y el acceso a Python mediante un ícono según lo indicado en este anexo).

Anexo III.- Uso conjunto con 'EIDEGraphics'.

Lo más elemental que se puede hacer con la lectura de un sensor es mostrarla, y es lo que vamos a hacer. A tal efecto usaremos el módulo “EIDEGraphics”, cuya finalidad no es otra que esa, la de mostrar en forma gráfica (o numérica, en formatos grandes si se desea) los valores que se le pasen.

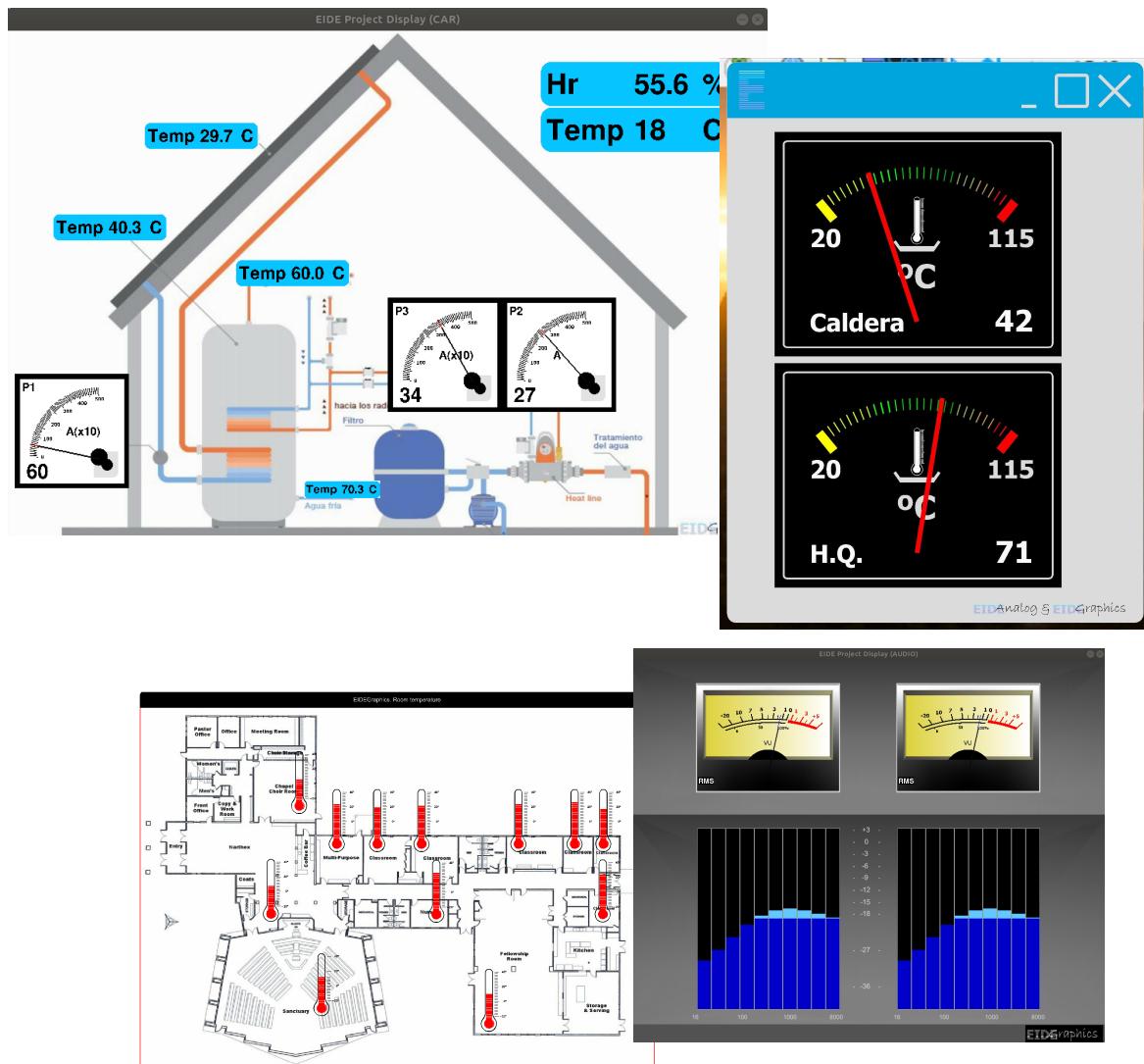


Figura III.1. Proyectos basados en EIDEGraphics.

EIDEGraphics es una librería cuya finalidad es únicamente la de mostrar en forma gráfica valores numéricos. Puede hacerlo como un gráfico de barras (o simular una barra de LEDs), con indicadores de aguja o con números de gran formato. Su versatilidad proviene del hecho de estar configurado a base de ficheros de texto por medio de los cuales el

usuario puede, sin tocar una sola línea de código, modificar los indicadores de acuerdo a sus necesidades; por ejemplo: convertir un indicador de aguja de aspecto industrial (Figura III.1; proyecto superior izquierda) en uno similar al de un vehículo (superior derecha) o un vúmetro (panel de audio). Además se pueden tener configurados tantos proyectos como se desee (los de la figura son cuatro ejemplos). También el fondo general de la pantalla es configurable por el usuario, con lo que se pueden elaborar (se insiste, sin modificar ni una línea de código) proyectos con aspecto de sinópticos. Gracias al diseño a base de capas, la velocidad de actualización de la pantalla puede llegar, incluso con proyectos muy poblados y con una CPU desde el modelo “raspberry Pi 2” (Figura III.1: panel de audio; son en total 22 displays diferentes) a más de 100 *frames* -planos- por segundo, más que suficiente para que la sensación sea completamente satisfactoria (para percibir el movimiento de una aguja o de una barra de LED como continuo basta con 24 planos por segundo). Es, en definitiva, la herramienta ideal para mostrar los valores capturados por *EIDEAnalog*.

III.1- Descarga e instalación de EIDEGraphics.

En la barra de direcciones del navegador copie la siguiente dirección:

<https://github.com/Vicente-Francisco/EIDEGraphics>

Aparecerá la siguiente pantalla.

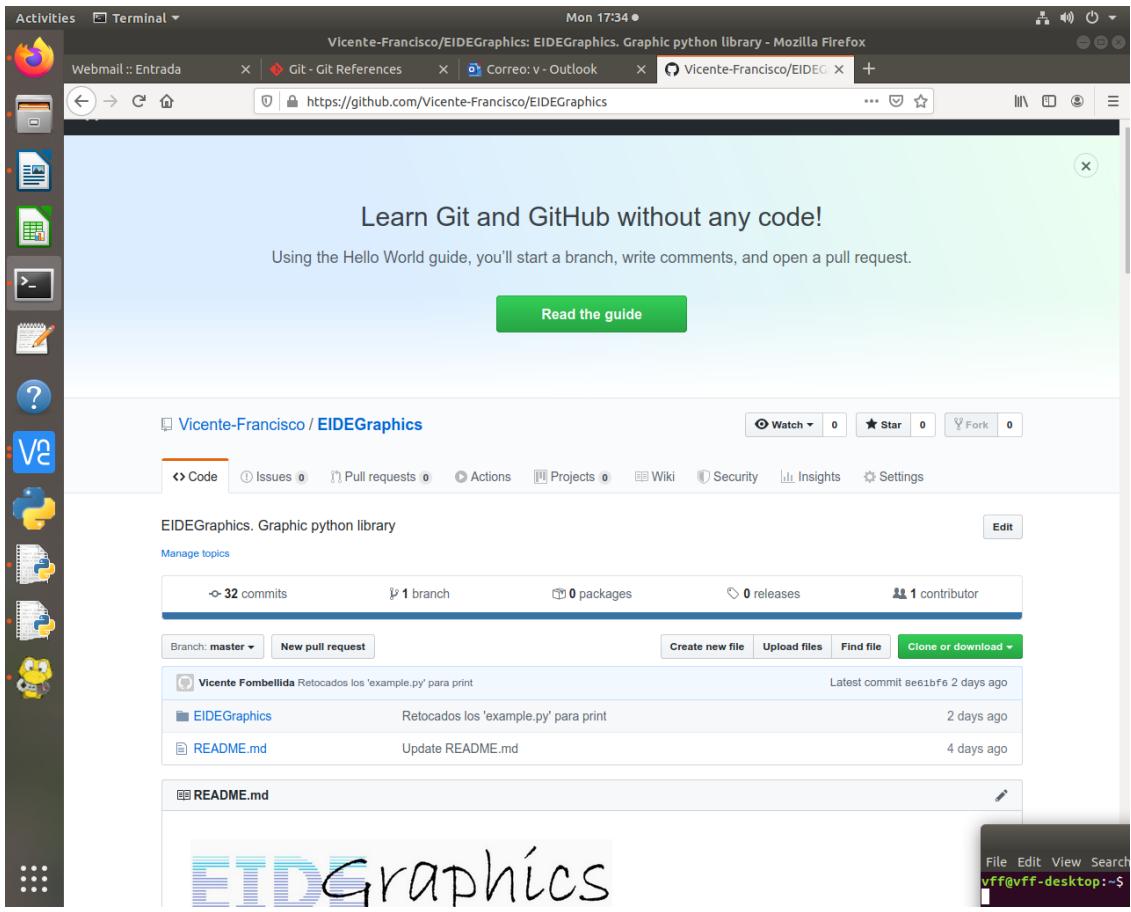


Figura 30. Página de descarga de EIDEGraphics.

Seleccione “Clone or Download” (a la derecha; botón verde). En su carpeta de descargas aparecerá el fichero “EIDEGraphics-master.zip” que deberá descomprimir (zip, 7zip, ...).

Cuando termine se encontrará con la siguiente estructura de carpetas:

```

.. |EIDEGraphics-master
   |
   +-- EIDEGraphics
       |
       +-- GRAPHIC_FILES
           |
           +-- ...
       |
       +-- PROJECTS_N_EXAMPLES
           |
           +-- ...
       |
       +-- __pycache__
           |
           +-- ...
       |
       +-- test
           |
           +-- test.py
           |
           +-- ...
       |
       +-- EIDE.py
       +-- EIDEChannelsManager.py
       +-- ...
       +-- EIDEGraphics.py

```

La librería funciona desde cualquier sitio (carpeta: donde está, mismamente). Si quiere comprobar su funcionamiento vaya a la carpeta *test* y ejecute el fichero *test.py*; arrancará una *simulación* con el proyecto que esté definido en *EIDESystem.txt* (carpeta *EIDEGraphics*; ‘project = ...’).

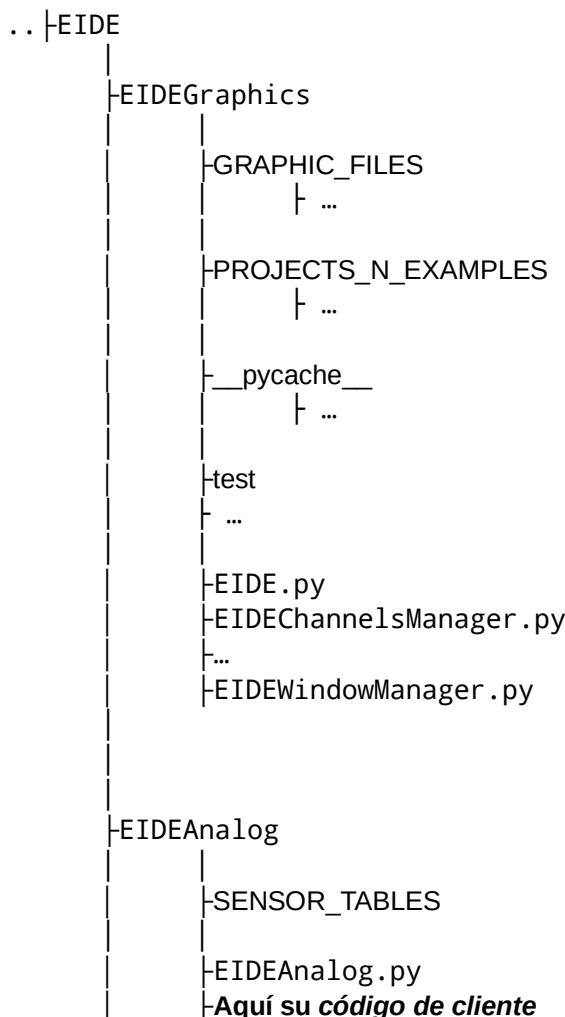


Python. Pygame: Para que funcione EIDEGraphics usted necesita tener instalado Python y *pygame* en Raspberry. No se alarme, ambos vienen “de fábrica” en cualquiera de las últimas distribuciones de Raspbian. Si tiene alguna duda quizás le convenga echar un vistazo al anexo II, “Configuración de Raspbian”.

La simulación es eso, una simulación, pero a efectos de valorar lo que puede (o no puede) hacer EIDEGraphics es real, toda vez que, aunque los valores que se representan son *fake*, no lo es la forma de representarlos, es decir, no se trata de un *video*, si no que EIDEGraphics está realmente visualizando en forma gráfica los valores que se le entregan en forma numérica; que sean unos valores reales o simulados es, se insiste, irrelevante a la hora de valorar cómo se ven. Tres de las cuatro visualizaciones de la Figura III.1 corresponden con “HEAT”, “480_320” y “AUDIO”. Cambie en *EIDESystem.txt* la línea *project* a cualquiera de estas y comprobará muchas de las posibilidades de EIDEGraphics. También puede probar cualquiera de las otras carpetas -proyectos- en *PROJECTS_N_EXAMPLES*.

III.2- Funcionamiento con EIDEAnalog.

Si quiere usted que los resultados -las conversiones- que vaya obteniendo con EIDEAnalog se muestren con EIDEGraphics necesita tener las librerías reorganizadas en una estructura de carpetas como la siguiente:



- Las carpetas EIDEAnalog y EIDEGraphics no tienen que llamarse así; tampoco la carpeta EIDE de la que cuelgan (aunque los nombres, dadas las circunstancias, parecen bastante razonables).
- Su código de cliente tiene que estar donde se indica (“Aquí su código de cliente”).

Para ilustrar lo anterior, estudiaremos la adaptación del ejemplo `EIDEAnalog_8200_EXERCISEb.py` (ver 8.2, “Uso con clases intermedias. Clase ‘ADCBus_manager’”) para visualizar los resultados con EIDEGraphics (el código resultante -que se muestra y analiza a continuación- en el fichero `EIDEAnalog_III.py`).

```
# Import EIDEAnalog.
```

```

import EIDEAnalog
# Import and instance EIDEGraphics
import facade as EGClient
EGUser = EGClient.EIDE.EIDEGraphics(1) # 1 sec. Interval

```

para simplificar el uso de EIDEGraphics se usa el módulo *facade.py*; se instancia la clase EIDEGraphics -EGUser = EGClient.EIDE.EIDEGraphics(1)- con un segundo de tiempo de actualización de los displays (en este caso; se puede elegir cualquier valor: 0,04 serían 25 FPS -frames per second-). El objeto así instanciado se llama *EGUser*.

```

instancedSensors = []
currentValues = [0, 0, 0]

# Instance buses and sensors. Add them to list.
myBus = EIDEAnalog.ADS1115(0x48)
NTC_china = EIDEAnalog.sensor('ADS1115_NTC', myBus, 1, name='NTC_placa')
T_Exterior = EIDEAnalog.sensor('LM35', myBus, 2, name='TE_LM35')
instancedSensors.append(NTC_china)
instancedSensors.append(T_Exterior)

myOtherBus = EIDEAnalog.oneWire(19)
DS18B20_1 = EIDEAnalog.sensor('DS18B20', myOtherBus, 1, name='ow1')
DS18B20_2 = EIDEAnalog.sensor('DS18B20', myOtherBus, 2, name='ow2')
instancedSensors.append(DS18B20_1)
instancedSensors.append(DS18B20_2)

while True:
    for counter, i in enumerate(instancedSensors):
        i.sensorBus.setChannelGain(i)
        i.sensorBus.singleShot()
        while not(i.sensorBus.ready()):
            pass
        currentValues[counter] = i.readout(i.sensorBus.readConversion())

```

se modifica el lazo de programa para que se ejecute continuamente (*while True*). Se añade el índice del lazo -*counter*- para depositar las lecturas en la lista:
(*currentValues[counter]* = *i.readout(i.sensorBus.readConversion())*)

```
EGUser.EIDEGLoop(currentValues)
```

Llamada al método *EIDEGLoop* de *EGUser* para que se muestren los valores.

Referencias

OOP

Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. (Libro).

OBJECT-ORIENTED ANALYSIS AND DESIGN With applications. SECOND EDITION. Grady Booch. Rational Santa Clara, California. (Libro).

Object-Oriented Software Construction. SECOND EDITION. Bertrand Meyer ISE Inc. Santa Barbara (California). (Libro).

https://en.wikipedia.org/wiki/Object-modeling_technique

Python

<https://docs.python.org/3/tutorial>

<https://wxpython.org/>

<https://numpy.org/>

Raspberry

www.raspberrypi.org

<https://thepihut.com/blogs/raspberry-pi-tutorials/an-introduction-to-raspberry-pi-gpio>

Raspbian. Bus 1-wire.

www.raspberrypi.org

<https://www.raspberrypi.org/downloads/raspbian/>

<https://raspberrytips.com/install-raspbian-raspberry-pi/>

<https://honeWirechoo.com/g/ywmxmza2ndf/raspbian-buster-install-or-upgrade>

Linux

The Linux Command Line: A Complete Introduction. William E. Shotts Jr. (Libro).

<https://www.linuxadictos.com/5-comandos-imprescindibles-en-linux>

<https://franciscomoya.gitbooks.io/taller-de-raspberry-pi/content/es/intro/gnu.html>

recursostic.educacion.es/observatorio/web/ca/software/software-general/295-jose-ignacio-lopez

ADS1115

www.ti.com/lit/ds/symlink/ads1115.pdf

<http://www.smartypies.com/projects/ads1115-with-raspberrypi-and-python/>

Arduino

<https://www.arduino.cc/>

Bus i2c

<https://learn.sparkfun.com/tutorials/raspberry-pi-spi-and-i2c-tutorial/all>

Sensores

<https://es.wikipedia.org/wiki/LM35>

<https://www.maximintegrated.com/en/products/sensors/DS1825.html>

<https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf>

<https://www.atlas-scientific.com>

Sears Zemansky. Física universitaria. (Libro).

Repositorios

<https://github.com/>

<https://www.raspberrypi.org/downloads>

<https://github.com/Vicente-Francisco/EIDEGraphics>

EIDEGraphics

<https://github.com/Vicente-Francisco/EIDEGraphics/README.md>