Trabajo Fin de Grado
Grado de Ingeniería de las Tecnologías de Telecomunicación

Diseño de una Interfaz Gráfica de Usuario para placa Arduino

Autor: Francisco Ramos Bratos
Tutora: Susana Hornillo Mellado

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Diseño de una Interfaz Gráfica de Usuario para placa Arduino

Autor:
Francisco Ramos Bratos

Tutora:
Susana Hornillo Mellado
Profesora Contratada Doctora

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo de Fin de Grado: Diseño de una Interfaz Gráfica de Usuario para placa Arduino

Autor: Francisco Ramos Bratos

Tutora: Susana Hornillo Mellado

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

Caminante, no hay camino,
se hace camino al andar.
-Antonio Machado-

Agradecimientos

He de agradecer a todos aquellos que han hecho posible la realización de este Trabajo de Fin de Grado. A mis padres quienes me aconsejaron y me ayudaron a superar los momentos en los que no supe cuál era la mejor forma de proceder. A Susana, mi profesora y tutora de este TFG, por ofrecer el tema y haberme facilitado su realización con el material necesario, sus consejos y correcciones. A Mattia por los consejos y respuestas sobre diversas dudas a la hora de programar. A Carlos, Marcos, Garik y Tallón quienes me acompañaron los fines de semana durante los años que ha durado este grado. Por último, a mis compañeros del grado que durante trabajos, prácticas y clases me aguantaron mis quejas.

Francisco Ramos Bratos

Sevilla, 2020

Resumen

El mundo de los microcontroladores puede ser algo complejo a primera vista por la variedad de fabricantes, tipos de procesadores de 8, 16 o 32 bits, lenguaje de programación usado C o C++… No obstante, hay empresas que han comenzado a considerar estas circunstancias y han desarrollado productos como Lego Mindstorm, Arduino y Raspberry Pi. Estos productos, que a simple vista no parecen poder clasificarse de forma convencional en estos aspectos, son una forma excelente de iniciar a personas sin conocimiento alguno en programación y electrónica.

Este Trabajo de Fin de Grado tratará sobre la creación de una interfaz gráfica de usuario para Arduino. El objetivo que se persigue no es controlar la lógica que regirá la placa desde el ordenador con el que se comunique sino facilitar la labor de aquellos usuarios que realicen sus proyectos con las placas de manera que sean capaces de visualizar aquellos datos que estimen convenientes de la forma más sencilla posible. Para ello, la interfaz permitirá la representación de múltiples gráficas, con ajuste automático de escalas, de aquellos datos procedentes de la placa Arduino que el usuario seleccione, así como los valores de estos datos y los de las variables seleccionadas lo que facilitará una comprensión del funcionamiento de una manera más intuitiva y rápida.

Por otro lado, dado que existe una gran cantidad de distintos modelos y con diferentes características, se hace necesario que el diseño del software sea sólido y expandible. Debido a esta circunstancia, se ha decidido usar Python para la creación de la GUI pues se trata de un lenguaje de alto nivel que a su vez resulta muy flexible. En cambio, se usará C++ para las librerías de las placas Arduino, justificándose este uso por razones más que evidentes de rendimiento.

Abstract

Microcontroller fields can be something at a first approach due to the variety of manufacturers, processor type of 8, 16 or 32 bits, the programming language being either C or C++. Nonetheless, there are several businesses who have started to consider this situation developing several products such as Lego Mindstorm, Arduino and Raspberry Pi. These products cannot be classified with by using traditional aspect. What make them special is that they are a great way to introduce people without any prior knowledge of programming and electronics.

This project will be focused on the development of a graphical user interface for Arduino boards. The aim isn't controlling the logic on the side of the computer which the board will be exchanging messages with, rather helping the work of those users who develop their projects by allowing to check the needed data in a graphical way. In order to achieve this goal, the interface will allow the representation of multiple graphs and two tables filled with data originated from the Arduino board.

Due to the amount of different board models with their own features, it's needed that the software is developed over a solid base. Due to this, it's has been decided to use Python to create the graphical user interface. On the other hand, C++ will be used to create the libraries for Arduino boards because the hardware is quite limited.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

El mundo de los microcontroladores es extenso y diverso debido a los distintos tipos y funcionalidades de los mismos. El rango de aplicaciones es bastante diverso, desde los usos industriales hasta los médicos. Los fabricantes de microcontroladores más conocidos, aunque no los únicos, son Microchip, Texas Instruments, Intel y Dallas Semiconductor.

En los últimos años se ha popularizado Arduino, empresa surgida del desarrollo de un proyecto universitario que no ofrece un microcontrolador propio, sino una placa de prototipado, así como un entorno de desarrollo.

Con este enfoque, se facilita que esta placa sea un punto de iniciación ideal para aquellos que comienzan a aprender sobre electrónica y programación. Pensando en este tipo de usuarios, sería de gran ayuda para los mismos la posibilidad de disponer de una interfaz gráfica de usuario que facilite la obtención, presentación e interpretación de los datos obtenidos. La interfaz gráfica resultaría muy útil también para usuarios más avanzados, pues permite ver con facilidad ciertos datos y, por tanto, hacerse con rapidez una idea del funcionamiento del dispositivo que se analiza.

1.1 Objetivos

El objetivo de este proyecto es hacer una GUI (*Graphical User Interface*) para las placas de Arduino con las siguientes características:

- Diseño amigable enfocado a personas que carecen de conocimientos avanzados.
- Uso intuitivo y sencillo de la interfaz.
- Posibilidad de visualización de múltiples gráficas que representen los valores obtenidos en los diferentes PIN de la placa a elección del usuario.
- Autoescalado automatizado de las gráficas en ambos ejes para un enfoque adecuado de las representaciones gráficas de los datos.
- Presentación simultánea de los valores de los datos elegidos por el usuario, así como de algunas variables internas del microcontrolador.
- Posibilidad de ver los mensajes de comunicación entre el ordenador y la placa.

La información que se pone a disposición del usuario mediante esta GUI facilitará al mismo la toma de decisiones que dependan de los datos así obtenidos pudiendo aplicarse al control de dispositivos, domótica, etcétera.

2 ESTADO DE LA CUESTIÓN

La importancia de presentar el contexto técnico es vital para la comprensión de este documento. Lo que se busca con este apartado es presentar al lector con las diferentes alternativas actuales a la hora de desarrollar interfaces gráficas y las pautas a seguir. También las distintas tecnologías posibles a usar para realizar desarrollos de interfaces gráficas.

Las interfaces gráficas surgieron motivadas por las dificultades que presentaban para los usuarios las que estaban regidas por comandos, mucho más áridas y que requerían un mayor esfuerzo de aprendizaje y manejo.

Una GUI permite al usuario interactuar mediante ventanas, iconos, gráficos e indicadores visuales, además del propio texto para representar la información y las acciones disponibles para el usuario. El uso de ventanas, iconos, menús y punteros (WIMP en inglés) permiten la interactividad de forma mucho más intuitiva y sencilla.[1]

2.1 Pautas en el diseño de interfaces gráficas

La interacción humano-ordenador, en adelante HCI (*Human-Computer Interaction*), es un tema que, si bien en una primera aproximación puede parecer algo sencillo, cuando se profundiza pasa a ser bastante complejo. La HCI se ocupa del estudio de la creación de productos informáticos que ayuden en la realización de tareas a sus usuarios atendiendo a la facilidad de uso, al tiempo de ejecución, evitar posibles errores y, en consecuencia, a su satisfacción.[2]

La HCI puede darse mediante muchos tipos de interfaces, destacan entre ellos las interfaces mediante comandos, de forma gráfica, multimedia, web, móviles, táctil, voz, realidad aumentada y realidad virtual. No necesariamente son excluyentes entre ellas y puede darse el caso del uso simultáneo de una línea de comandos en una GUI. A modo de ejemplo, se puede observar en la figura 2-1, el IDE Pycharm de la compañía JetBrains, una GUI para el desarrollo en Python que permite utilizar comandos al mismo tiempo, como se puede apreciar en la zona inferior izquierda de la imagen.

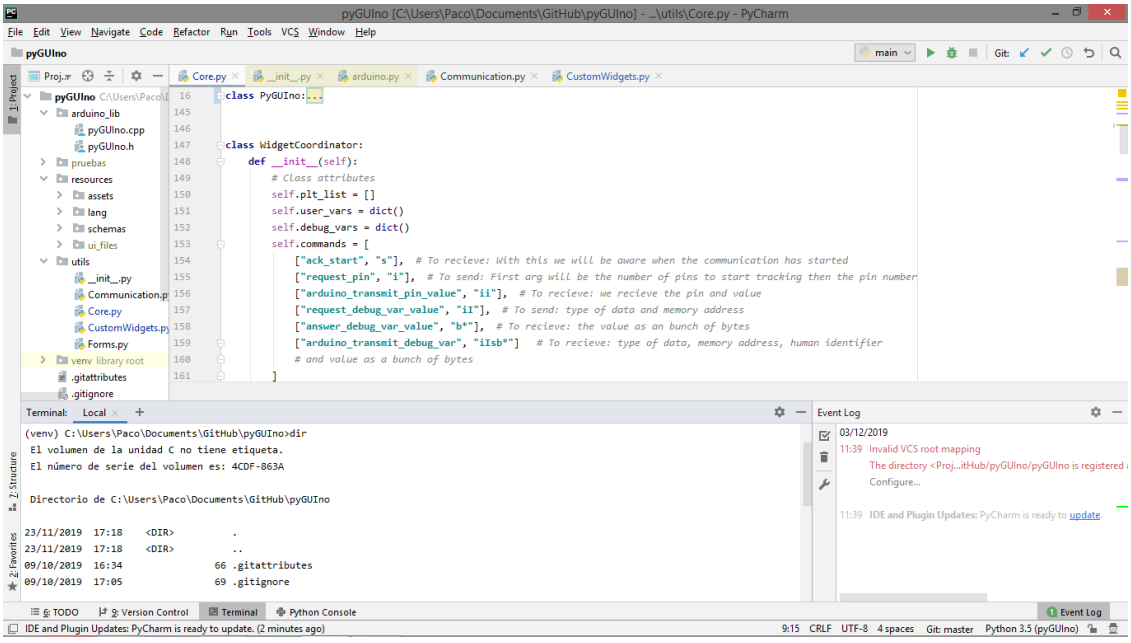


Figura 2-1 PyCharm siendo simultáneamente una GUI y permitiendo al usuario el uso de comandos

En el diseño de una GUI debemos tener en cuenta que su finalidad es presentar grandes cantidades de información y funciones sobre un elemento de tipo visual y hacerlo de una manera clara y eficiente.

Para ello, se ha pensado que el diseño se haga partiendo del concepto de escritorio, con un sistema con ventanas que dividen la pantalla en áreas funcionales y donde se pueda interaccionar con diferentes elementos y gráficos. Para ello se prestará atención a los siguientes aspectos:

- Diseño de la pantalla, las relaciones y la navegabilidad.
- Efectividad.
- Adecuar la presentación de los elementos a las características del tipo de usuario al que se dirige.

Actualmente, existen diversas guías y principios enfocados a orientar la forma de realizar interfaces de usuario. En los últimos años Google ha desarrollado "*Material Design*", un conjunto de pautas para el desarrollo de aplicaciones web y móviles que persigue unificar el diseño de las aplicaciones [3], ofreciendo para ello ayudas y guías a los desarrolladores. La documentación de *"Material Design"* tiene un alto grado de detalle para desarrollar aplicaciones que, si bien en una primera aproximación puede facilitar el desarrollo, a su vez convierte esta opción en una norma difícil de seguir.

El enfoque que tiene "*Material Design*" no resulta conveniente para nuestra aplicación [4], por lo que se ha optado por seguir un conjunto de pautas más tradicionales como propone Jeff Johnson:

1. Centrarse en los usuarios y sus tareas, no en la tecnología.
2. Considerar la funcionalidad primero, después la presentación.
3. Adecuar a la visión de la tarea que tiene el usuario.
4. Diseñar para los casos de usos comunes.
5. No complicar la tarea del usuario.
6. Facilitar el aprendizaje.
7. Entregar la información, no solo datos.
8. Diseño responsivo.
9. Probarlo en los usuarios, ¡después arreglarlo!

2.2 Placas de prototipado en el mercado

Existe una gran variedad de placas de prototipado en el mercado cada una de las cuales cuenta con diversas características. Es tan amplio el rango de selección que, a modo de ejemplo, sólo se mencionarán algunas de las opciones más populares:

- Thunderboard Sense 2 IoT Development Kit SLTB004A
- Adafruit Feather M0 Bluefruit LE Board
- Arduino Uno Rev3
- TM4C123G LaunchPad Evaluation Kit

Un *framework* es un conjunto de librerías y tecnologías diversas que tienen como objetivo facilitar la escritura de código a los desarrolladores. En los casos de las placas anteriormente citadas, la implementación del *framework Wiring* es una característica que tienen en común algunas de ellas y que no está relacionada con el hardware.

Este *framework* de código abierto para microcontroladores permite su control sin necesidad de trabajar a un bajo nivel, es decir, tratar con registros y direcciones de memorias específicas, lo que supone una mayor facilidad para el programador. [5]

Por otro lado, este *framework* basa su proceso de desarrollo en *"sketches"* con el hardware, esto se podría definir como escribir programas de breve extensión utilizando algunos componentes electrónicos permitiendo así la introducción en un periodo de corta duración en una gran cantidad de conceptos, seleccionar aquellas más interesantes, refinarlas y producir prototipos de forma interactiva.

Adicionalmente Arduino AG, empresa propietaria de las placas de prototipado Arduino, ofrece un entorno de desarrollo integrado (IDE) mediante el cual los usuarios no deben preocuparse de tareas como la compilación y grabado de los sketches en las placas. A esta circunstancia, se añade la ventaja que supone el bajo coste de estas placas.

De los distintos modelos de placas de prototipado Arduino, en la tabla 2-1 se pueden observar algunas de las que la compañía ofrece en la actualidad [6]. La tabla permite observar las características de los distintos modelos y su comparación. Podemos observar que:

- Los modelos Uno, Leonardo, Micro y Mega portan variantes de microcontroladores de la serie ATmega, con arquitectura AVR de 8 bits, existiendo mínima diferencia en sus características, a excepción del Arduino Mega cuya memoria flash, EEPROM y SRAM es mayor respecto a los otros modelos. [7]
- Los modelos Due y MKR Zero usan microcontroladores con arquitectura ARM de 32 bits[8], [9], de diseño mucho más reciente. Por este motivo, las prestaciones son mejores en comparación con los modelos que utilizan los microcontroladores con arquitectura AVR.

Tabla 2-1. Características de los distintos modelos de Arduino

Modelo	Frecuencia	Microcontrolador	Memoria SRAM total	Memoria EEPROM	Memoria Flash
Arduino Uno Rev3	16MHz	ATmega328P	2KB	1KB	32KB
Arduino Leonardo	16MHz	ATmega32u4	2.5KB	1KB	32KB
Arduino Micro	16MHz	ATmega32u4	2.5KB	1KB	32KB
Arduino Mega 2560 Rev3	16MHz	ATmega2560	8KB	4KB	256KB
Arduino Due	84MHz	AT91SAM3X8E	96KB	No	512KB
Arduino MKR Zero	48MHz	SAMD21 Cortex-M0 + 32bit low power ARM MCU	32KB	No	256KB

Tras analizar las características, se ha optado por el modelo Arduino Uno para el desarrollo de este proyecto porque se adecúa mejor a los objetivos y las características del presente Trabajo de Fin de Grado.

2.3 Contexto para la elección del software adecuado

2.3.1 Lenguaje de programación

Con el fin de realizar la interfaz de usuario se debe escoger un lenguaje de programación. Analizadas las opciones disponibles, nos encontramos con que las principales son C++, Java y Python. Estas tres opciones tienen características que se analizan para la elección adecuada del más apropiado para los propósitos que se persiguen en este Trabajo de Fin de Grado.

- Todos ellos soportan objetos, herencia, polimorfismo y otras características propias de lenguajes de alto nivel.
- Tanto Java como Python son lenguajes interpretados, C++ por el contrario es un lenguaje compilado. No obstante, Java tiene una compilación en dos pasos, la primera fase pasa el código fuente a *bytecode*, que es el conjunto de instrucciones que procesa la máquina virtual de Java, tras ello, en la segunda fase el *bytecode* sea compilado para su uso por el microprocesador que ejecuta la máquina virtual.

- Si comparamos el rendimiento, normalmente el código creado en C++ será mucho más eficiente que aquel escrito en Java o Python.
- C++ no tiene integrados tipos de datos de alto nivel, tampoco operaciones primitivas de alto nivel [10].
- C++ requiere que el programador gestione manualmente la memoria, su asignación y posteriormente su liberación.
- C++ requiere especificar los tipos de datos en las operaciones que se realizan, aunque esto puede ser mitigado con el uso de plantillas.
- Java, es un lenguaje de tipos estáticos de datos, pues comprueba los tipos de datos en el momento de la compilación. Esto permite un mayor rendimiento frente a otros lenguajes como Python o JavaScript.
- La máquina virtual de Java ha sido portada a multitud de plataformas.
- Python permite un rápido desarrollo gracias a la flexibilidad de su sintaxis, así como la multitud de módulos que posee.
- Python cuenta con revisiones mucho más frecuentes si lo comparamos con los otros dos lenguajes que se han presentado previamente.
- Python tiene dos versiones principales:
    - Python 2.7: aunque a esta versión le será retirado el soporte en 2020, recomendando Python Software Foundation el uso de versiones más recientes, se tiene en cuenta para facilitar la compatibilidad con código que no será portado a la siguiente versión puesto que aún existe cierta cantidad considerable de programadores desarrollando para esta versión.
    - Python 3: esta versión recibe nuevas actualizaciones, destacando la versión 3.5 que tiene la característica de añadir la posibilidad de realizar tareas de forma asíncrona mediante el uso del módulo *asyncio.* Además, parte de la sintaxis varía en algunos casos con respecto a la versión 2.7.

Por todo lo anteriormente expuesto, para la implementación se ha optado por descartar C++ para el desarrollo de la GUI debido a que es un lenguaje de bajo nivel y la gestión que debe hacerse de la memoria de forma manual. Tampoco se ha optado por elegir Java, debido a la redundancia de su sintaxis a la hora de escribir código en este lenguaje.

Así pues, la mejor elección es Python por la flexibilidad de su sintaxis y la diversidad de módulos que posee, así como la facilidad en la instalación de los mismos.

2.3.2 Módulos para la creación de interfaces gráficas

Habiendo optado por el lenguaje de programación Python, uno de los puntos en los que tiene más robustez es en la diversidad de módulos que los desarrolladores tienen a su disposición para realizar diversas tareas. Esto se debe a que Python tiene integrado un conjunto de herramientas que permiten la comunicación del código creado en este lenguaje con C y C++[11]. En algunas ocasiones es necesario recurrir al uso de *bindings,* son interfaces diseñadas para permitir que un conjunto de funciones y servicios desarrollados para un lenguaje de programación especifico puedan ser utilizados por otro lenguaje distinto.

Para el desarrollo de interfaces gráficas destacan los siguientes módulos:

- PyQt5: es un *binding* de Qt creado por RiverBank Computing.
- Pyside2: otro *binding* de Qt, creado por The Qt Company.
- Kivy: módulo que busca ser eficiente, flexible, escrito desde cero. Gracias a esto último en el desarrollo se pudieron volver a pensar muchas ideas en lo referente a la HCI. [12]
- TkInter: módulo por defecto incluido en toda instalación de Python.

Qt es un *framework* desarrollado en C++, para el desarrollo de interfaces gráficas en ordenadores, sistemas embebidos y móviles. Soporta plataformas tales como Linux, OS X, Windows, Android, IOS y BlackBerry entre otros muchos. [13]

Tanto PyQt5 como PySide2 son *bindings* de este *framework* para obtener sus funciones en Python, apenas hay diferencia en la creación del código.

De las opciones disponibles, se ha escogido PyQt5 debido a que tiene implementadas ciertas funciones que el *binding* PySide2 no ofrece porque parte de su implementación no tiene un correcto funcionamiento.

2.3.3 Módulos para comunicación con Arduino

Las placas de prototipado tienen diversas formas de comunicarse con otros dispositivos. Las opciones más comunes que suele escoger la comunidad de Arduino a la hora de desarrollar sus proyectos son conexiones de tipo serie, Bluetooth o IP (mediante WiFi o Ethernet).

El presente trabajo quiere dar soporte a las conexiones serie y bluetooth, no contemplando dar soporte a conexiones sobre IP, por lo tanto, teniendo en cuenta que Python da soporte a conexiones serie y bluetooth mediante los módulos *pyserial* y *pyBluez*, respectivamente, será con dichos módulos con los que se desarrollará la comunicación con la placa Arduino.

Una vez conseguida la comunicación a través de los módulos anteriores, se plantea la lógica del correcto funcionamiento de dicha comunicación entre el ordenador y la placa Arduino. Es posible resolver dicho problema con la librería *CmdMessenger*; esta librería para C++ y su implementación para Python, *PyCmdMessenger*, permitirá identificar distintos tipos de mensajes y sus correspondientes argumentos entre los dos extremos.

2.3.4 Módulos para la representación gráfica de datos

Uno de los objetivos de este Trabajo de Fin de Grado es cubrir la necesidad de mostrar al usuario parte de la información obtenida de Arduino mediante gráficas, por lo que se requiere de un módulo para realizar esta tarea. Las principales alternativas que se ofrecen en Python son los módulos *matplotlib* [14] y *pyqthgraph* [15].

El módulo *pyqtgprah* no es tan completo como *matplotlib*, pero funciona mucho más rápido. El módulo *matplotlib* está más enfocado a la creación de gráficos de alta calidad para publicaciones, mientras que *pyqtgraph* está pensado para su uso en la captura de datos y aplicaciones de análisis. *Matplotlib* es más intuitivo para los programadores de Matlab, mientras que pyqtgraph es más intuitivo para los programadores de Python y/o Qt.

Según Luke Campagnola, desarrollador de *pyqtgraph*, *matplotlib* no incluye muchas de las características propias de *pyqtprah* como interacción con imágenes, renderizado volumétrico, diagramas de flujo...[16]

La diferencia en la eficiencia de ambos módulos en lo referente al renderizado de *frames,* o imágenes, por segundo es notable y puede comprobarse en el siguiente ejemplo donde se ejecuta en la misma máquina un código con el módulo *matplotlib* y el mismo ejemplo adaptado al módulo *pyqtgraph*. El resultado es muy clarificador, pues en *matplotlib* se obtienen unos 18 *frames* por segundo y en *pyqtgraph* unos 250 *frames* por segundo.[17]

Código ejecutado con matplotlib

```
import time
from matplotlib import pyplot as plt
import numpy as np


def live_update_demo():
    x = np.linspace(0, 50., num=100)
    fig = plt.figure()
    ax1 = fig.add_subplot(2, 1, 1)
    fig.canvas.draw()

    h1, = ax1.plot(x, lw=3)
    ax1.set_ylim([-1, 1])

    t_start = time.time()
```

```
    plt.pause(0.000000000001)
    k = 0.
    for i in np.arange(1000):
        h1.set_ydata(np.sin(x / 3. + k))
        # print tx
        k += 0.11

        fig.canvas.draw()
        fig.canvas.flush_events()

    t_end = time.time()
    fps_avg = 1000/(t_end-t_start)
    print('fps average: {}'.format(fps_avg))


live_update_demo()
```

Código ejecutado con pyqtgraph

```
import sys
import time
from pyqtgraph.Qt import QtCore, QtGui
import numpy as np
import pyqtgraph as pg


class App(QtGui.QMainWindow):
    def __init__(self, parent=None):
        super(App, self).__init__(parent)

        self.mainbox = QtGui.QWidget()
        self.setCentralWidget(self.mainbox)
        self.mainbox.setLayout(QtGui.QVBoxLayout())

        self.canvas = pg.GraphicsLayoutWidget()
        self.mainbox.layout().addWidget(self.canvas)

        self.label = QtGui.QLabel()
        self.mainbox.layout().addWidget(self.label)

        self.otherplot = self.canvas.addPlot()
        self.h2 = self.otherplot.plot(pen='y')

        self.x = np.linspace(0, 50., num=100)

        self.counter = 0
        self.fps = 0.
        self.lastupdate = time.time()

        self._update()

    def _update(self):

        self.ydata = np.sin(self.x/3. + self.counter/9.)
        self.h2.setData(self.ydata)

        now = time.time()
        dt = (now-self.lastupdate)
        if dt <= 0:
            dt = 0.000000000001
        fps2 = 1.0 / dt
        self.lastupdate = now
        self.fps = self.fps * 0.9 + fps2 * 0.1
        tx = 'Mean Frame Rate:  {fps:.3f} FPS'.format(fps=self.fps )
        self.label.setText(tx)
        QtCore.QTimer.singleShot(1, self._update)
```

```
        self.counter += 1


if __name__ == '__main__':
    app = QtGui.QApplication(sys.argv)
    thisapp = App()
    thisapp.show()
    sys.exit(app.exec_())
```

Debido a la naturaleza de este Trabajo de Fin de Grado resulta más apropiado el uso del módulo *pyqtgraph*. Además, como se ha dicho anteriormente, resulta mucho más eficiente.

3 DISEÑO DE LA INTERFAZ

Para cumplir los objetivos planteados para el diseño de la interfaz gráfica elegimos un modelo basado en un diseño enfocado a personas sin conocimientos avanzados, con un uso sencillo e intuitivo. Por otro lado, se busca la funcionalidad específica de captura y visualización de datos de la placa Arduino tanto en modo gráfico, incluyendo la posibilidad de múltiples gráficas como los valores de dichos datos. Además, se ofrecerá al usuario los mensajes de comunicación entre ordenador y placa de Arduino.

Durante el diseño de la interfaz se distinguirán tres partes diferenciadas. La primera de ellas es la definición de la ventana principal, que engloba todos los datos que el usuario podrá visualizar. La segunda etapa, es el diseño de los diálogos que el usuario deberá usar para interactuar con el programa. Finalmente, se hablará sobre cómo son mostrados los errores al usuario.

3.1 Ventana principal

La ventana principal es elemento básico con el que el usuario accederá a la aplicación. Por tanto, la distribución, funcionalidad y uso de sus elementos debe ser intuitiva, de forma que el usuario pueda acceder a las funciones deseadas sin excesiva complejidad, así que se ha decido hacer separación de la ventana principal en dos partes:

1. Barra de herramientas: permitirá al usuario guardar, cargar, añadir elementos a la zona de trabajo e iniciar o finalizar la comunicación con la placa y seleccionar el tipo de placa Arduino.
2. Zona de trabajo: será la parte de la interfaz de usuario que contendrá los datos recuperados de la placa Arduino.
3. Barra de estado: en la parte inferior de la ventana se ha añadido barra de estado, en ella se muestran pequeñas descripciones sobre algunas acciones contenidas en la barra de herramientas.

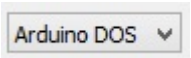
3.1.1 Barra de herramientas

La barra herramientas consta de 6 botones y cuadro de selección. La barra de herramientas permite ser ubicada a elección del usuario mediante su selección, desplazamiento y fijación de la posición final con el botón izquierdo del ratón.

Figura 3-1 Barra de herramientas

Por otro lado, en la tabla siguiente se describe la funcionalidad de cada uno de los botones de la barra de herramientas para los que se han seleccionado imágenes que recuerden dicha funcionalidad que proceden de *Google Material Design*, donde están disponibles bajo licencia Apache versión 2.0.

Tabla 3-1. Funcionalidad de cada botón de la barra de herramientas

Icono	Funcionalidad
	Guarda la configuración de las gráficas que el usuario haya definido previamente para cada pestaña en un archivo .json
	Carga la configuración de las gráficas almacenada en un archivo .json
	Lanza el cuadro de diálogo que permite al usuario configurar los parámetros de visualización de las gráficas.
	Abre el cuadro de diálogo para configurar los parámetros de la conexión con la placa.
	Inicia la recepción de mensajes y datos desde la placa.
	Detiene la recepción de mensajes y datos desde la placa.
	Permite seleccionar el modelo de placa de Arduino para adecuar la correcta interpretación de la longitud de los datos en bytes. Las opciones son modelos citados en la tabla 2-1

3.1.2 Zona de trabajo

La zona de trabajo se ha diseñado mediante una composición de 4 elementos, dispuestos como aparecen en la figura 3-2, que representa el diseño inicial o *mock-up* de la interfaz.

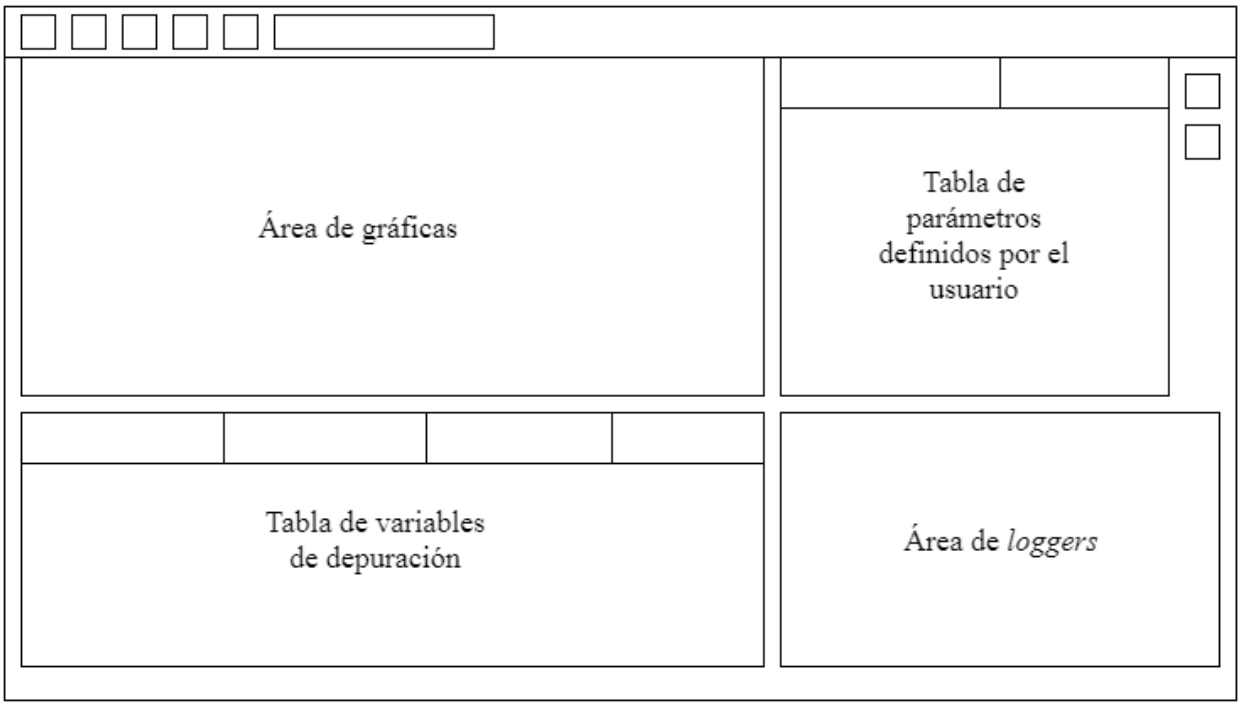


Figura 3-2 Esquema de diseño básico de la interfaz

Los cuatro elementos de la zona de trabajo son: el área de gráficas, tabla de variables de usuario, tabla de variables de depuración y área de *loggers,* que son aquellas partes del software que tienen designado registrar mensajes de distinto tipo sobre el proceso de ejecución. Cada área de la zona de trabajo es de tamaño variable,

pudiendo el usuario ajustar el tamaño de cada zona desplazando el separador vertical u horizontal entre ellas.

A continuación, describimos cada uno de estos cuatro elementos:

- Área de gráficas: situada en la parte superior izquierda. Permite al usuario obtener una representación gráfica de los datos recuperados de Arduino. El usuario puede crear múltiples pestañas en cada una de las cuales podrá generarse una gráfica que a su vez podrá contener varias curvas. Las curvas podrán representar los datos originales que proceden de la placa o bien los datos que resulten de una transformación de los originales mediante una función matemática que habrá sido predefinida por el usuario. Las curvas representadas podrán ser configuradas por el usuario para que tengan diversos colores. Las escalas para la correcta representación en la ventana de las curvas son ajustadas automáticamente por la aplicación, es decir, las gráficas son autoescalables.
- Tabla de parámetros definidos por el usuario: situada en la parte superior derecha, en esta tabla el usuario puede definir parámetros propios, que se pueden usar en las fórmulas matemáticas que se apliquen para la representación de las curvas. En esta zona, además, se encuentran dos botones que permiten la adición de parámetros a la tabla, así como su eliminación.
- Tabla de variables de depuración: esta tabla muestra información de variables de Arduino que el usuario indicó en el s*ketch*. Para cada variable que el usuario haya indicado, se muestran el identificador que el usuario le dio en el s*ketch*, su valor, su dirección de memoria y el tipo de dato almacenado en la memoria de Arduino. Dichas variables, permiten también su uso en las fórmulas matemáticas del área de gráficas.
- Área de *loggers*: provee al usuario de la información sobre los mensajes que la placa Arduino recibe de sus posibles periféricos, así como aquellos mensajes referentes a la comunicación entre la placa Arduino y el ordenador. Existen 3 pestañas que actúan como filtros para los mensajes, de forma que estos se muestran en la pestaña que corresponda:

Tabla 3-2. Pestañas del área de *loggers* y filtros aplicados a los mensajes

Pestaña	Filtro aplicado a los mensajes
Todos	No se aplica ningún filtro. Por tanto, muestra todos los mensajes.
I2C	Selecciona para mostrar los mensajes procedentes de la comunicación I2C.
SPI	Selecciona para mostrar los mensajes procedentes de la comunicación SPI.

3.2 Cuadros de diálogo

Los distintos cuadros de diálogo que se han previsto en el programa provienen en su mayoría de las funcionalidades asignadas a los botones de la barra de herramientas, no obstante, también aparece un cuadro de diálogo al pulsar el botón “Añadir” del área correspondiente a la tabla de parámetros definidos por el usuario.

- Guardar: el usuario guarda las configuraciones para gráficas, así como los parámetros que haya definido a un archivo. Se puede apreciar en la figura 3-3.

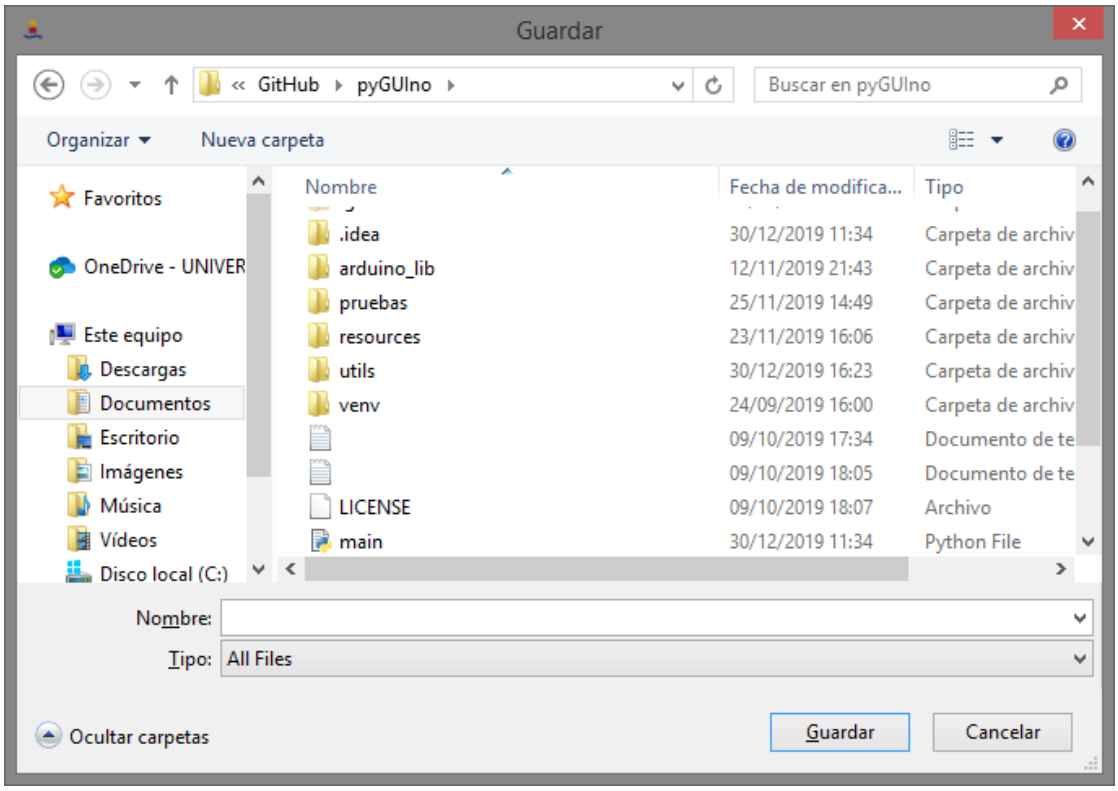


Figura 3-3 Diálogo de guardar

- Cargar: permite al usuario seleccionar un fichero el cual será leído por el programa y cargará las opciones de las curvas previamente guardadas y parámetros definidos por el usuario. Se aprecia en la figura 3-4.

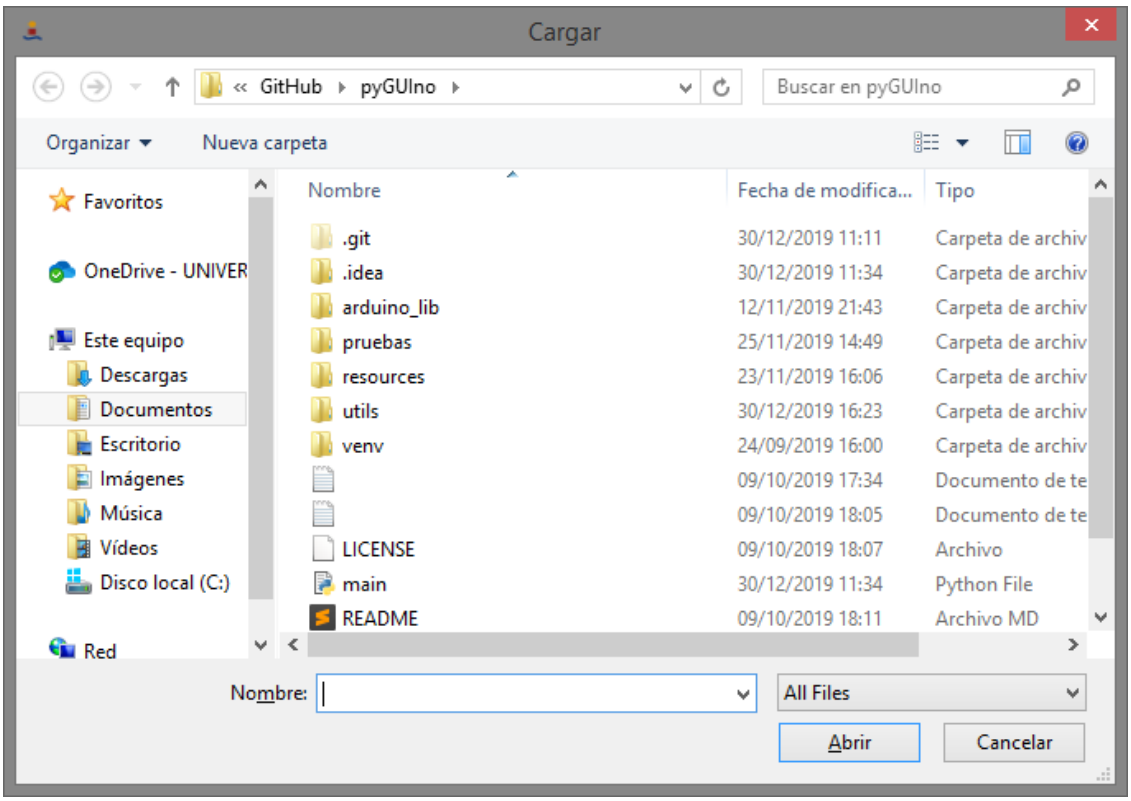


Figura 3-4 Diálogo de cargar

- Añadir pestaña gráfica: el usuario añadirá una gráfica al área de graficas mediante este diálogo (figura 3-5). La gráfica que añadirá podrá contener múltiples curvas. Para añadir una curva a la gráfica que se está creando se debe acceder mediante el botón "Añadir" del cuadro de diálogo representado en la figura 3-5, lo que hará que se despliegue un nuevo cuadro de diálogo para configurar los parámetros de la curva que se va a incluir en la gráfica creada (figura 3-6), selección del pin o variable con que se actualizará la curva correspondiente, color de esta y expresión matemática.

  Los datos de configuración para cada gráfica deben añadirse a una tabla contenida en el primer diálogo con el fin de que el usuario sea capaz de ver la configuración escogida para las curvas. Debido a que existe la posibilidad de que se desee cambiar o eliminar parte de las configuraciones para las curvas debe darse la posibilidad de eliminar las entradas de la tabla contenida en el diálogo mediante un botón, para lo cual se ha previsto el botón "Suprimir" en el cuadro de diálogo de la figura 3-5.

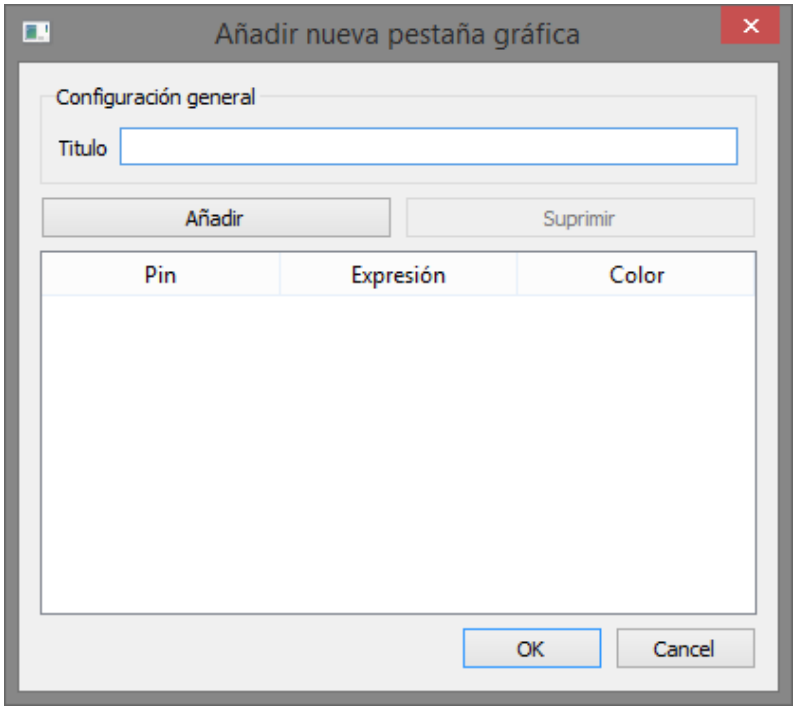


Figura 3-5 Diálogo para añadir una pestaña al área de gráficas

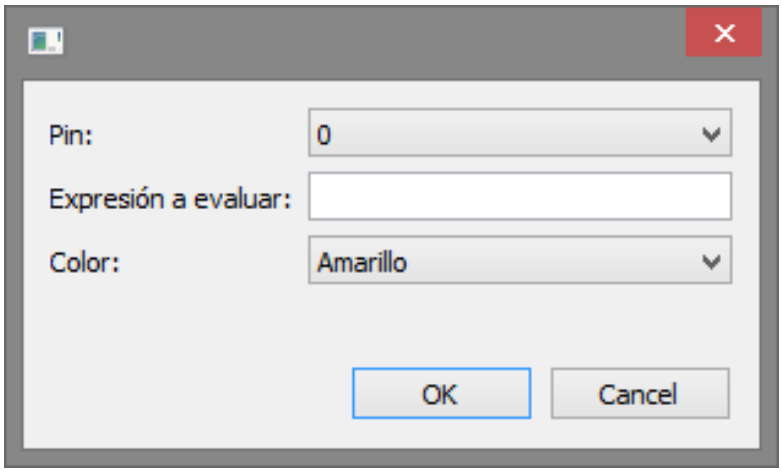


Figura 3-6 Diálogo para escoger los parámetros de las curvas

- Configuración de la conexión: este cuadro de diálogo permite que el usuario escoja el tipo de conexión que usará para comunicarse con Arduino y la configuración de los parámetros necesarios para esta. Por ello el cuadro de diálogo contendrá dos casillas de selección que permitirán escoger entre las conexiones soportadas que son serie y Bluetooth (figuras 3-7 y 3-8). Dependiendo de la opción seleccionada el cuadro de diálogo cambia para adaptarse a los parámetros de conexión necesarios, en el caso de conexión serie son el puerto y la tasa de baudios, en el caso de conexión Bluetooth se trata de un cuadro de selección mostrando los dispositivos disponibles para enlazar.

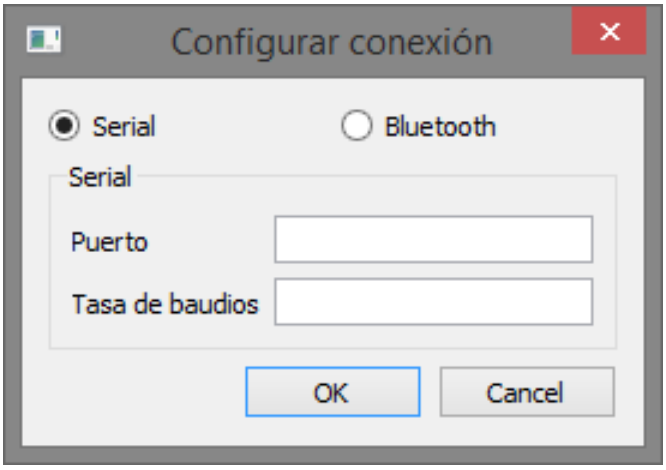


Figura 3-7 Diálogo para configurar la conexión, serie seleccionado

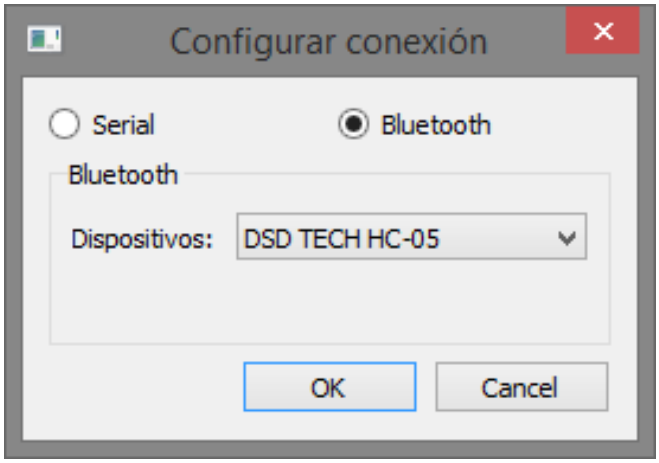


Figura 3-8 Diálogo para configurar la conexión, Bluetooth seleccionado

- Añadir parámetro: en este cuadro de diálogo, el usuario introduce el nombre y valor del parámetro que va a añadir y que podrá ser utilizado en las expresiones matemáticas de las curvas que serán representadas en el área de gráficas. La figura 3-9 muestra cómo es el cuadro de diálogo.

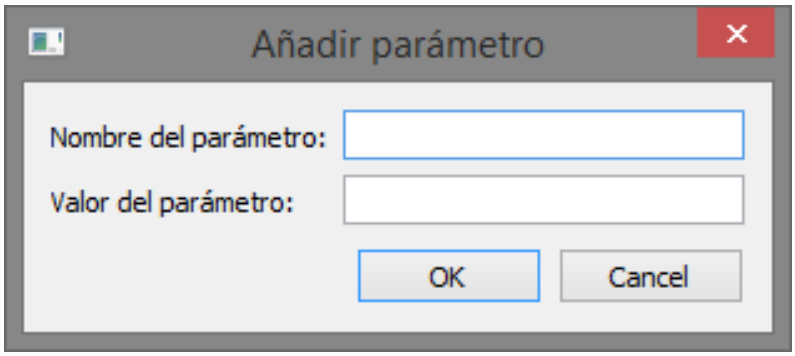


Figura 3-9 Diálogo para añadir parámetros

3.3 Ventana de errores

Los errores que se pueden producir son debidos a diversas circunstancias, por ello se ha previsto la presentación del tipo y motivo del error mediante una ventana como la de la figura 3-10, cuyo título informa del tipo de error producido y cuyo mensaje aclara el motivo por el que se ha producido. También se permitirá al usuario evitar que el error vuelva a mostrarse en el futuro en caso de que ocurriera de nuevo.

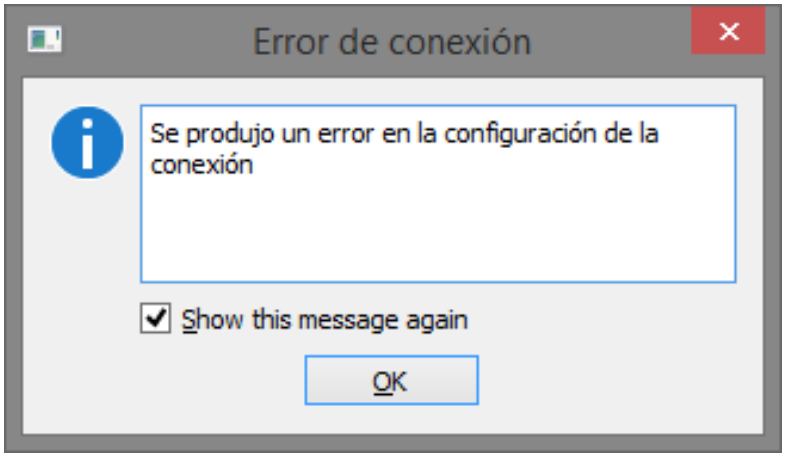


Figura 3-10 Ventana de errores

4 DISEÑO DE LA SOLUCIÓN SOFTWARE

En el capítulo previo hemos definido los requisitos de la interfaz de usuario, en este abordaremos el problema técnico con el que nos enfrentamos, los módulos necesarios y una descripción de las clases que aprovechen las funcionalidades que nos permitan resolver el problema.

4.1 Análisis del problema

Para llevar a cabo el objetivo del proyecto debemos descomponer el problema en partes, la solución debe cumplir los requisitos para la GUI establecidos en el capítulo anterior, establecer un conjunto de mensajes para la comunicación entre Arduino y el ordenador, la forma en la que deben ser tratados estos mensajes y, al mismo tiempo, lidiar con algunas restricciones del módulo utilizado para la creación de la GUI.

4.1.1 Comunicación entre el ordenador y Arduino

El ordenador debe comunicarse con Arduino. Para realizar esto es necesario establecer una conexión física y una lógica para que Arduino comprenda las peticiones y las atienda, además se debe asegurar la interpretación de los datos recibidos correctamente. La comunicación física se ha realizado a través de conexión serie o conexión bluetooth.

La librería *CmdMessenger*, desarrollada en C++, junto con su correspondiente implementación para Python, *PyCmdMessenger*, ofrecen una solución a la conexión a nivel lógico y también a nivel físico, aunque con ciertas limitaciones en esta última. La forma en la que trabaja esta herramienta es mediante el uso de un conjunto de mensajes, tanto en el código correspondiente a Arduino como en el ordenador. En Arduino, el tratamiento de estos mensajes se realiza asignándole a cada tipo de mensaje una función. La idea en el lado del ordenador es parecida, ya que se obtiene el mensaje junto con sus argumentos, pero no se obliga a que sea una misma función la que trate el mensaje siempre, permitiendo mayor flexibilidad.

Los mensajes que definiremos para el programa son los siguientes:

- Aviso de comienzo: enviado por Arduino al ordenador indicando que ha comenzado la conexión con el ordenador.
- Transmisión del valor de un pin: Arduino transmite el número con el que se identifica el pin y su valor correspondiente.
- Envío de variable de depuración: Arduino envía los datos correspondientes a una variable de depuración, su identificador, tipo de dato, valor y dirección de memoria.

Internamente, el módulo *PyCmdMessenger* trabaja basándose en la separación de las siguientes tareas:

- Establecer y mantener la conexión, los parámetros de esta, el envío y recepción de datos a través del enlace físico y conservar los valores propios del modelo de placa de Arduino. Esta tarea es abordada por la clase *ArduinoBoard*.
- La transformación de la información recibida desde *ArduinoBoard* (en bytes) hacia el usuario del módulo (como mensajes y sus argumentos) y viceversa es una tarea de la que se encarga la clase *CmdMessenger*[1].

[1] No confundir CmdMessenger de PyCmdMesseger, que es una clase, con la librería propia para C++ cuyo nombre es el mismo.

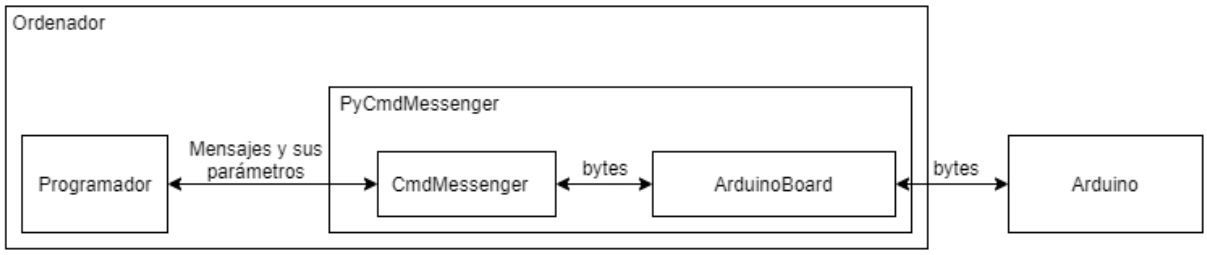


Figura 4-1 Diagrama de flujo de información de *PyCmdMessenger*

Mike Harms[18], desarrollador de este módulo, creó el mismo únicamente con soporte para conexiones de tipo serie, pero gracias a la estructura del código original es posible crear clases que simulen el comportamiento de la clase *ArduinoBoard*. Esto permite extender la conectividad del módulo *PyCmdMessenger* para Bluetooth o conexiones sobre IP.

En el apartado 4.2.4 se describe la implementación de la clase que permite la conexión Bluetooth realizada por el autor de este trabajo de fin de grado, cumpliendo con uno de los objetivos del mismo.

4.1.2 Qt, bucle de eventos y bloqueos

Qt no es un lenguaje de programación en sí. Es un *framework* escrito en C++. Utiliza un preprocesador llamado MOC (*Meta Object Compiler*), que es usado para extender C++ con funcionalidades como señales y *slots*, funciones asociadas a ciertos eventos que obligan la ejecución de estas. Antes del compilado, el MOC modifica el código fuente escrito en C++ extendido para Qt y lo genera en C++ estándar. Por esto, el *framework* en sí y las aplicaciones o librerías que lo usan pueden ser compiladas por compiladores estándar como *Clang*, GCC, ICC, MinGW y MSVC.

Qt se basa en la distribución y ejecución de eventos. Un evento en Qt es un objeto que representa algún suceso de importancia, la mayor diferencia entre una señal y un evento es que estos últimos tienen un objetivo específico en la aplicación desarrollada, mientras que las señales son emitidas sin ninguna garantía de ser procesadas.

Los eventos se pueden generar desde el interior y el exterior de la aplicación, por ejemplo:

- Los objetos *QKeyEvent* y *QMouseEvent* representan algún tipo de interacción del teclado y ratón, y provienen del gestor de ventanas.
- Los objetos *QTimerEvent* son enviados a *QObjet* cuando el temporizador expira y suelen proceder del sistema operativo.
- Los objetos *QChildEvent* son enviados a *QObject* cuando un objeto hijo (se aplica el concepto hijo cuando hay una relación jerarquizada de dependencia en el software de forma que un objeto hijo depende jerárquicamente de uno que sea padre) es añadido o eliminado y suelen ser originados en el interior de la aplicación.

Cuando se produce un evento, este no es enviado en el momento de ocurrencia, sino que es añadido a una cola de eventos. El manejador de eventos trabaja en bucle comprobando la cola de eventos para luego enviar los eventos a sus objetos designados. Este es el motivo por el que se le llama “bucle de eventos”. En la figura 4-2 se puede observar la lógica que sigue el bucle de eventos.

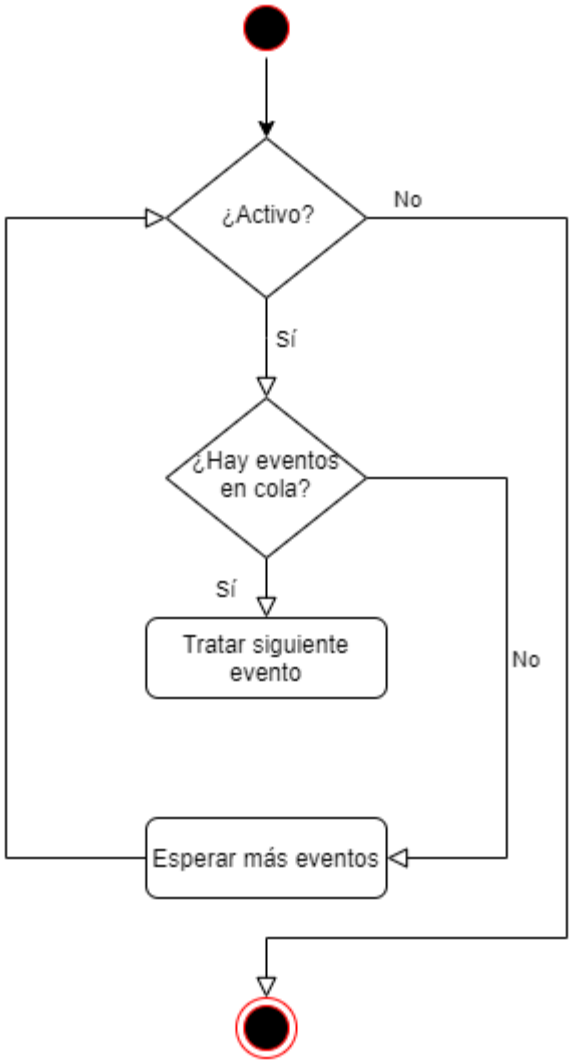


Figura 4-2 Diagrama de flujo del bucle de eventos

A continuación, se muestra una lista de los principales elementos que requieren que el bucle de elementos funcione:

- La interacción y dibujado de *widgets*, diversos elementos que componen la GUI con funciones específicas asociadas que normalmente desembocan en la creación de eventos.
- Temporizadores
- Comunicación de red.

En Qt puede producirse un problema conocido como el bloqueo del bucle de eventos. Este problema consiste en que, cuando se procede a tratar un evento que requiere una larga ejecución, el bucle permanece en espera hasta que el evento que se está ejecutando finalice, lo que produce que se dejen de enviar los siguientes eventos del bucle a sus respectivos objetivos. Las repercusiones más inmediatas son que los *widgets* no puedan interactuar con el usuario ni vuelvan a ser dibujados haciendo que la aplicación quede bloqueada. El bloqueo llega a producir que la mayoría de los gestores de ventanas detecten que la aplicación ya no está tratando los eventos, notificando al usuario que la aplicación no responde.[19]

La razón por la que se ha introducido este reto técnico es debido a que cualquier tipo de comunicación, ya sea mediante sockets, bluetooth o serie es de naturaleza bloqueante pues se espera durante un tiempo la recepción de datos. El caso de la comunicación con Arduino no es una excepción.

La forma de solventar este problema es mediante el uso de hilos, con ellos logramos paralelismo y evitamos el bloqueo del bucle de ejecución. Esto es debido a que no es necesario esperar la finalización del tratamiento de un evento, simplemente se notifica al objeto y se pasa al siguiente.

4.2 Descripción de la implementación

El software escrito para Python está organizado en un paquete llamado "*utils*", en el que encontramos 4 archivos que contienen el grueso del código: *Core.py, CustomWidgets.py, Forms.py* y *Communication.py*. En la figura 4-3 se puede observar un diagrama sobre la disposición y dependencias de los archivos.

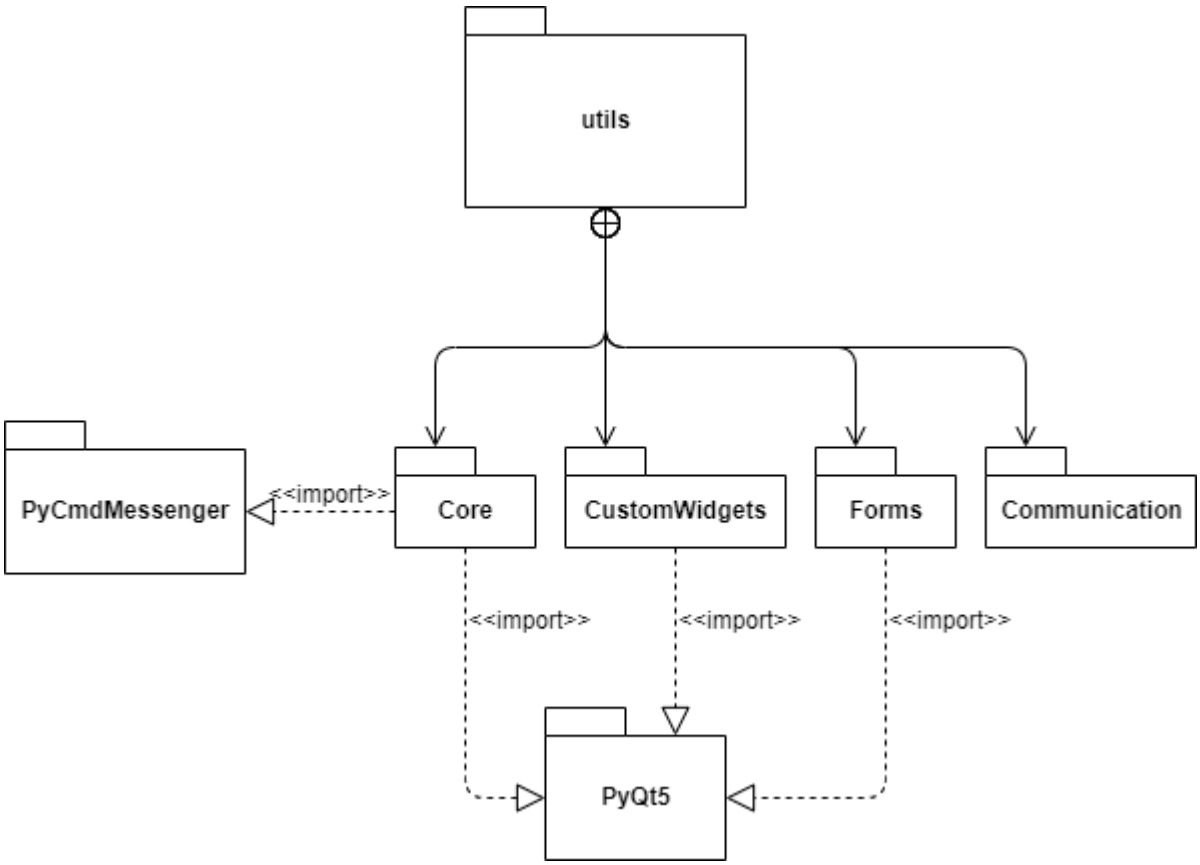


Figura 4-3 Diagrama del módulo *utils*

También hay una librería creada para Arduino que gestionará los mensajes que se enviarán al ordenador, se compone de 2 archivos *pyGUIno.cpp* y *pyGUIno.h*.

A continuación, pasamos a describir las clases contenidas en los archivos del módulo *utils*.

4.2.1 Core.py

Este archivo contiene tres clases, que conforman el núcleo del programa: *PyGUIno, WidgetCoordinator* y *QThreadComm*. Las tareas que tienen designadas como conjunto es el manejo de los mensajes que se reciben y envían a la placa de prototipado, así como la creación y manejo del diseño sobre el cual las clases de *CustomWidgets.py* serán situadas.

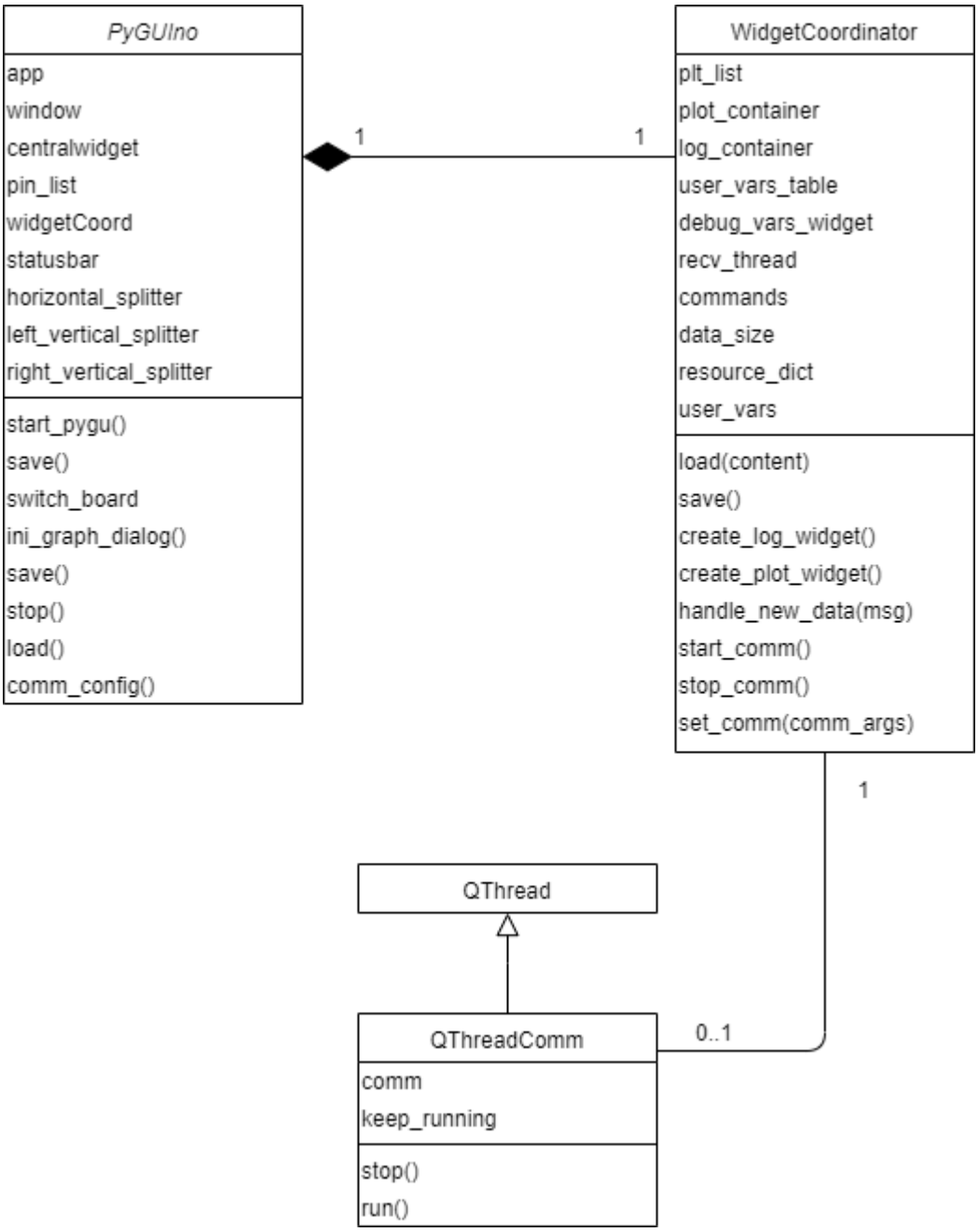


Figura 4-4 Diagrama de clases de *Core.py*

4.2.1.1 PyGUIno

Esta clase es la que define los diseños de la aplicación. También está diseñada para que contenga la barra de herramientas que permite el acceso a diálogos para la creación de gráficas sobre datos obtenidos de Arduino, así como para la configuración de la conexión con la placa Arduino. Los métodos presentes en esta clase son:

- ***start_pygu***: hace que comience la ejecución del bucle de eventos de Qt, así como que se muestre la ventana principal.
- ***load***: permite al usuario guardar la configuración creada para los datos a mostrar en las gráficas y de las variables definidas en *UserVarsTable*.
- ***start***: hace que la ejecución del bucle de eventos de Qt comience, haciendo que la aplicación en sí se ejecute.

- ***save***: permite al usuario guardar la configuración creada para representar gráficamente de datos y de parámetros definidos en *UserVarsTable*.
- ***stop***: para la comunicación con Arduino.
- ***comm_config***: crea una instancia de *ConnectionForm*.
- ***ini_graph_dialog***: crea una instancia de *PlotForm*.
- ***switch_board***: permite seleccionar entre distintos tipos de placas de Arduino.

4.2.1.2 WidgetCoordinator

Tiene como objetivo el tratamiento de los mensajes que recibe de la instancia de la clase *QThreadComm*, notificando sobre los mensajes entrantes que contienen posible información relevante a los widgets definidos en *CustomWidgets.py*. Además, esta clase realiza la gestión del tiempo de vida de la instancia *QThreadComm*.

- ***set_comm***: crea una instancia de la clase *CmdMessenger*.
- ***start_comm***: crea una instancia de *QThreadComm*.
- ***stop_comm***: elimina la instancia de *QThreadComm*.
- ***handle_new_data***: maneja los mensajes recibidos de *QThreadComm* para enviar la información a los objetos de las clases definidas en *CustomWidgets.py*.
- ***create_plot_widget***: instancia un objeto de la clase *WidgetPlot*.
- ***create_logger_widget***: instancia un objeto de la clase *CustomLogger*.
- ***save***: *WidgetCoordinator* llamará a los objetos de la clase *WidgetPlot* para que estos le suministren sus atributos internos necesarios para ser guardados.
- ***load***: *WidgetCoordinator* toma la información entregada junto con la llamada a este método para la instanciación de objetos de la clase *WidgetPlot* y también añadir los parámetros de usuario a la instancia de *UserVarsTable*.

4.2.1.3 QThreadComm

Esta clase que hereda de *QThread*, una clase propia de *PyQt5* para la abstracción de los hilos de ejecución. Tiene como objetivo evitar que la interfaz quede bloqueada con la recepción de datos procedentes de Arduino. La clase se compone de una señal para comunicarse con la clase *WidgetCoordinator* y dos métodos:

- ***run***: este método se encarga de la recepción de los mensajes de la placa Arduino para emitirlos a *WidgetCoordinator*.
- ***stop***: hace que la clase pare la recepción de los mensajes de Arduino.

4.2.2 CustomWidgets.py

Este archivo contiene las clases cuya finalidad es interactuar con el usuario, principalmente mostrando los datos recogidos de la placa Arduino. En la figura 4-5 se pueden apreciar los diagramas de clase contenidos en el archivo. Las cuatro principales clases de este archivo son *CustomLogger*, *WidgetPlot*, *DebugVarsTable* y *UsersVarsTable* que heredan de la clase *QWidget*. Además, *CustomLogger* hereda de la clase *Handler* del módulo *logging*. Por último, la clase *AddUsersVarMenu* hereda de la clase *QDialog*.

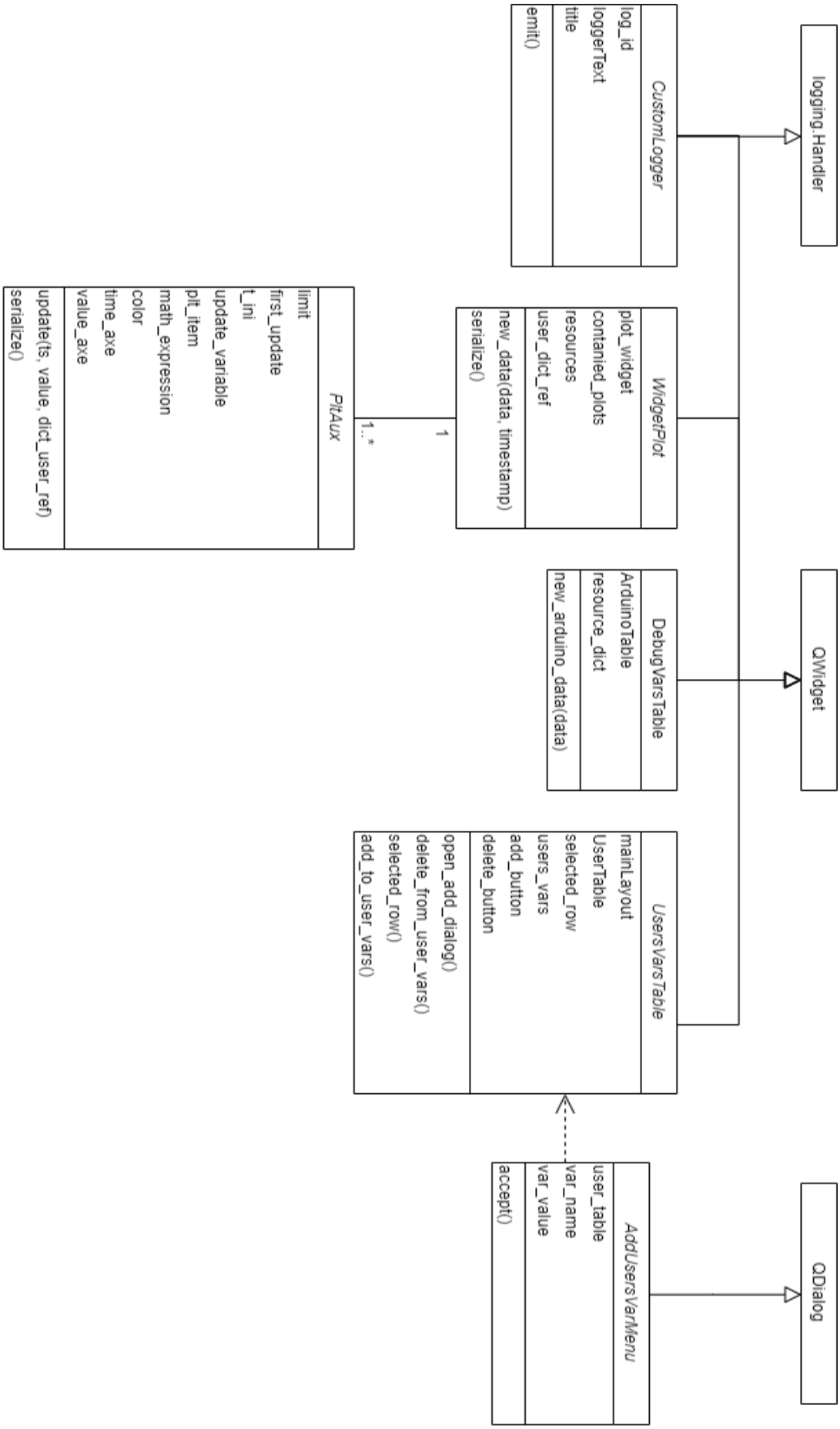


Figura 4-5 Diagrama de clases de *CustomWidgets.py*

4.2.2.1 WidgetPlot

Esta clase es la que está encargada de administrar los datos recogidos de Arduino para su posterior tratamiento gráfico por *PltAux*. Solo tiene un único método:

- ***new_data***: este método itera entre los objetos de la clase *PltAux* para actualizarlos.
- ***serialize***: devuelve los parámetros necesarios para la instanciación de los objetos *PltAux*.

La clase ***PltAux*** es una clase auxiliar definida en el interior de *WidgetPlot*, contiene la información necesaria para poder representar cada curva mediante los datos suministrados por *WidgetPlot*, es decir, los datos necesarios para el eje temporal, los valores asociados, la expresión matemática que será usada y otros atributos. Posee un método llamado *update* con el que se calculan los pares de valores que se van a representar en cada punto de la curva y renderiza la gráfica, obteniéndose así su representación. También posee un método llamado *serialize* que devuelve los atributos necesarios para que esta clase pueda ser instanciada.

4.2.2.2 CustomLogger

El objetivo de esta clase es mostrar los mensajes que proceden de Arduino al ordenador. Muestra también los mensajes intercambiados entre Arduino y sus periféricos mediante I2C o SPI. Sólo posee un único método:

- ***emit***: este método es llamado cuando llega un mensaje al ordenador de parte de Arduino actualizando de manera secuencial, a medida que se reciben, el listado de mensajes alojado en el área de *loggers* de la ventana principal.

4.2.2.3 DebugVarsTable

Muestra información sobre las variables de depuración que son definidas en los *Sketches* de Arduino. Con esta clase podemos conocer el valor de las variables internas de la memoria de Arduino que el usuario ha previsto en su código. Dado que las placas Arduino no tienen ningún método de depuración propio la clase *DebugVarsTable* facilita la depuración de errores ofreciendo el valor de las variables y contribuye a visualizar otros datos de importancia que no necesariamente están relacionados con la depuración. Posee un único método:

- ***new_arduino_data***: determina si al llegar nueva información debe añadirla a la tabla o solo actualizar algunos campos como el valor y la dirección de memoria.

4.2.2.4 UserVarsTable

Esta clase permite que los usuarios sean capaces de añadir, para su uso posterior, parámetros propios en las expresiones matemáticas definidas que se van a usar para la representación gráfica de los datos transformados mediante las mismas. Posee dos botones para añadir y eliminar parámetros. Esta clase cuenta con cuatro métodos:

- ***open_add_dialog***: crea una instancia de la clase *AddUserVarMenu*. Este método está asociado al botón de añadir parámetro como un evento de pulsación.
- ***delete_from_user_vars***: elimina de la tabla el parámetro que el usuario tiene seleccionado en el momento de pulsar el botón eliminar, así como del atributo interno *user_vars*. Este método está asociado al botón de eliminar parámetro como un evento de pulsación.
- ***selected_row***: método que se ejecuta cuando el usuario selecciona un parámetro de la tabla para actualizar un atributo del objeto que indica el número de fila seleccionado. El atributo que actualiza es usado por el método *delete_from_user_vars*.
- ***add_to_user_vars***: este método añade una entrada a la tabla y actualiza los parámetros definidos por el usuario.

AddUsersVarsMenu es un cuadro de diálogo en el que se introducen el nombre del parámetro y su valor, solo se aceptan valores numéricos. Posee único método llamado *accept*, este método se ejecuta al cumplimentar y aceptar en el cuadro de diálogo, llamando al método *add_to_user_vars* de *UserVarsTable*.

4.2.3 Forms.py

Este archivo contiene los cuadros de diálogo con los que el usuario interactuará para introducir los datos necesarios para el correcto funcionamiento del programa. Todas las clases presentadas en este apartado heredan de la clase *QDialog* a excepción de *ErrorMessage*, que hereda de *QErrorMessage*, y *ErrorMessageWrapper* que no hereda de ninguna clase.

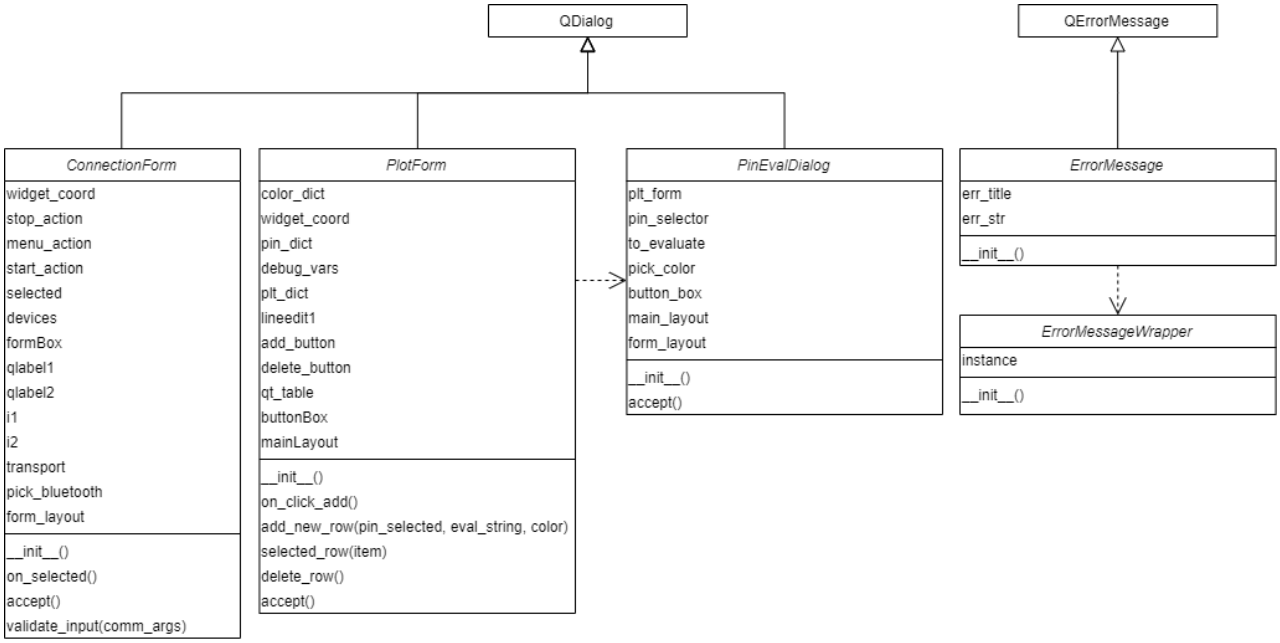


Figura 4-6 Diagrama de clases de *Forms.py*

4.2.3.1 ConnectionForm

Esta clase se encarga de recolectar los datos necesarios para configurar el tipo de conexión, junto con sus respectivos parámetros, del ordenador con la placa Arduino. El diseño de formulario contenido en el cuadro de diálogo se adapta para solicitar los parámetros de la conexión según el botón seleccionado en el su interior. Los métodos de la clase son tres:

- ***on_selected***: cambia los campos del formulario contenido en el diálogo para adaptarlos según el tipo de conexión seleccionada. Toma ventaja de la funcionalidad de Qt de señales y *slots*.
- ***accept***: obtiene los valores introducidos en los campos del formulario y los valida llamando a *validate_input*, en caso de que sean erróneos crea una instancia de *ErrorMessageWrapper* con el fin de avisar al usuario de que los parámetros son erróneos.
- ***validate_input***: este método comprueba que los datos introducidos son correctos. En el caso de conexión serie se comprueba que el puerto introducido exista y que la tasa de baudios sea un número entero positivo. Para la conexión bluetooth siempre se aceptarán los datos introducidos como válidos debido a que el usuario solo podrá escoger entre dispositivos bluetooth escaneados por el ordenador y por tanto no se comprueba la validez de los datos de entrada en este caso.

4.2.3.2 PlotForm

El objetivo que persigue esta clase es la creación de instancias de la clase *WidgetPlot*. Para ello, deberá comunicarse con *WidgetCoordinator*. Entre sus atributos de la clase *PlotForm*, destaca un objeto de la clase *QTableWidget* pues dicho objeto será quien contenga la información que se le suministrará a *WidgetCoordinator*. Los métodos de la clase son los siguientes:

- ***on_click_add***: instancia un objeto de la clase *PinEvalDialog*.
- ***add_new_row***: añade a la tabla contenida en el diálogo una fila con los datos que el usuario ha introducido en la instancia de *PinEvalDialog*.
- ***accept***: este método es llamado para cerrar el diálogo y que la instancia de *WidgetCoordinator* cree un objeto de la clase *WidgedPlot*.
- ***selected_row***: actualiza el valor de un atributo interno para saber la fila seleccionada
- ***delete_row***: elimina la fila seleccionada y en caso de quedar la tabla vacía, deshabilita el botón de "Suprimir".

Adicionalmente, la clase *PlotForm* define en su interior la clase ***PinEvalDialog***. Esta clase es otro diálogo que contiene un formulario cuyos campos deberán ser completados por el usuario con los datos que necesitarán las curvas, es decir, la expresión matemática, el color y la variable o pin que se utilizará para actualizar la curva. Solo cuenta con un único método, *accept*. Este método llama a *add_new_row* del objeto *PlotForm* desde el que se instanció para que añada los datos introducidos a la tabla y cierre el diálogo.

4.2.3.3 ErrorMessageWrapper y ErrorMessage

El objetivo estas dos clases es la muestra de errores dirigidos al usuario por diversos motivos, los cuales son especificados cuando estas clases son instanciadas. Al contrario que el resto de las clases, estas solo poseen los métodos más básicos de la programación orientada a objetos, el constructor y el método de borrado. Este último método es llamado por el colector de basura, parte del sistema de Python se encarga de la liberación de recursos utilizados por los objetos que ya no son referenciados.

Debido a que no se desea que el usuario sea interrumpido con más de un mensaje de error a la vez, el manejo de errores está diseñado usando un patrón *singleton*, patrón de diseño de software que consiste en la única instancia de una clase de forma simultánea.

La forma en la que se implementa este patrón *singleton* es mediante sendas clases. *ErrorMessage* es quien mostrará el mensaje de error. No obstante, cuando se produce un error es *ErrorMessageWrapper* quien es llamado a ser instanciado. En su constructor, llamará solo al constructor de *ErrorMessage* si el atributo de clase *instance* es nulo. El siguiente diagrama de flujo refleja este comportamiento.

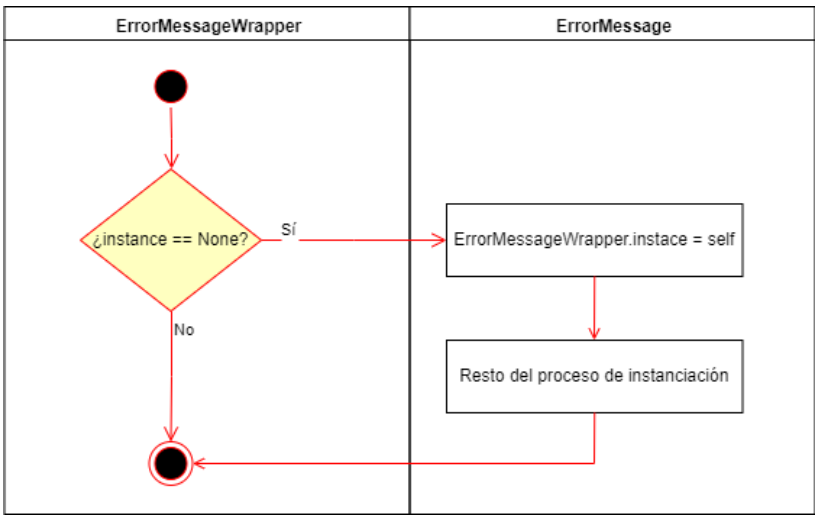


Figura 4-7 Lógica seguida para implementar el patrón *singleton*

Finalmente, cuando *ErrorMessage* es llamado por el colector de basura, ejecuta el método *__del__()* que reestablece el valor del atributo de clase *instance* de *ErrorMessageWrapper* a su valor nulo por defecto.

4.2.4 Communication.py

En el apartado 4.1.1 se habló sobre la posibilidad de extender las funcionalidades del módulo *PyCmdMessenger* a otros tipos de conexión. Esto es posible si conceptualmente abordamos el reto como si se tratase de la implementación de una interfaz con los siguientes atributos.[18]

Tabla 4-1 Atributos a implementar

Atributo	Significado
device	Con esta variable se debe indicar una forma de identificar la placa. Dependerá del tipo de implementación que se desee realizar.
timeout	Tiempo máximo para la recepción de un mensaje.
int_bytes	Número de bytes necesarios para representar un *integer*. Sólo podrá valer 2, 4 u 8.
long_bytes	Número de bytes necesarios para representar un *long*. Sólo podrá valer 2, 4 u 8.
float_bytes	Número de bytes necesarios para representar un *float*. Sólo podrá valer 4 u 8.
double_bytes	Número de bytes necesarios para representar un *double*. Sólo podrá valer 4 u 8.
int_min	Valor mínimo posible que puede tomar un *integer*.
int_max	Valor máximo posible que puede tomar un *integer*.
unsigned_int_min	Valor mínimo posible que puede tomar un *unsigned integer*. Siempre será 0.
unsigned_int_max	Valor máximo posible que puede tomar *unsigned intger*.
long_min	Valor mínimo posible que puede tomar *long*
long_max	Valor máximo posible que puede tomar *long*
unsigned_long_min	Valor mínimo posible que puede tomar un *unsigned long*. Siempre será 0.
unsigned_long_max	Valor mínimo posible que puede tomar un *unsigned long*.
float_min	Valor mínimo posible que puede tomar un *float*
float_max	Valor máximo posible que puede tomar un *float*
double_min	Valor mínimo posible que puede tomar un *double*.
double_max	Valor máximo posible que puede tomar un *double*.
int_type	Su valor dependerá de int_bytes. Si int_bytes toma el valor 2 su valor será “<h”, para el valor 4 será “<i” y para el valor 8 será “<l”
unsigned_int_type	Su valor dependerá de int_bytes. Si int_bytes toma el valor 2 su valor será “<H”, para el valor 4 será “<I” y para el valor 8 será “<L”
long_type	Su valor dependerá de long_bytes. Si long_bytes toma el valor 2 su valor será “<h”, para el valor 4 será “<i” y para el valor 8 será “<l”

unsigned_long_type	Su valor dependerá de long_bytes. Si long_bytes toma el valor 2 su valor será "<H", para el valor 4 será "<I" y para el valor 8 será "<L"
float_type	Su valor dependerá de float_bytes. Si float_bytes toma el valor 4 su valor será "<f", para el valor 8 será "<d"
double_type	Su valor dependerá de double_bytes. Si double_bytes toma el valor 4 su valor será "<f", para el valor 8 será "<d"

Adicionalmente debemos implementar los siguientes métodos para que la clase *CmdMessenger* sea capaz de hacer uso del nuevo tipo de conexión.

Tabla 4-2 Métodos a implementar

Método	Descripción
open()	Hace que la conexión con Arduino comience. Debe ser llamada durante la creación del objeto.
read()	Lee un byte y el cual será dado como retorno.
readline()	Lee una línea, es decir, hasta "\r\n" y la devuelve como retorno.
write(message)	Envía el argumento a Arduino.
close()	Finaliza la conexión.
connected()	Devuelve el estado de la conexión, siendo "*True*" si hay una conexión y "*False*" en caso contrario.

4.2.4.1 ArduinoBoardBluetooth

Esta clase añade el soporte bluetooth, para ello implementa todos los atributos y métodos expuestos en las tablas 4-1 y 4-2. Todos los atributos necesarios para un correcto uso por parte de la clase *CmdMessenger* se encuentran definidos e inicializados en el constructor de la clase. Como consecuencia de lo anterior, el diagrama de clase de *ArduinoBoardBluetooth* refleja todos estos métodos.

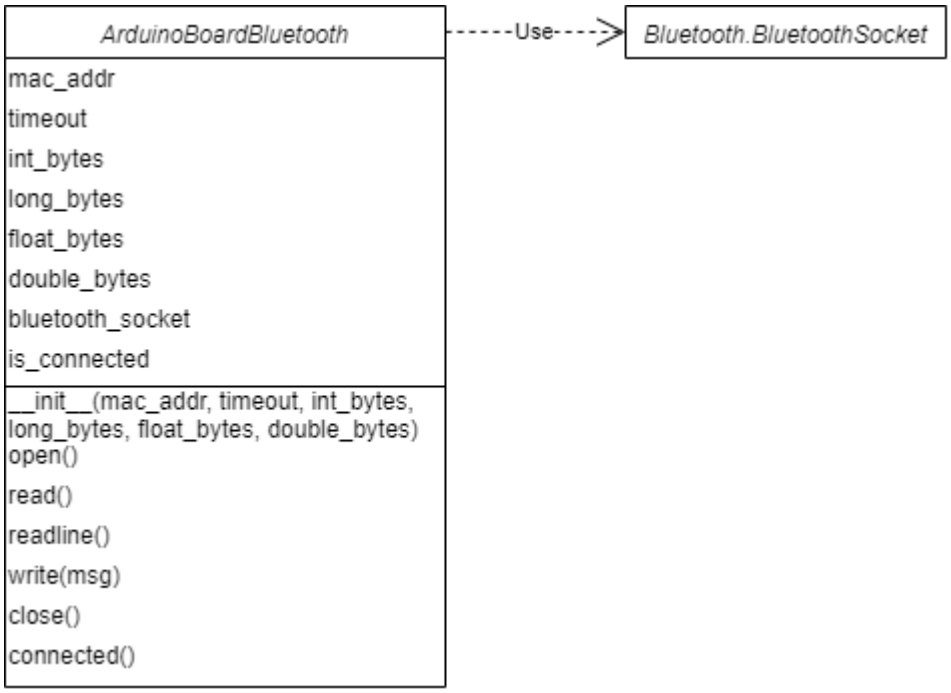


Figura 4-8 Diagrama de clases de *Communication.py*

Respecto a los métodos para la comunicación mediante bluetooth, en el siguiente extracto de código se puede apreciar el uso de sockets para la apertura y cierre de la conexión.

```
def open(self):
    port = 1
    try:
        self.bluetooth_socket = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
        self.bluetooth_socket.settimeout(self.timeout)
        self.bluetooth_socket.connect((self.mac_addr, port))
        time.sleep(2.0)
    except bluetooth.btcommon.BluetoothError as err:
        # Error handler
        print(err)
    else:
        self._is_connected = True

def close(self):
        if self._is_connected:
                self.bluetooth_socket.close()
                self._is_connected = False
```

La forma en la que Bluetooth está diseñado es mediante el uso de sockets, dando a escoger entre dos tipos de protocolos:

- RFCOMM: diseñado para simular puertos RS-232. Busca dar las mismas características que una conexión TCP, fiabilidad para transmitir flujos de datos de punto a punto. Utiliza L2CAP para ser transportado.
- L2CAP: es un protocolo orientado a conexión, enviando datagramas de una longitud fija. Puede ser configurado para dar distintos niveles de fiabilidad.

Para la clase *ArduinoBoardBluetooth* se ha optado por usar el protocolo RFCOMM.[20]

4.2.5 Librerías creadas para Arduino

Para la interfaz cumpla con el objetivo marcado, se debe recibir información desde la placa Arduino. Aunque, el software previamente descrito funcione correctamente en su totalidad, sería inútil sin esta información de la placa. Se han desarrollado con este propósito librerías específicas para Arduino que deberán ser utilizadas por los usuarios cuando estos realicen sus programas si quieren tener la posibilidad de visualizar la información en el ordenador mientras que en Arduino se ejecute su programa.

4.2.5.1 PyGUIno

Con esta librería el usuario podrá utilizar funciones que notificarán al ordenador valores contenidos en variables del programa, así como valores de los pines y reinicios de la placa Arduino. Consta de tres funciones principales:

- ***attach_callbacks***: esta función deberá ser llamada por el programa creado por el usuario ya que es necesaria para poder hacer uso de las otras funciones contenidas en esta librería.
- ***transmit_pin_value***: permite la transmisión del valor del pin especificado en su argumento al ordenador.
- ***add_update_debug_var***: transmite los datos necesarios para identificar a una variable de depuración o control en el ordenador, es decir, nombre establecido por el usuario, valor asociado, dirección de memoria y tipo de dato.

4.2.5.2 PyGUInoSPI

Mediante esta librería se pueden visualizar los valores recibidos y enviados cuando se produzca comunicación a través de SPI. Se ha utilizado el patrón fachada para la creación de esta librería, que permite al usuario usar el objeto de la clase *PyGUInoSPI* como si de la clase *SPI*, propia de la librería *SPI.h*, se tratase. Para ello se han implementado la mayor parte de los métodos propios de la clase *SPI*, no obstante, los métodos que han añadido un comportamiento adicional con el fin de enviar información al ordenador son los siguientes:

- ***transfer***: envía al ordenador el valor, de 8 bits de longitud, que se enviará por el protocolo *SPI* y el valor que se recibe en el registro *SPDR*.
- ***transfer16***: sigue el mismo comportamiento que el anterior método, pero está diseñado para ser utilizado en caso de enviar valores de 16 bits de longitud por parte de Arduino al periférico con el que se comunique.
- ***read***: permite leer el valor del registro *SPDR*, está pensado para ser utilizado en la función que el usuario implementará para tratar cuando se produzca una llegada de información por parte del protocolo.

4.2.5.3 PyGUInoWire

Al igual que la librería previa, se basa en el uso de uso del patrón fachada para dar sus servicios al usuario. No obstante, esta librería es específica para el uso de I2C. Su patrón fachada se aplica a la librería *Wire.h*, mediante el objeto *PyGUInoWire*. Los métodos que tienen un comportamiento adicional son:

- ***requestFrom***: cambia el atributo *read_addr* de la instancia de *PyGUInoWire*.
- ***write***: envía el valor o los valores que van a ser transmitidos por Arduino mediante I2C al ordenador
- ***read***: envía al ordenador el valor leído por Arduino en el buffer de recepción para I2C.

5 DESCRIPCIÓN DEL FUNCIONAMIENTO DE LA IMPLEMENTACIÓN

Ahora que la implementación ha sido descrita en su totalidad, en este capítulo se hablará sobre cómo interactúan las clases definidas entre ellas. La comunicación de las clases entre sí y de estas con los actores es un proceso en el que intervienen las distintas partes con diversas atribuciones que implican una mayor importancia para algunas de ellas en el proceso de comunicación. Una representación básica del flujo de la información puede verse en el siguiente esquema.

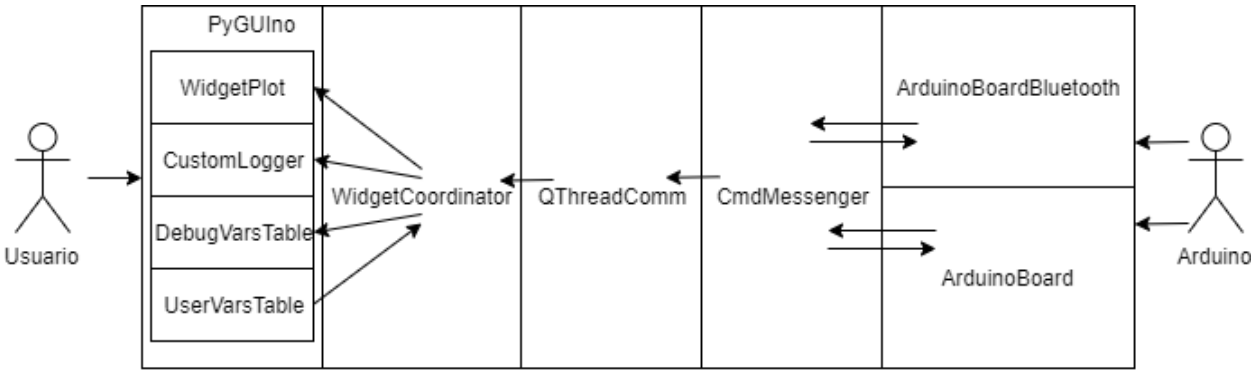


Figura 5-1 Representación básica de la comunicación interna

Además, se puede observar el papel que ejerce la instancia de *WidgetCoordinator* pues solo esta clase tiene permitida la gestión de las clases que se definieron en *CustomWidgets.py*

5.1 Inicio del programa

El proceso de inicio del programa requiere la instanciación de varios objetos de distintas clases. Con el fin de clarificar este proceso que se describirá a continuación se acompaña el siguiente diagrama.

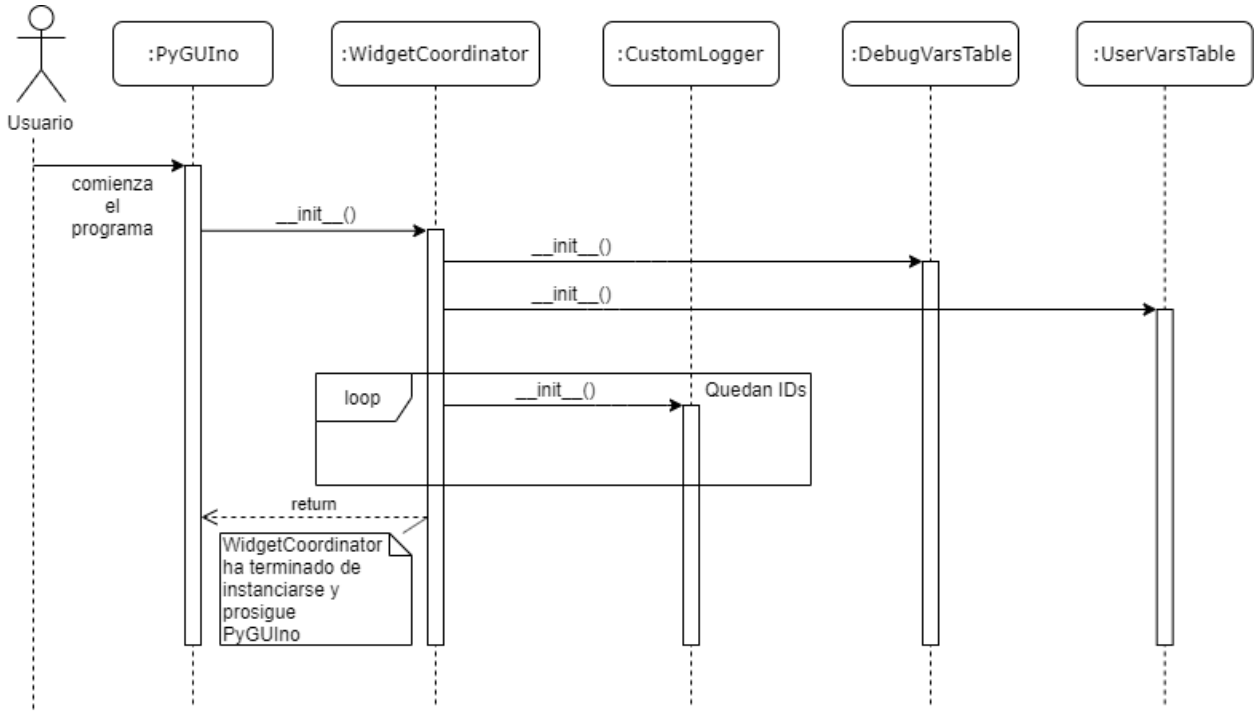


Figura 5-2 Diagrama de secuencia del inicio del programa

Como se aprecia en el diagrama tras el comienzo del programa por parte del usuario, se instancia la clase *PyGUIno*. El constructor de esta clase está estructurado de la siguiente forma:

1. Crea la ventana principal del programa y la configura, dándole el tamaño adecuado, nombre e icono. También se escogen las reglas que seguirá cuando haya un cambio en el tamaño de la ventana. También se le añade al diseño.
2. Se crea la barra de herramientas. También se crean las acciones que serán contenidas en ella, estas acciones también son configuradas con sus iconos, nombre y mensaje de ayuda asociado. Finalmente, se crea un cuadro combinado para la elección del tipo de placa.
3. Se añaden los manejadores de señales a las acciones previamente definidas para cuando estas son pulsadas. Además, al cuadro combinado también se le asocia una función para cuando se produce una nueva selección de sus opciones. Tras esto, son añadidos a la barra de herramientas.
4. La barra de herramientas, junto a la barra de estados, es añadida a la ventana. Se procede a instanciar *WidgetCoordinator*.
5. Una vez que la instancia de *WidgetCoordinator* se ha realizado, se obtienen los elementos que se corresponden con las áreas de la interfaz definidas en el capítulo 3. Se procede a hacer la separación entre ellas mediante *splitters* y son añadidas al diseño.

Como se puede observar, el constructor *PyGUIno* crea un objeto de la clase *WidgetCoordinator*, esto hace que se ejecute el constructor de *WidgetCoordinator* cuyo proceso de actuación puede describirse de la siguiente forma:

1. Establece los atributos que serán usados posteriormente a la hora de configurar la comunicación, es decir, los distintos identificadores de los tipos de mensajes y sus argumentos asociados, también referencias a las futuras instancias de *CmdMesseger* y *QThreadComm*. Asimismo otros atributos necesarios para la correcta notificación de los mensajes a aquellos futuros objetos de las clases *WidgetPlot* y *DebugVarsTable* que necesiten datos.
2. Creación de los *widgets* que se añadirán a las áreas de trabajo en la ventana. Creará para el área de

gráficas y el área de *loggers* un objeto de la clase *QTabWidget*, para el área de variables de depuración y el área de definición de parámetros de usuario una instancia de *DebugVarsTable* y *UsersVarsTable* respectivamente.

3. Inicia los *loggers*, creando las pestañas de clasificación de los mismos e instanciando para cada *logger* un objeto de la clase *CustomLogger*.

Se van a explicar los procesos seguidos por los constructores de las clases *DebugVarsTable, UsersVarsTable* y *CustomLogger*.

- *DebugVarsTable*: se inicia haciendo una llamada al constructor de la clase *QWidget*, de la que hereda, y luego crea el diseño que contendrá el objeto de la clase *QTableWidget*, lo configura dando un correcto nombre a las columnas y sus políticas de tamaño y añade el objeto al diseño.
- *UsersVarsTable*: su constructor sigue el mismo proceso que el de *DebugVarsTable*, con algunas acciones adicionales que son la creación de dos botones y la asignación de las funciones que manejarán el evento de pulsado.
- *CustomLogger*: hace una llamada a los constructores de las clases sobre las que hereda, *QWidget* y *Handler*, después crea el diseño, instancia un objeto de la clase *QPlainTextEdit*, lo configura a solo lectura y lo añade al diseño.

5.2 El proceso de comunicación

Antes de la recepción de mensajes entre Arduino y el ordenador el usuario debe configurar la comunicación. Este proceso está ilustrado en la figura 5-3, cuando el usuario realiza una pulsa sobre la acción "Configurar comunicación" hace que se ejecute el método *comm_config* de la instancia de *PyGUIno*, cuyo objetivo no es otro que abrir un diálogo para escoger el tipo de comunicación. El usuario interaccionará ahora con el dialogo escogiendo el tipo de comunicación y los parámetros que esta requiere.

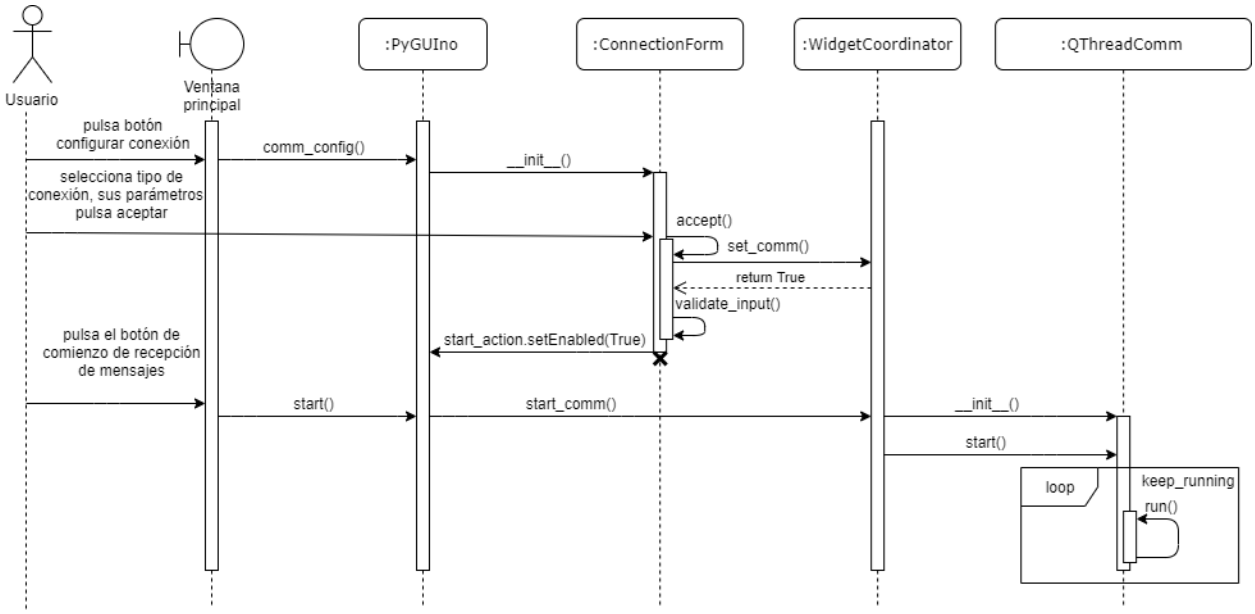


Figura 5-3 Diagrama de secuencia de la configuración de la comunicación

Una vez el usuario haya introducido los parámetros de la conexión y pulse aceptar en el cuadro de diálogo, se llamará a la instancia de *WidgetCoordinator* para la creación de un objeto de la clase *CmdMessenger* además de comprobar la validez de los parámetros.

Superado el paso anterior y ahora que toda la configuración de la comunicación ha sido realizada, esta puede comenzar. El usuario pulsa el botón de comienzo, creando una instancia de *QThreadComm* y comenzando esta

a recibir datos.

5.2.1 Tratamiento de los mensajes

El tratamiento de los mensajes se realiza de forma distinta para cada tipo, aunque cada mensaje recibido es siempre notificado a los *loggers*. Los tipos más importantes de mensajes son dos, cuyo tratamiento será descrito a continuación en profundidad.

5.2.1.1 Transmisión del valor de un pin

Desde la placa Arduino se transmite al ordenador el número con el que se identifica el pin y su valor correspondiente. Con esta información se inicia una cadena de eventos en el programa. La figura 5-4 muestra el proceso.

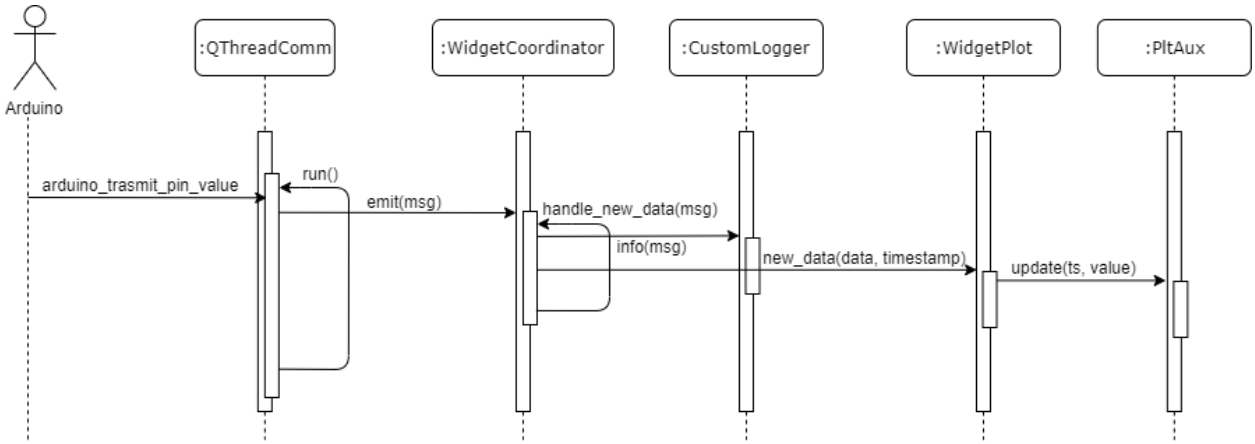


Figura 5-4 Diagrama de recepción y tratamiento de la identificación y valor de un pin

El proceso comienza con la recepción del mensaje por parte de *QThreadComm* que se lo entrega sin cambio alguno a *WidgetCoordinator* mediante su método *handle_new_data*.

WidgetCoordinator notifica a los objetos de la clase *CustomLogger* de la llegada de un mensaje y estos lo muestran al usuario en el área de *loggers* mediante su objeto *QPlaintTextEdit*.

Se produce un filtro para ver el tipo de mensaje mediante el valor del identificador del mensaje que procede de la tupla proporcionada por la instancia de *CmdMessenger*. Esta tupla contiene el identificador del mensaje, los argumentos asociados y por último el instante de recepción del mensaje.

WidgetCoordinator entrega a cada objeto de la clase *WidgetPlot* el mensaje.

Cada instancia de *WidgetPlot* comprueba si el mensaje es necesario para sus objetos *PltAux*. En este caso, el objeto *PltAux* que lo necesite, calculará el valor correspondiente. El instante temporal se añadirá su atributo *time_axe*, y el valor calculado se añadirá a su atributo interno *value_axe*. Ambos valores se guardan, en la última posición descartando a su vez el valor inicial de dicho atributo cuando en la lista de este se acumulan el máximo número de valores permitido.

Tras todo este proceso, la curva que los representa es reiniciada para presentar en cada momento los puntos correspondientes a los pares de valores (instante, valor) que se encuentran en los atributos ajustándose de forma automática por parte de *pyqtgrapgh* las escalas de los ejes.

5.2.1.2 Envío de variable de depuración

Arduino puede enviar los datos correspondientes a una variable previamente definida por el usuario en el *sketch* que puede ser usada para la ejecución de pruebas, control de funcionamiento o depuración de código.

Desde Arduino se envía el identificador del mensaje y como argumentos un identificador del tipo de datos, la

dirección de memoria, el nombre de la variable y el valor como una secuencia de bytes.

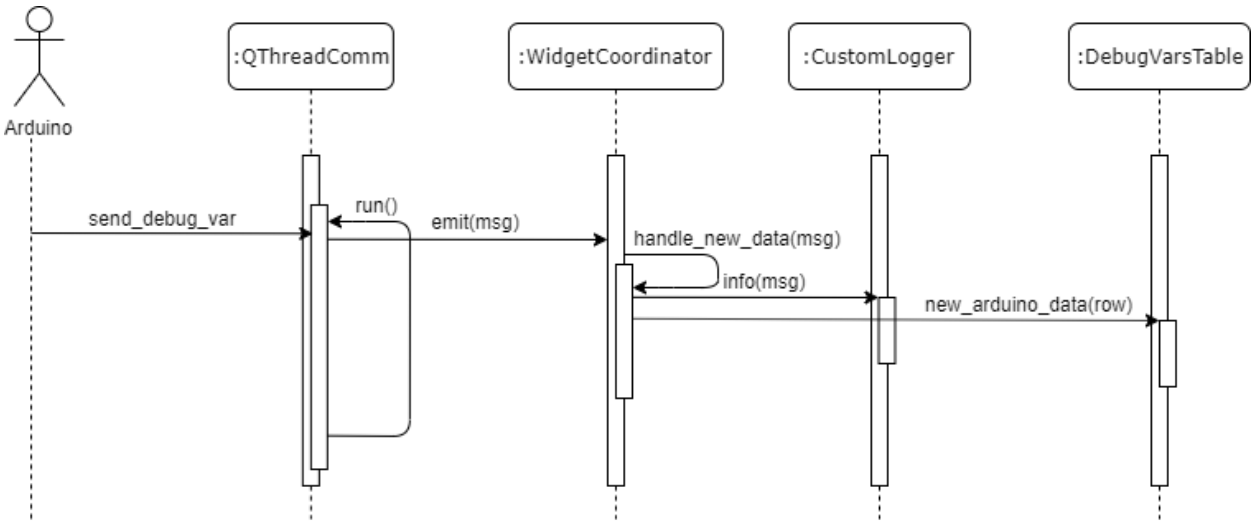


Figura 5-5 Recepción y tratamiento de una variable de depuración procedente de Arduino

El proceso se inicia de la misma forma que con el resto de los mensajes, siendo recibido por *QThreadComm* y enviado a *WidgetCoordinator* junto con sus argumentos y el instante de recepción.

WidgetCoordinator ejecuta la función *handle_new_data* con la cual filtra el mensaje, esta vez detecta que es el envío de una variable de depuración. Ya que esta vez no tiene a múltiples objetos que notificar procede a preparar los datos para la instancia de *DebugVarsTable* donde los entregará como un diccionario con los siguiente pares clave, valor:

- *data_type*: texto indicando el tipo de dato.
- *addr*: dirección de memoria donde se encuentra la variable en Arduino.
- *name*: nombre que designó el usuario en su *Sketch* para la variable.
- *value*: valor de la variable transformada al tipo de dato apropiado.

De los argumentos recibidos, el identificador del tipo de datos y el valor como secuencia de bytes no pueden ser mostrados en pantalla sin ser tratados. Ambos parámetros se usan para pasar del identificador del tipo de dato a una cadena de texto que informa del tipo que se trata y determinar el tipo de función a utilizar para transformar la cadena de bytes que contiene el valor de la variable en el valor real.

Ahora que finalmente todos los datos están listos para ser añadidos a la tabla de *DebugVarsTable*, son incorporados mediante el método *new_arduino_data*, que comprueba si la variable está ya en la tabla y actualiza su valor o si es una nueva entrada y lo anexa.

5.3 Creación de nuevas gráficas

El proceso de creación de gráficas por el usuario requiere el paso por varios diálogos y requiere la interacción de distintos objetos. De manera esquemática la totalidad de este puede observarse en la figura 5-6.

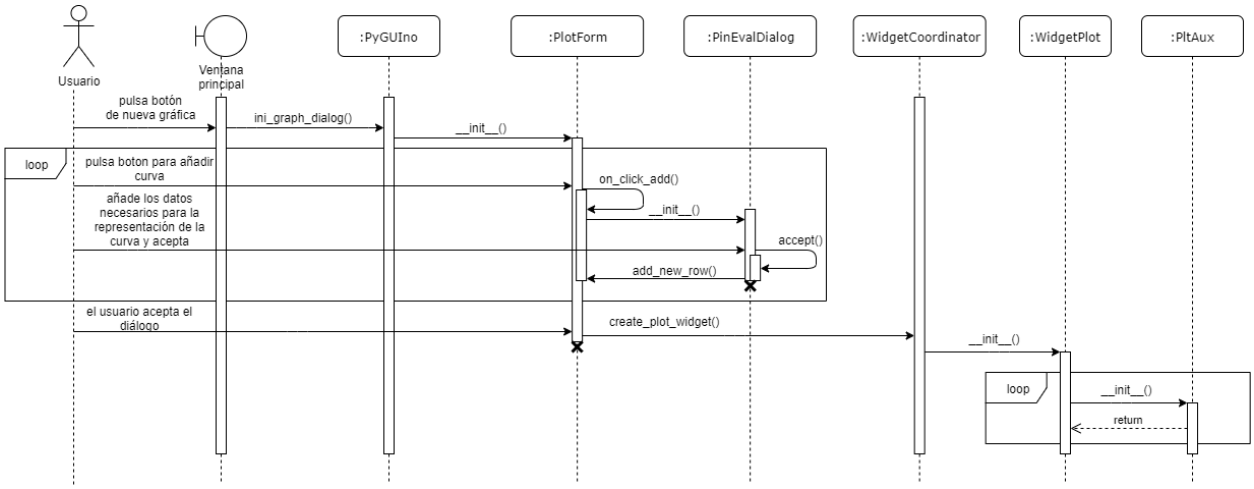


Figura 5-6 Diagrama de secuencia para creación de gráficas

El proceso puede describirse de la siguiente manera:

1. Cuando el usuario desee añadir una gráfica, pulsará el botón contenido en la barra de herramientas que lanza el método *ini_graph_dialog* de la instancia de *PyGUIno*, esto instancia un objeto de la clase *PlotForm* para que el usuario interactúe con él a partir de ese momento.
2. Cuando el usuario quiere introducir los datos sobre las curvas llamando a un nuevo cuadro de diálogo, siendo este una instancia de la clase *PinEvalDialog*. En este diálogo el usuario añade los datos que son el pin con el que se actualizará la gráfica resultante cuando se registre un mensaje de este, la expresión matemática que se utilizará para la evaluación de los datos y el color con el que se representarán. Una vez el usuario pulse "Aceptar" en el cuadro de diálogo se volverá al cuadro previo, objeto de la clase *PlotForm*. Este paso puede darse de forma indefinida.
3. Al pulsar "Aceptar" en el cuadro de diálogo la instancia de *PlotForm* será eliminada, habiendo realizado una llamada previamente al método *create_plot_widget* de la clase *WidgetCoordinator*.
4. *WidgetCoordinator* hará una instancia de *WidgetPlot*, que será contenido en una pestaña de *QTabWidget*. Al mismo tiempo, *WidgetPlot* instanciará tantos objetos de la clase *PltAux* como entradas haya realizado previamente el usuario en *PlotForm*.

5.4 Uso desde el punto de vista del usuario

Para ilustrar el uso de la interfaz gráfica y el funcionamiento del software desde el punto de vista del usuario, se ha comprobado mediante un ejemplo concreto que se obtienen los resultados esperados.

5.4.1 Descripción del ejemplo

A la hora de proponer un ejemplo de uso se ha buscado uno que permita mostrar las capacidades de la mayor parte del software creado, tanto para el ordenador como para Arduino. Por este motivo el escenario elegido consta de dos placas Arduino comunicadas mediante I2C, aunque también podría haberse hecho mediante comunicación SPI.

La finalidad perseguida es aplicar el uso del software creado para Arduino, así como de la interfaz gráfica para que se muestre en tiempo real una gráfica con la temperatura ambiente detectada mediante un sensor TMP36. Para ello se han usado dos placas Arduino comunicadas mediante I2C, siendo una de ellas la que obtiene la señal procedente del sensor y la envía a la otra placa Arduino, siendo esta última la que envía el valor del sensor al ordenador mediante una conexión Bluetooth. Es evidente que con una sola placa podría implementarse este ejemplo, pero, en ese caso, no se mostrarían una buena parte del software desarrollado.

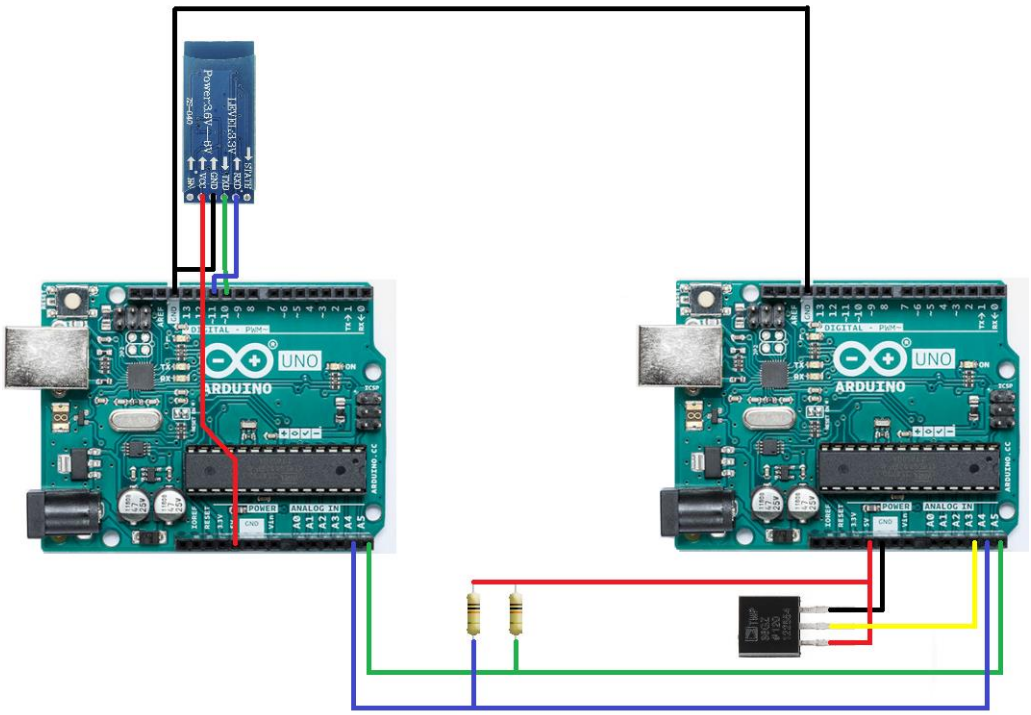


Figura 5-7 Esquema del montaje realizado

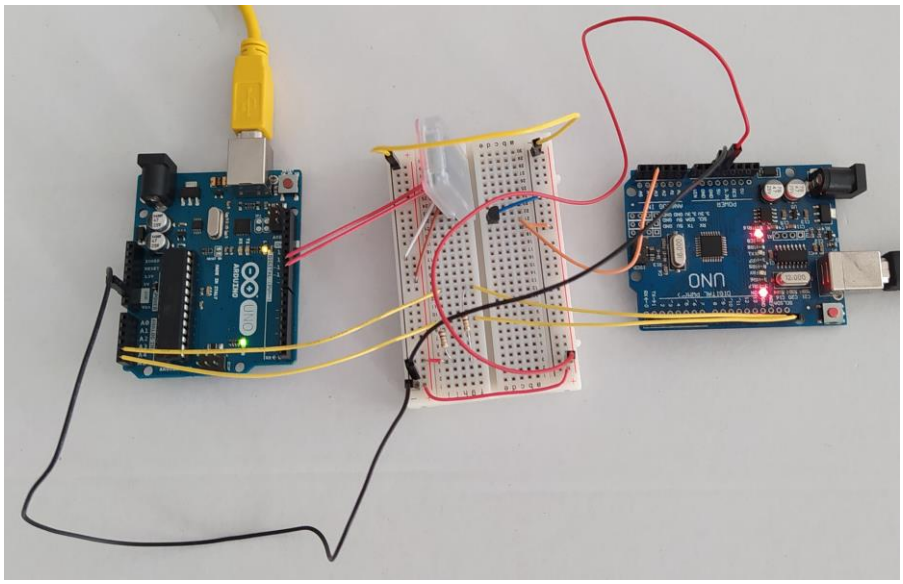


Figura 5-8 Montaje realizado

5.4.2 Códigos de Arduino usados en el ejemplo

En este apartado se muestran los códigos usados para la creación de este ejemplo. Debido a que la comunicación es mediante I2C entre las placas de Arduino, una actuará como maestro, quien realizará las peticiones, y otra como esclavo, quien las servirá.

Código ejecutado por la placa maestro

Este código tiene la finalidad de pedir a la otra placa un byte que contendrá el valor del voltaje recogido procedente del sensor en la placa que actúa como esclavo y envía, a través de la conexión Bluetooth, dicho valor junto con un contador del número de ejecuciones de la función *loop* al ordenador.

```
#include "PyGUInoWire.h"
#include <SoftwareSerial.h>
#include <CmdMessenger.h>

#define DIR_ESCLAVO 80

int contador = 0;
SoftwareSerial bluetooth = SoftwareSerial(10, 11);  // Rx y Tx
CmdMessenger cmd = CmdMessenger(bluetooth);
PyGUInoWire pWire = PyGUInoWire(cmd);

void setup(){
  Serial.begin(9600);
  bluetooth.begin(9600);
  attach_callbacks(cmd);
  pWire.begin();
}

void loop(){
  pWire.requestFrom(DIR_ESCLAVO, 1);
  if(pWire.available()){
    int temp_v  = pWire.read();
    add_update_debug_var(int_var, temp_v, "temp_v");
  }
  add_update_debug_var(int_var, contador++, "contador");
}
```

Código ejecutado por la placa esclavo

Este código tiene como único fin el envío del voltaje leído desde el pin conectado al sensor cuando la placa que actúa como maestro le hace una petición.

```
#include <Wire.h>

#define DIR_ESCLAVO 80

void setup(){
  Serial.begin(9600);
  Wire.begin(DIR_ESCLAVO);
  Wire.onRequest(requestEvent);
}

void loop(){
}

void requestEvent(){
  uint8_t temp = analogRead(A3);
  Wire.write(temp);

}
```

5.4.3 Visualización del proceso en la interfaz gráfica

El proceso se inicia mediante la selección del tipo de placa que será utilizada en la conexión del ordenador y la placa de Arduino.

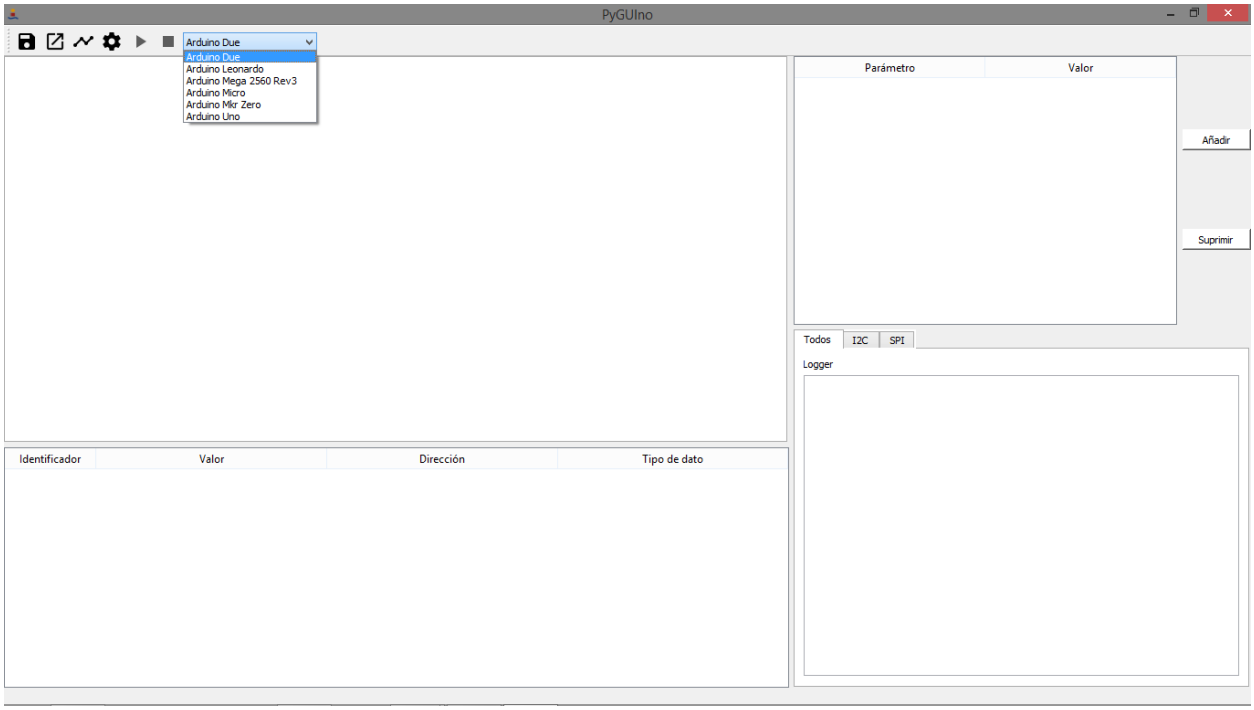


Figura 5-9 Selección de la placa utilizada

Una vez seleccionado el tipo de placa, se procede a cargar la configuración desde un fichero donde previamente fue almacenada.

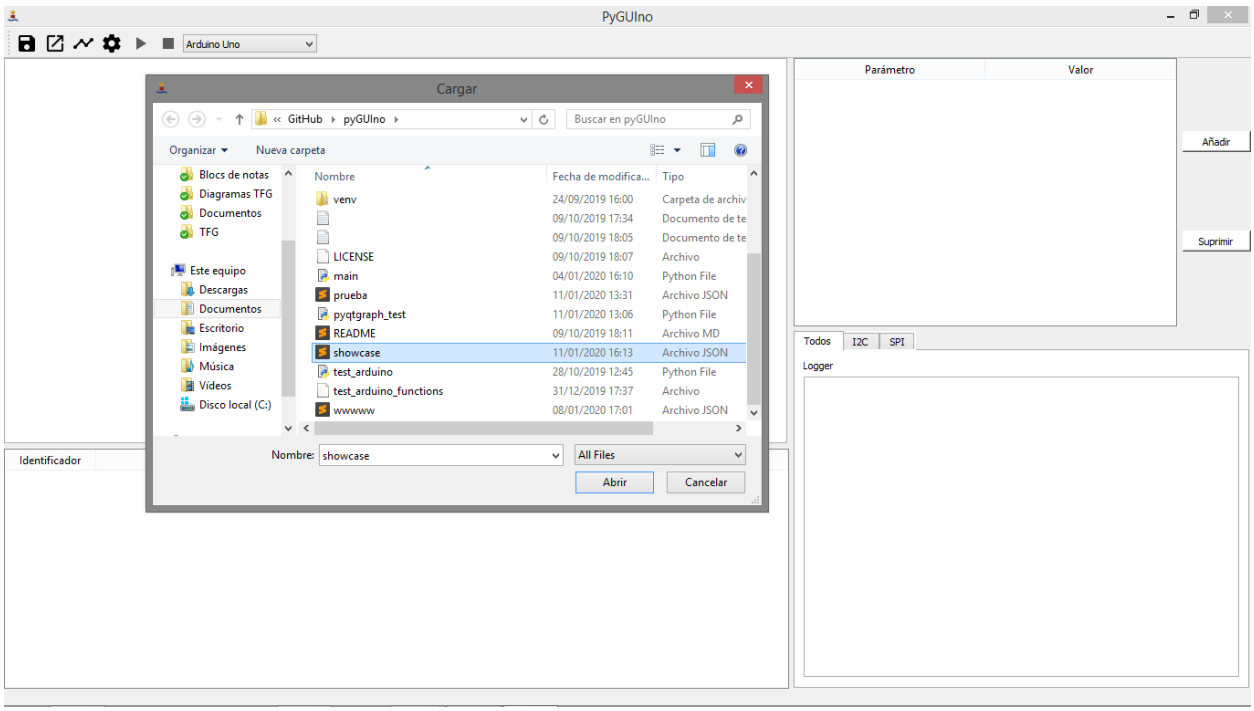


Figura 5-10 Carga de la configuración.

Una vez cargada la configuración se puede observar que en el área de gráficas se ha creado una pestaña y en la tabla de parámetros de usuario se han añadido algunas entradas de parámetros con sus valores.

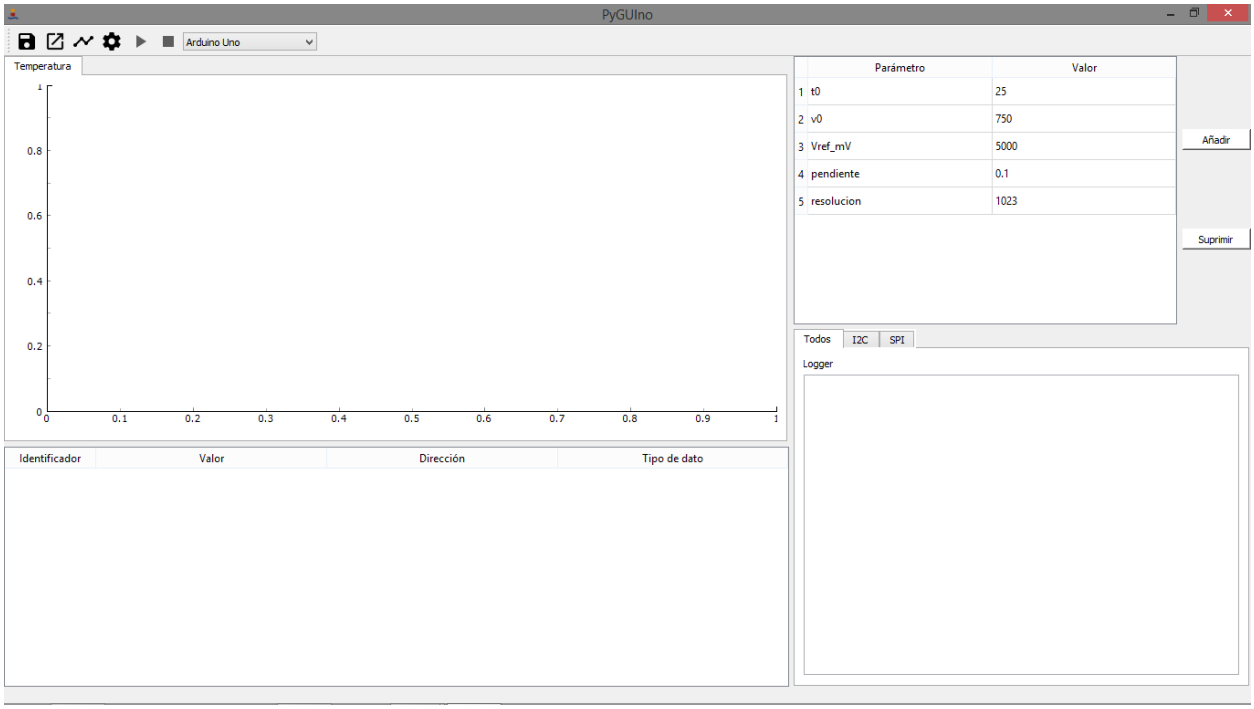


Figura 5-11Aspecto de la interfaz una vez cargada la configuración

Se procede a seleccionar el tipo de conexión con la placa, como se ha especificado anteriormente la conexión es mediante Bluetooth. Puede observarse una lista de los elementos disponibles para su selección.

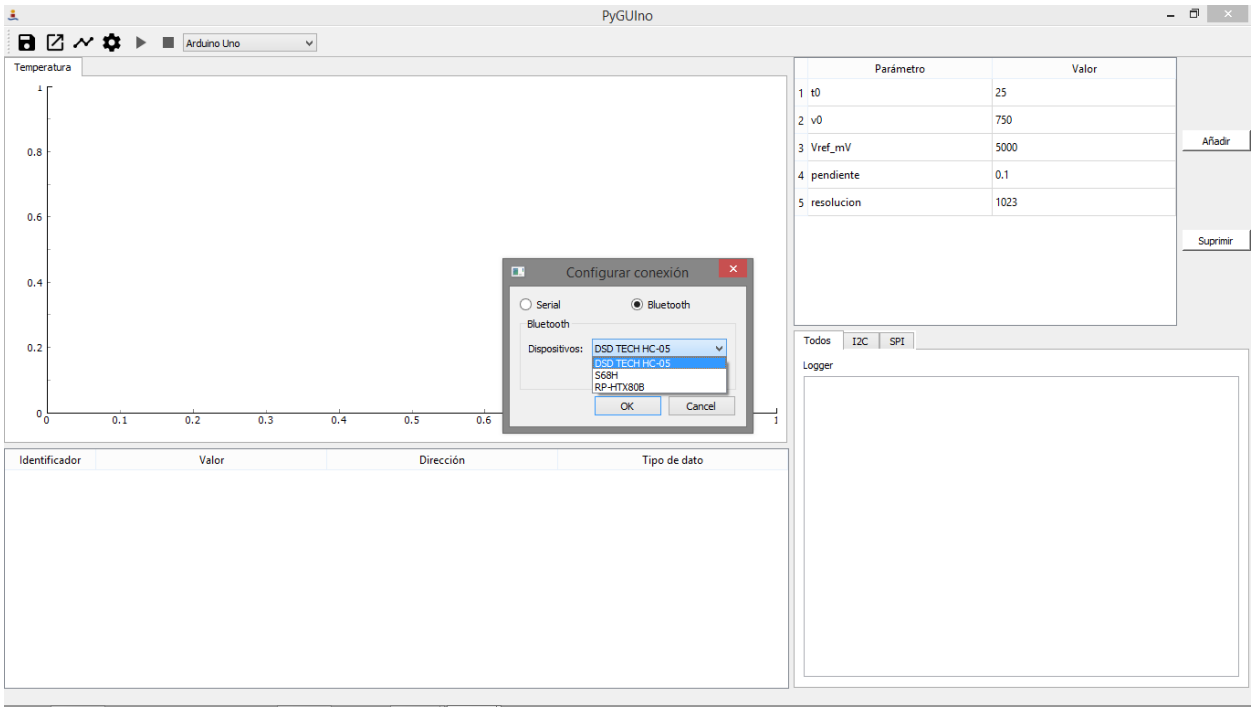


Figura 5-12 Selección de la conexión con Arduino mediante Bluetooth

Una vez establecida la conexión y habiendo pulsado el botón de inicio, comienza la recepción de datos y la representación de los mismo en la interfaz.

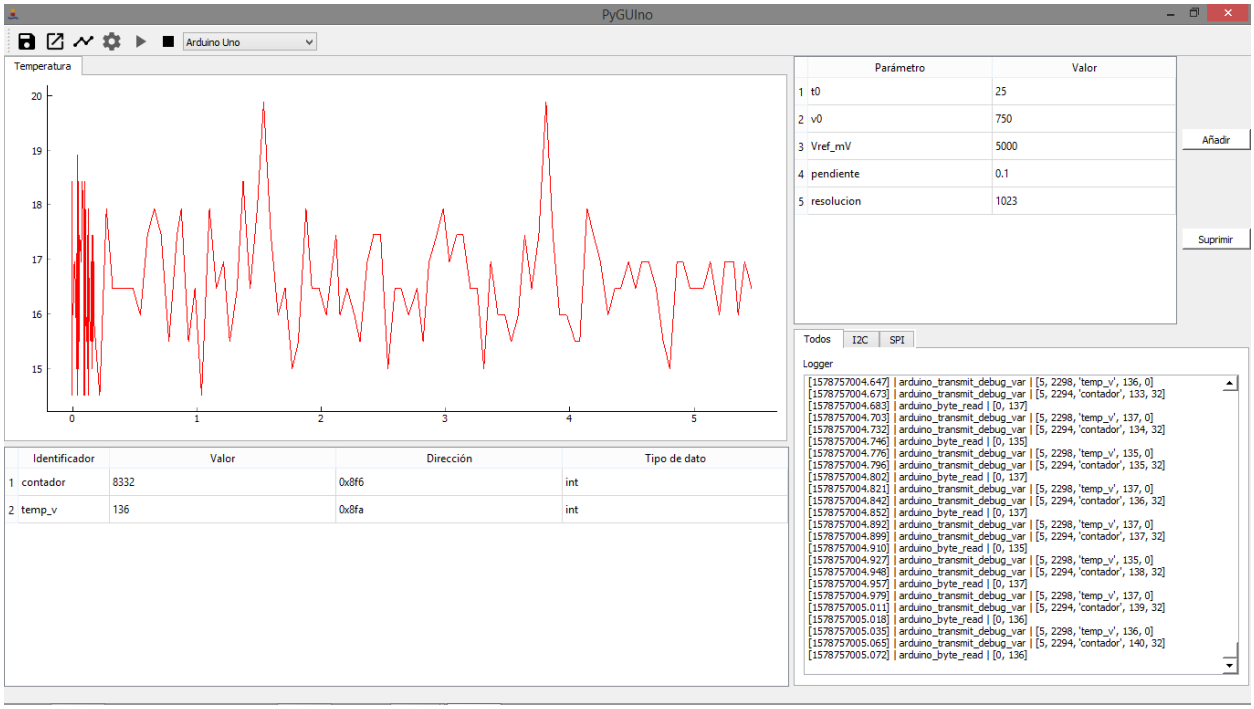


Figura 5-13 Interfaz en funcionamiento representando datos recibidos

Para mostrar que pueden añadirse gráficas adicionales con otros datos, se ha optado por crear una nueva pestaña que contenga la variable "contador" como gráfica, en la figura 5-14 podemos ver como se selecciona esta variable junto con los diálogos asociados.

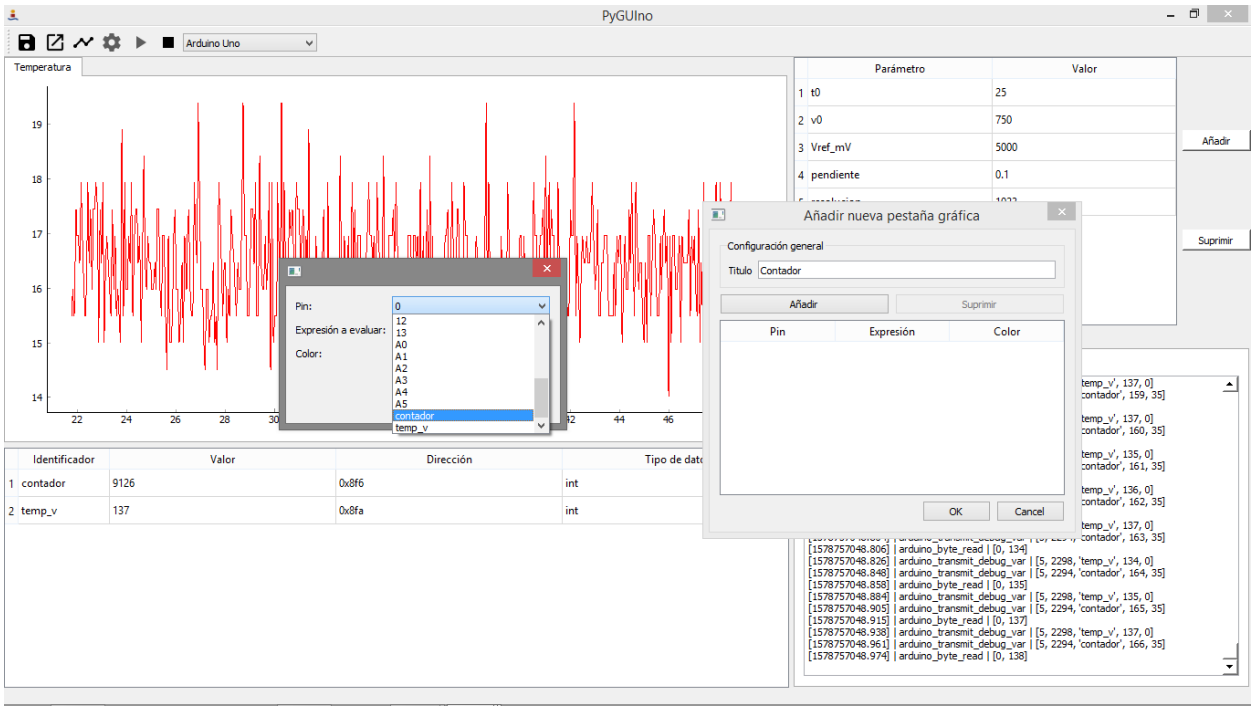


Figura 5-14 Creación de nueva pestaña para representar otras variables

Debido a que la variable elegida como ejemplo es “contador” su gráfica será como se ve en la figura siguiente:

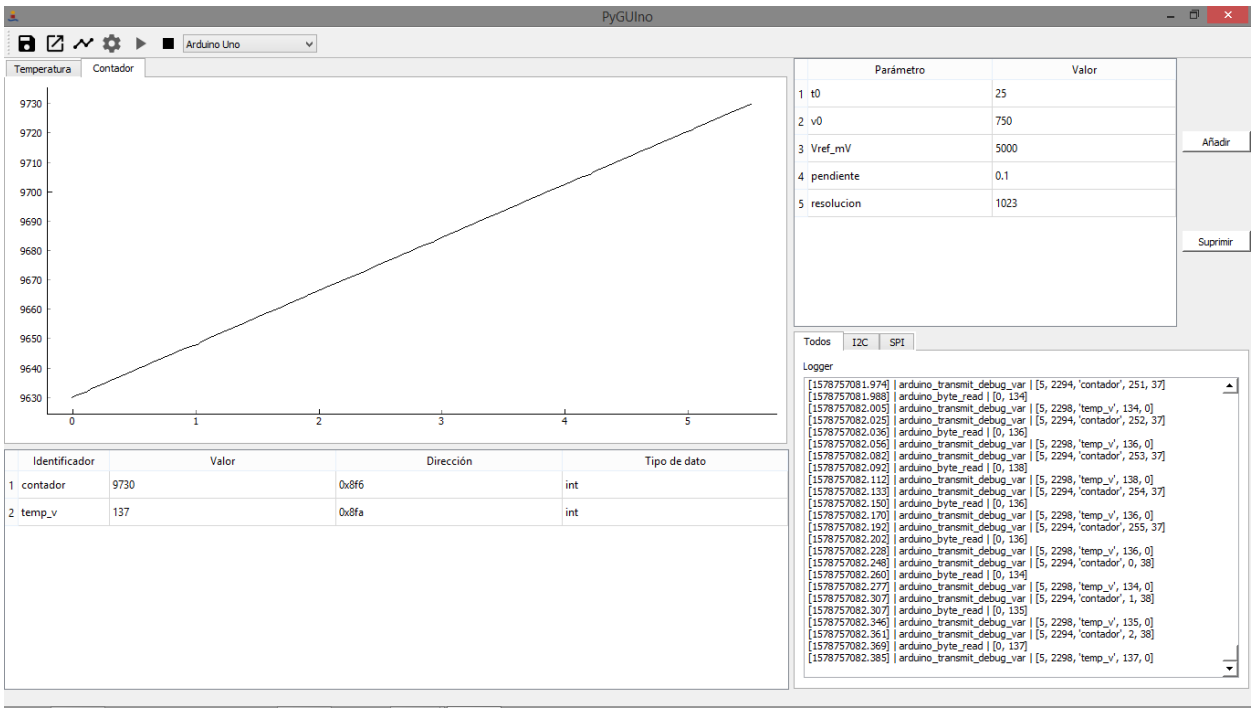


Figura 5-15 Gráfica asociada a la variable contador

6 CONCLUSIONES

Este Trabajo de Fin de Grado, se planteó como objetivo fundamental el desarrollo de una GUI para las placas de Arduino. Para abordar el objetivo planteado y desarrollar una GUI para Arduino se debe implementar una herramienta especifica de software que sea lo más polivalente posible. El incluir como objetivo la previsión de que la herramienta se pueda adaptar a situaciones diversas y a su ampliación se debe a que la comunidad que se ha creado a lo largo de los años realiza todo tipos de proyectos, fundamentalmente se trata de proyectos de aprendizaje de robótica, aunque podrían aplicarse a la domótica, control remoto, etcétera.

Durante la elaboración de la GUI se han perseguido unos objetivos que en estas conclusiones finales merece la pena analizar en cuanto a su situación de partida inicial, proceso de desarrollo y grado de cumplimiento.

El diseño inicial de la GUI se pensó basándose en una disposición de los elementos mediante una interfaz de documentos múltiples. Este tipo de disposición se basa en una ventana conteniendo otras ventanas. Tras analizar las posibilidades disponibles, se concluyó que era preferible un diseño basado en zonas fijas separadas con tamaño regulable ya que resultaba, a mi juicio, una mejor solución porque, para el usuario, trabajar con múltiples ventanas contenidas en una, podría resultar frustrante si estas deben ser continuamente acomodadas en tamaño y ubicación a sus preferencias.

Respecto al objetivo de diseño amigable que se propuso como objetivo al inicio del presente documento estimo que este ha sido logrado, no obstante, es un objetivo que permite un planteamiento de continua mejora. Por ello, todavía es posible realizar algunas pequeñas mejoras como añadir cierto espacio vacío a los lados de la ventana principal.

La experiencia de uso del programa es satisfactoria, se ha comprobado su funcionamiento y se han ido corrigiendo los errores que se han ido detectando a lo largo de su desarrollo. Se ha conseguido implementar la presentación de múltiples gráficas, así como múltiples curvas en una sola gráfica y los mensajes de entrada por parte de Arduino.

Si se valora globalmente el trabajo que se ha desarrollado, quizás el mayor reto de este proyecto ha sido la familiarización necesaria con las tecnologías que se han utilizado. Los lenguajes que se han usado no han sido el mayor reto, sino que este ha consistido en el estudio para su uso de algunas de las librerías y módulos. *PyQt5*, el módulo que se utilizó para crear la GUI apenas tiene documentación disponible, la mayoría está pendiente de hacerse. Este inconveniente se pudo solventar parcialmente con la lectura de documentación de la librería para C++. Además, se debe mencionar que esta versión es más reciente y sustituye a *PyQt4* cambiando completamente la sintaxis para señales y *slots* que proporciona la librería.

6.1 Posibilidades de ampliación y mejoras

Sobre las posibilidades de mejoras en la interfaz desarrollada, así como de ampliación de sus funcionalidades se podrían destacar las siguientes:

- **Control de la placa Arduino**: se podría añadir un pequeño editor de texto en el cual, el usuario escribiría un *script* en Python y también se proveería de una clase que representase a la placa, de forma que el papel de la placa de Arduino solamente sería para realizar lecturas y escrituras a través de los pines, usando estos últimos también para la comunicación con periféricos. En otras palabras, la lógica residiría en el ordenador y Arduino solo recibiría órdenes para interactuar con los pines y los periféricos conectados a este.
- **Representar los datos en el dominio frecuencial**: dar la posibilidad al usuario a la hora de añadir una gráfica, de escoger el dominio frecuencial mediante el uso de la transformada discreta de Fourier para la representación de los datos. Esto requeriría el uso de alguna implementación de la transformada como la que posee el módulo *numpy* y reescribir el método *update* de *PltAux* para su adecuación, además también se debería actualizar la clase *PlotForm* para permitir al usuario la selección del dominio temporal o frecuencial.

- **Añadir conexión mediante IP**: esta funcionalidad requeriría una implementación de los atributos y métodos que se expusieron en las tablas 4-1 y 4-2, también requeriría que el periférico utilizado por Arduino para la comunicación entregase los datos mediante un objeto de la clase *Stream*.
- **Pequeñas mejoras diversas**: lograr algunas mejoras de forma que la interfaz sea todavía más visualmente agradable para el usuario, añadir un botón para eliminar gráficas, lograr un código todavía más robusto replanteando los parámetros definidos por el usuario.

6.2 Valoración del autor

Personalmente, este trabajo me ha permitido el aprendizaje del lenguaje Python y algo de C++, los cuales no habían sido exigidos en el grado. Se ha podido trabajar con Python aprovechando la mayor parte de características que ofrece el lenguaje, paralelismo con hilos y comunicación entre ellos y el uso de los conceptos introducidos en programación orientada a objetos. Como lenguaje creo que tiene mucho potencial por el auge del *machine learning* y *big data*.

En lo referente al diseño de interfaces gráficas mediante el módulo utilizado, que como se ha dicho tiene algunos problemas, creo que ha sido a mi parecer un buen inicio en lo referente al desarrollo de GUIs. En cuanto a Arduino, el hecho de haber tenido que enviar datos internos de Arduino al ordenador me ayudó a comprender mejor sobre cómo poder trabajar con la memoria a bajo nivel.

En general tengo que valorar muy positivamente las aportaciones que a mi formación añade la elaboración de este Trabajo de Fin de Grado, la búsqueda de distintas fuentes de información, el tiempo dedicado a la construcción de la interfaz objeto de este trabajo y la redacción final del mismo. Todo esto ha supuesto un esfuerzo personal que sin duda incrementan el valor de la formación adquirida a lo largo de este grado.

7 ANEXO A: CÓDIGOS REALIZADOS

7.1 main.py

```
from utils.Core import PyGUIno


if __name__ == "__main__":
    pygu = PyGUIno()
    pygu.start_pygu()
```

7.2 Core.py

```
from PyQt5.QtWidgets import QWidget, QVBoxLayout, QSizePolicy,\
    QFileDialog, QToolBar, QAction, QStatusBar, QTabWidget, \
    QHBoxLayout, QApplication, QMainWindow, QComboBox, QSplitter
from PyQt5.QtCore import Qt, pyqtSignal, QThread
from PyQt5.QtGui import QIcon
from utils import CustomWidgets, Forms, Communication

import logging
import PyCmdMessenger
import os
import json
import sys
import traceback


class PyGUIno:
    def __init__(self):
        # Set up all the window related stuff
        self._app = QApplication(sys.argv)
        self._window = QMainWindow()
        size = self._app.primaryScreen().size()
        self._window.resize(int(size.width()*0.7),
                            int(size.height()*0.8))
        self._window.setObjectName("PyGUIno")
        self._window.setWindowTitle('PyGUIno')
        self._window.setWindowIcon(QIcon('resources\\assets\\etsi.png'))
        self._centralwidget = QWidget(self._window)
        self._window.setCentralWidget(self._centralwidget)
        self._centralwidget.setSizePolicy(QSizePolicy.Expanding,
QSizePolicy.Expanding)
        self.layout = QHBoxLayout()
        self._centralwidget.setLayout(self.layout)

        # Create the toolbar and the actions
        self.toolbar = QToolBar()
        save_action = QAction(QIcon('resources\\assets\\ic_save_3x.png'), 'Guardar',
self._window)
        save_action.setStatusTip('Guarda la configuración de las gráficas actuales')
        load_action =
QAction(QIcon('resources\\assets\\ic_open_in_new_18pt_3x.png'), 'Cargar',
self._window)
        load_action.setStatusTip('Cargar la configuración de gráficas')
        self.connect_menu =
QAction(QIcon('resources\\assets\\ic_settings_black_48dp.png'), 'Conectar',
self._window)
        self.connect_menu.setStatusTip('Configuración de la conexión a Arduino')
        self.start_action =
QAction(QIcon('resources\\assets\\ic_play_arrow_3x.png'), 'Comenzar', self._window)
```

```
        self.start_action.setEnabled(False)
        self.start_action.setStatusTip('Comienza la conexión con Arduino')
        self.stop_action = QAction(QIcon('resources\\assets\\ic_stop_3x.png'),
'Detener', self._window)
        self.stop_action.setEnabled(False)
        self.stop_action.setStatusTip('Detiene la conexión con Arduino')

        add_plot = QAction(QIcon('resources\\assets\\ic_timeline_48pt_3x.png'),
'Añadir gráfica', self._window)
        add_plot.setStatusTip('Añadir gráfica')

        board_selector = QComboBox()
        self.board_choices = []
        self.current_board = None
        schemas = os.listdir("resources\\schemas")

        self.widgetCoord = WidgetCoordinator()
        for schema in schemas:
            fd = open("resources\\schemas\\" + schema, 'r')
            data = json.load(fd)
            board_selector.addItem(data['name'])
            tmp = list()

            # añadir al dict los pines digitales
            for x in range(0, data['digital']):
                tmp.append(x)

            # añadir al dict los pines analogicos
            for x in range(data['digital'], data['digital'] + data['analog']):
                tmp.append("A"+str(x-data['digital']))

            self.board_choices.append((tmp, data["data_size"]))
            self.widgetCoord.data_size = data["data_size"]
            self.widgetCoord.pin_list = tmp
            self.current_board = tmp
            fd.close()

        # Signal handlers
        save_action.triggered.connect(self.save)
        load_action.triggered.connect(self.load)
        add_plot.triggered.connect(self.ini_graph_dialog)
        self.connect_menu.triggered.connect(self.comm_config)
        self.start_action.triggered.connect(self.start)
        self.stop_action.triggered.connect(self.stop)
        board_selector.currentIndexChanged.connect(self.switch_board)

        self.toolbar.addAction(save_action)
        self.toolbar.addAction(load_action)
        self.toolbar.addAction(add_plot)
        self.toolbar.addAction(self.connect_menu)
        self.toolbar.addAction(self.start_action)
        self.toolbar.addAction(self.stop_action)
        self.toolbar.addWidget(board_selector)

        self._window.addToolBar(self.toolbar)

        # Create status bar
        self.statusbar = QStatusBar(self._window)
        self._window.setStatusBar(self.statusbar)

        # Create Workspace area
        left_layout = QVBoxLayout()
        right_layout = QVBoxLayout()
        left_layout.setContentsMargins(0, 0, 0, 0)
        right_layout.setContentsMargins(0, 0, 0, 0)

        top_left_side = self.widgetCoord.plot_container
        top_right_side = self.widgetCoord.user_vars_table
        low_left_side = self.widgetCoord.debug_vars_widget
```

```
        low_right_side = self.widgetCoord.log_container

        # Prepare horizontal splitter and left and right vertical splitters
        horizontal_splitter = QSplitter(Qt.Horizontal)
        horizontal_splitter.setStyleSheet('background-color:rgb(239, 239, 239)')
        left_vertical_splitter = QSplitter(Qt.Vertical)
        left_vertical_splitter.setStyleSheet('background-color:rgb(239, 239, 239)')
        right_vertical_splitter = QSplitter(Qt.Vertical)
        right_vertical_splitter.setStyleSheet('background-color:rgb(239, 239, 239)')

        # First add left then add right
        left_vertical_splitter.addWidget(top_left_side)
        left_vertical_splitter.addWidget(low_left_side)
        left_layout.addWidget(left_vertical_splitter)

        # The same but on the right side
        right_vertical_splitter.addWidget(top_right_side)
        right_vertical_splitter.addWidget(low_right_side)
        right_layout.addWidget(right_vertical_splitter)

        # Finally add the vertical splitters to the horizontal splitter
        # And add it to the horizontal layout
        horizontal_splitter.addWidget(left_vertical_splitter)
        horizontal_splitter.addWidget(right_vertical_splitter)
        self.layout.addWidget(horizontal_splitter)
        self.layout.setContentsMargins(0, 0, 0, 0)

    def start_pygu(self):
        self._window.show()
        sys.exit(self._app.exec_())

    # Functions to trigger several menu actions
    def save(self):
        try:
            some_tuple = QFileDialog.getSaveFileName(parent=None, caption='Guardar',
                                                     directory='')
            abs_path = some_tuple[0]
            file = open(abs_path, 'w')
            data_to_save = json.dumps(self.widgetCoord.save(), sort_keys=True,
                                      indent=4, separators=(',', ': '))
            print(data_to_save)
            file.write(data_to_save)
            file.close()
        except Exception as e:
            print(e)

    def load(self):
        try:
            some_tuple = QFileDialog().getOpenFileName(parent=None,
caption='Cargar',
                                                       directory='')
            abs_path = some_tuple[0]
            file = open(abs_path, 'r')
            content = json.load(file)
            self.widgetCoord.load(content)
            file.close()
        except Exception as e:
            print(e)

    def comm_config(self):
        Forms.ConnectionForm(self.connect_menu, self.start_action, self.stop_action,
                             self.widgetCoord)

    def stop(self):
        self.widgetCoord.stop_comm()
        self.connect_menu.setEnabled(True)
        self.start_action.setEnabled(False)
        self.stop_action.setEnabled(False)

    def start(self):
```

```
          self.widgetCoord.start_comm()
          self.start_action.setEnabled(False)
          self.stop_action.setEnabled(True)

      def ini_graph_dialog(self):
          Forms.PlotForm(self.widgetCoord, self.current_board,
                         self.widgetCoord.resource_dict)

      def switch_board(self, index):
          self.current_board = self.board_choices[index][0]
          self.widgetCoord.pin_list = self.board_choices[index][0]
          self.widgetCoord.data_size = self.board_choices[index][1]


  class WidgetCoordinator:
      def __init__(self):
          # Class attributes
          self.plt_list = []
          self.data_size = None
          self.pin_list = None
          self.user_vars = dict()
          self.resource_dict = dict()
          self.commands = [
              ["ack_start", "s"],
              ["request_pin", "i"],
              ["arduino_transmit_pin_value", "ii"],
              ["request_debug_var_value", "iI"],
              ["answer_debug_var_value", "b*"],
              ["arduino_transmit_debug_var", "iIsb*"],
              ["arduino_byte_read", "Ib"],
              ["arduino_byte_write", "Ib*"],
              ["arduino_spi_transmit", "II"]
          ]
          self.arduino = None
          self.comm = None
          self.recv_thread = None

          # Create the widgets
          self.debug_vars_widget =
CustomWidgets.DebugVarsTable(resource_dict=self.resource_dict)
          self.user_vars_table = CustomWidgets.UserVarsTable(user_vars=self.user_vars)
          self.plot_container = QTabWidget()
          self.plot_container.setMinimumWidth(400)
          self.plot_container.setMinimumHeight(300)
          self.log_container = QTabWidget()
          self.plot_container.setStyleSheet('background-color:white')
          self.log_container.setStyleSheet('background-color:white')
          # Create loggers
          logs = ['Todos', 'I2C', 'SPI']
          for log_id in logs:
              log = logging.getLogger(log_id)
              log.setLevel(logging.INFO)
              self.create_logger_widget(log_id)

      # Communication related methods
      def set_comm(self, comm_args):
          try:
              if comm_args['type'] == 'Serial':
                  self.arduino = PyCmdMessenger.ArduinoBoard(comm_args['port'],
comm_args['baudrate'],
                                                             timeout=3.0,
settle_time=3.0,

int_bytes=self.data_size['int_bytes'],

float_bytes=self.data_size['float_bytes'],

double_bytes=self.data_size['double_bytes'],
```

```
long_bytes=self.data_size['long_bytes'])
                    self.comm = PyCmdMessenger.CmdMessenger(self.arduino, self.commands)
                elif comm_args['type'] == 'WiFi':
                    pass
                elif comm_args['type'] == 'Bluetooth':
                    self.arduino =
Communication.ArduinoBoardBluetooth(mac_addr=comm_args['mac_addr'],

int_bytes=self.data_size['int_bytes'],

float_bytes=self.data_size['float_bytes'],

double_bytes=self.data_size['double_bytes'],

long_bytes=self.data_size['long_bytes'])
                    self.comm = PyCmdMessenger.CmdMessenger(self.arduino, self.commands)
            except Exception as err:
                print(err.__class__)
                print(err)
                if self.recv_thread and self.recv_thread.isRunning():
                    self.recv_thread.stop()
                return False
            else:
                return True

        def stop_comm(self):
            if self.recv_thread:
                self.recv_thread.stop()
            self.comm.board.close()
            self.comm = None  # To force the garbage collector

        def start_comm(self):
            if self.comm:
                self.recv_thread = QThreadComm(self.comm)
                self.recv_thread.signal.connect(self.handle_new_data)
                self.recv_thread.start()
            else:
                Forms.ErrorMessageWrapper('Connection Error', 'There was an error
setting up the connection')

        def handle_new_data(self, msg):
            try:

                command = msg[0]
                payload = msg[1]
                ts = msg[2]

                input_log = logging.getLogger('Todos')
                input_log.info("[{:5.3f}] | {} | {} ".format(ts, command, payload))

                # only arduino_byte_read, arduino_byte_write y arduino_spi_transmit
                # needs to be send to the SPI and
                if command == "arduino_byte_read":
                    text = "Read from {} | Value: {}".format(payload[0],
hex(payload[1]))
                    input_log = logging.getLogger('I2C')
                    input_log.info(text)

                elif command == "arduino_byte_write":
                    if type(payload) == type([]):
                        payload.pop(0)
                        hex_payload = []
                        for x in payload:
                            hex_payload.append(hex(x))
                        text = "Writing value: {}".format(hex_payload)
                    else:
                        text = "Writing value: {}".format(hex(payload))
                    input_log = logging.getLogger('I2C')
                    input_log.info(text)
```

```
            elif command == "arduino_spi_transmit":
                text = "Tx value: 0x{:02X} | SPDR value:
0x{:02X}".format(payload[0], payload[1])
                input_log = logging.getLogger('SPI')
                input_log.info(text)

            elif command == self.commands[0][0]:  # ack_start
                print("Comunicación establecida {}".format(payload))

            elif command == self.commands[2][0]:  # arduino_transmit_pin_value
                # Obtener la key correspondiente para el numero y añadir valor
                # siempre sera un dict con los pines y sus identificadores
                index = payload[0]
                value = payload[1]
                key = self.pin_list[index]
                self.resource_dict[key] = value
                for widgetPlot in self.plt_list:
                    widgetPlot.new_data(data=(key, value), timestamp=ts)

            elif command == self.commands[5][0]:  # arduino_transmit_debug_var
                row_as_dict = dict()
                list_index = payload[0]
                type_list = [
                    (self.comm._recv_bool, 'bool'),
                    (self.comm._recv_byte, 'byte'),
                    (self.comm._recv_char, 'char'),
                    (self.comm._recv_float, 'float'),
                    (self.comm._recv_double, 'double'),  # 8 in Arduino Due
                    (self.comm._recv_int, 'int'),  # 4 on Due, Zero...
                    (self.comm._recv_long, 'long'),
                    (self.comm._recv_int, 'short'),
                    (self.comm._recv_unsigned_int, 'unsigned int'),  # 4 on Due
                    (self.comm._recv_unsigned_int, 'unsigned short'),
                    (self.comm._recv_unsigned_long, 'unsigned long')
                ]
                # type of data, memory address, human identifier
                try:
                    to_python_data = type_list[list_index][0]
                    row_as_dict['data_type'] = type_list[list_index][1]
                    row_as_dict['addr'] = payload[1]
                    row_as_dict['name'] = payload[2]
                    row_as_dict['value'] = to_python_data(bytes(payload[3:]))
                    self.debug_vars_widget.new_arduino_data(row_as_dict)
                    self.resource_dict[payload[2]] =
to_python_data(bytes(payload[3:]))
                    for widgetPlot in self.plt_list:
                        widgetPlot.new_data(data=[row_as_dict['name'],
row_as_dict['value']],
                                            timestamp=ts)
                except Exception as err:
                    Forms.ErrorMessageWrapper('Error en variable de depuración',
                                              'Ha habido un error en relacionado '
                                              'con variables de
depuracion\n{}'.format(err))
            except Exception as e:
                print(e)
                Forms.ErrorMessageWrapper(e.__class__, e)

        # Widget related methods
        def create_plot_widget(self, conf_ui, conf_plots):
            plot_widget = CustomWidgets.WidgetPlot(configuration_data=conf_ui,
                                                   config_plt_data=conf_plots,
                                                   resource_dict_ref=self.resource_dict,
                                                   user_dict_ref=self.user_vars)
            self.plt_list.append(plot_widget)
            self.plot_container.addTab(plot_widget, conf_ui['title'])

        def create_logger_widget(self, log_id):
```

```
          log_widget = CustomWidgets.CustomLogger(log_id=log_id)
          self.log_container.addTab(log_widget, log_id)

      # Load and save related methods
      def save(self):
          save_dict = dict()
          return_list = list()

          for widgetPlot in self.plt_list:
              return_list.append(widgetPlot.serialize())
          save_dict['user_dict'] = self.user_vars
          save_dict['widget_plot_list'] = return_list

          return save_dict

      def load(self, content):
          # Primero user_dict
          user_dict = content['user_dict']
          for key in user_dict.keys():
              self.user_vars_table.add_to_user_vars(key, user_dict[key])

          # Ahora debemos crear los WidgetPlot
          for widgetPlot in content['widget_plot_list']:
              # Preparamos los pltAux que contendra
              conf_plots = list()
              for plt_aux in widgetPlot['pltAux_list']:
                  conf_plots.append((plt_aux['update_variable'],
plt_aux['math_expression'],plt_aux['color']))
              # Preparamos el conf_ui
              conf_ui = dict()
              conf_ui['title'] = widgetPlot['title']
              self.create_plot_widget(conf_ui=conf_ui, conf_plots=conf_plots)


  class QThreadComm(QThread):

      signal = pyqtSignal(tuple)  # Mandatory to be defined at level class

      def __init__(self, comm):
          QThread.__init__(self, parent=None)
          self.comm = comm
          self.keep_running = True

      def run(self):
          msg = None
          while self.keep_running:
              try:
                  msg = self.comm.receive()
                  if msg:
                      self.signal.emit(msg)
              except Exception as err:
                  print('Error en la recepción de msg')
                  print(traceback.format_exception(None,  # <- type(e) by docs, but
ignored
                                                   err, err.__traceback__),
                        file=sys.stderr, flush=True)
                  if msg:
                      print(msg)
                  print(err)

      def stop(self):
          self.keep_running = False
```

7.3 Forms.py

```
  from PyQt5 import QtWidgets, QtCore
  from serial.tools.list_ports import comports
  import bluetooth
  import re

  import traceback
  import sys


  class ConnectionForm(QtWidgets.QDialog):
      def __init__(self, menu_action, start_action, stop_action, widget_coord):
          try:
              QtWidgets.QDialog.__init__(self)
              self.setWindowFlags(QtCore.Qt.CustomizeWindowHint                          |
QtCore.Qt.WindowCloseButtonHint)
              self.widgetCoord = widget_coord
              self.stop_action = stop_action
              self.menu_action = menu_action
              self.start_action = start_action
              self.selected = 'Serial'  # Default value
              self.devices = dict()  # Just used on bluetooth connections

              # Select type of connection
              conn_selection_layout = QtWidgets.QHBoxLayout()
              serial_button = QtWidgets.QRadioButton('Serial')
              serial_button.setChecked(True)
              bluetooth_button = QtWidgets.QRadioButton('Bluetooth')
              # internet_button = QtWidgets.QRadioButton('Internet')
              serial_button.toggled.connect(self.on_selected)
              bluetooth_button.toggled.connect(self.on_selected)
              # internet_button.toggled.connect(self.on_selected)
              conn_selection_layout.addWidget(serial_button)
              conn_selection_layout.addWidget(bluetooth_button)
              # conn_selection_layout.addWidget(internet_button)

              # Middle section for the type of connection
              self.formBox = QtWidgets.QGroupBox()
              self.qlabel1 = QtWidgets.QLabel(self.formBox)
              self.qlabel2 = QtWidgets.QLabel(self.formBox)
              self.i1 = QtWidgets.QLineEdit()
              self.i2 = QtWidgets.QLineEdit()
              self.transport = QtWidgets.QComboBox()
              self.pick_bluetooth = QtWidgets.QComboBox()

              # Accept/Reject
              button_box  =  QtWidgets.QDialogButtonBox(QtWidgets.QDialogButtonBox.Ok  |
QtWidgets.QDialogButtonBox.Cancel)
              button_box.rejected.connect(self.reject)
              button_box.accepted.connect(self.accept)

              # Add everything to the dialog layout
              self.formlayout = QtWidgets.QFormLayout()
              main_layout = QtWidgets.QVBoxLayout()
              self.setWindowTitle("Configurar conexión")
              self.formBox.setLayout(self.formlayout)
              main_layout.addLayout(conn_selection_layout)
              main_layout.addWidget(self.formBox)
              main_layout.addWidget(button_box)
              self.setLayout(main_layout)

              self.formBox.setTitle('Serial')
              self.qlabel1.setText('Puerto')
              self.qlabel2.setText('Tasa de baudios')
              self.formlayout.addRow(self.qlabel1, self.i1)
              self.formlayout.addRow(self.qlabel2, self.i2)
```

```
            self.exec_()
        except Exception as err:
            print(err)

    def on_selected(self):
        try:
            selected_button = self.sender()
            self.selected = selected_button.text()
            # Delete in formlayout
            while self.formlayout.count():
                item = self.formlayout.takeAt(0)
                if item.widget():
                    item.widget().deleteLater()

            # Now that the formLayout is empty we populate it
            self.formBox.setTitle(self.selected)
            if self.selected == 'Serial':
                self.qlabel1 = QtWidgets.QLabel(self.formBox)
                self.qlabel2 = QtWidgets.QLabel(self.formBox)
                self.qlabel1.setText('Port')
                self.qlabel2.setText('Baudrate')
                self.i1 = QtWidgets.QLineEdit()
                self.i2 = QtWidgets.QLineEdit()

                self.formlayout.addRow(self.qlabel1, self.i1)
                self.formlayout.addRow(self.qlabel2, self.i2)

            elif self.selected == 'Bluetooth':
                self.devices.clear()
                self.qlabel1 = QtWidgets.QLabel(self.formBox)
                self.qlabel1.setText("Dispositivos: ")
                self.pick_bluetooth = QtWidgets.QComboBox()
                available_devices      =      bluetooth.discover_devices(duration=5,
lookup_names=True)
                for addr, name in available_devices:
                    self.devices[name] = addr
                    self.pick_bluetooth.addItem(name)

                self.formlayout.addRow(self.qlabel1, self.pick_bluetooth)

            else:  # It can only be Internet option
                self.qlabel1 = QtWidgets.QLabel(self.formBox)
                self.qlabel2 = QtWidgets.QLabel(self.formBox)
                self.qlabel1.setText('IP')
                self.qlabel2.setText('Puerto')
                self.transport = QtWidgets.QComboBox()
                self.transport.addItem("TCP")
                self.transport.addItem("UDP")
                self.i1 = QtWidgets.QLineEdit()
                self.i2 = QtWidgets.QLineEdit()

                self.formlayout.addRow(self.qlabel1, self.i1)
                self.formlayout.addRow(self.transport)
                self.formlayout.addRow(self.qlabel2, self.i2)
        except Exception as err:
            print(err)

    def accept(self):
        comm_args = {'type': self.selected}
        if self.selected == 'Serial':
            comm_args['port'] = self.i1.text()
            comm_args['baudrate'] = self.i2.text()

        elif self.selected == 'Bluetooth':
            name = self.pick_bluetooth.currentText()
            comm_args['mac_addr'] = self.devices[name]

        elif self.selected == 'WiFi':
            comm_args['ip'] = self.i1.text()
            comm_args['protocol'] = self.transport.currentText()
```

```
            comm_args['port'] = self.i2.text()

        if self.validate_input(comm_args) and self.widgetCoord.set_comm(comm_args):
            # Enable dc
            self.menu_action.setEnabled(False)
            self.start_action.setEnabled(True)
            self.stop_action.setEnabled(False)
        else:
            ErrorMessageWrapper('Error de conexión', 'Se produjo un error en la
configuración de la conexión')
        super().accept()

    @staticmethod
    def validate_input(comm_args):
        if comm_args['type'] == 'Serial':
            for (port, desc, hardware_id) in comports(include_links=False):
                if comm_args['port'] == port:
                    try:
                        baud = int(comm_args['baudrate'])
                        if baud > 0:
                            return True
                        else:
                            return False
                    except ValueError:
                        return False
            return False

        # No need to check for bluetooth data
        elif comm_args['type'] == 'Bluetooth':
            return True
        elif comm_args['type'] == 'WiFi':
            valid_ip_address    =    "^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-
5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$"
            if re.match(valid_ip_address, comm_args['ip']):
                try:
                    port_number = int(comm_args['port'])
                    if 0 < port_number <= 65535:
                        return True
                except ValueError:
                    return False
            else:
                return False

        return False


class PlotForm(QtWidgets.QDialog):
    def __init__(self, widget_coord, pin_list, debug_vars):
        QtWidgets.QDialog.__init__(self)
        self.setWindowFlags(QtCore.Qt.CustomizeWindowHint                                    |
QtCore.Qt.WindowCloseButtonHint)
        self._color_dict = {'r': 'Rojo',
                            'g': 'Verde',
                            'b': 'Azul',
                            'c': 'Celeste',
                            'm': 'Cian',
                            'y': 'Amarillo',
                            'k': 'Negro',
                            'w': 'Blanco'}

        self.setWindowTitle("Añadir nueva pestaña gráfica")
        self.setMinimumWidth(400)
        self.setMinimumHeight(300)
        self.widgetCoord = widget_coord
        self.pin_list = list(pin_list)
        self.debug_vars = debug_vars
        self._selected_row = 0

        # Adding the debug vars to the list
```

```
          for key in sorted(debug_vars.keys()):
              if key not in self.pin_list:
                  self.pin_list.append(key)

          # form Box
          formlayout = QtWidgets.QFormLayout()
          formbox = QtWidgets.QGroupBox("Configuración general")
          self.lineedit1 = QtWidgets.QLineEdit()
          self.lineedit1.setStatusTip('Expression to evaluate when upon recieving a pin
'
                                      'value, leave blank to plot the just the pin
value')
          qlabel1 = QtWidgets.QLabel('Titulo')
          formlayout.addRow(qlabel1, self.lineedit1)

          add_button = QtWidgets.QPushButton("Añadir")
          self.delete_button = QtWidgets.QPushButton("Suprimir")
          self.delete_button.setEnabled(False)

          # Attach functions to buttons
          add_button.clicked.connect(self.on_click_add)
          self.delete_button.clicked.connect(self.delete_row)

          button_container = QtWidgets.QHBoxLayout()
          button_container.addWidget(add_button)
          button_container.addWidget(self.delete_button)

          # A table to display the given data given on the dialog
          # created by the add_button
          self.qt_table = QtWidgets.QTableWidget()
          self.qt_table.setColumnCount(3)
          self.qt_table.setRowCount(0)
          self.qt_table.setHorizontalHeaderLabels(['Pin', 'Expresión', 'Color'])
          self.qt_table.horizontalHeader().setSectionResizeMode(0,
QtWidgets.QHeaderView.Stretch)
          self.qt_table.horizontalHeader().setSectionResizeMode(1,
QtWidgets.QHeaderView.Stretch)
          self.qt_table.horizontalHeader().setSectionResizeMode(2,
QtWidgets.QHeaderView.Stretch)
          self.qt_table.setEditTriggers(QtWidgets.QAbstractItemView.NoEditTriggers)
          self.qt_table.itemClicked.connect(self.selected_row)

          # ButtonBox
          self.buttonBox = QtWidgets.QDialogButtonBox(QtWidgets.QDialogButtonBox.Ok |
QtWidgets.QDialogButtonBox.Cancel)
          self.buttonBox.rejected.connect(self.reject)
          self.buttonBox.accepted.connect(self.accept)

          # Adding everything to the vertical layout
          formbox.setLayout(formlayout)
          self.mainLayout = QtWidgets.QVBoxLayout()
          self.mainLayout.addWidget(formbox)
          self.mainLayout.addLayout(button_container)
          self.mainLayout.addWidget(self.qt_table)
          self.mainLayout.addWidget(self.buttonBox)
          self.setLayout(self.mainLayout)

          # Show the form
          self.exec_()

      def on_click_add(self):
          # open a PinEvalDialog
          self.PinEvalDialog(self, self.pin_list, color_dict=self._color_dict)

      def add_new_row(self, pin_selected, eval_string, color):
          self.delete_button.setEnabled(True)
          append_row = self.qt_table.rowCount()
          self.qt_table.insertRow(append_row)
          item_pin = QtWidgets.QTableWidgetItem(pin_selected)
          item_math = QtWidgets.QTableWidgetItem(eval_string)
```

```
          item_color = QtWidgets.QTableWidgetItem(color)
          self.qt_table.setItem(append_row, 0, item_pin)
          self.qt_table.setItem(append_row, 1, item_math)
          self.qt_table.setItem(append_row, 2, item_color)

      def delete_row(self):
          if self.selected_row:
              actual_row = self._selected_row - 1
              self.qt_table.removeRow(actual_row)
              self._selected_row = None
              # no hay columnas deshabilitamos delete
              if not self.qt_table.rowCount():
                  self.delete_button.setEnabled(False)

      def selected_row(self, item):
          self.qt_table.selectRow(item.row())
          self._selected_row = item.row() + 1  # Shenanigan

      def accept(self):
          n_rows = self.qt_table.rowCount()
          if n_rows:
              conf_plt_items = []
              for current_row in range(0, n_rows):
                  item_update = self.qt_table.item(current_row, 0)
                  item_math = self.qt_table.item(current_row, 1)
                  item_color = self.qt_table.item(current_row, 2)

                  if item_update and item_math and item_color:
                      key = item_update.text()
                      math_expression = item_math.text()
                      color_val = item_color.text()
                      color_key                                                    =
list(self._color_dict.keys())[list(self._color_dict.values()).index(color_val)]
                      conf_plt_items.append((key, math_expression, color_key))

              # Call the core to instantiate the PlotWidget
              general_config = dict()
              general_config['title'] = self.lineedit1.text()
              try:
                  self.widgetCoord.create_plot_widget(general_config, conf_plt_items)
              except Exception as err:
                  print(traceback.format_exception(None,   # <- type(e) by docs, but
ignored
                                                   err, err.__traceback__),
                        file=sys.stderr, flush=True)
          super().accept()

      class PinEvalDialog(QtWidgets.QDialog):
          def __init__(self, form_to_notify, plt_list, color_dict):
              QtWidgets.QDialog.__init__(self)
              self.setWindowFlags(QtCore.Qt.CustomizeWindowHint                    |
QtCore.Qt.WindowCloseButtonHint)
              self.setWindowTitle(" ")
              self.setFixedWidth(300)
              self.setFixedHeight(150)
              self.plt_form = form_to_notify

              qlabel1 = QtWidgets.QLabel("Pin:")
              qlabel2 = QtWidgets.QLabel("Expresión a evaluar:")
              qlabel3 = QtWidgets.QLabel("Color:")

              self.pin_selector = QtWidgets.QComboBox()
              for plt_item in plt_list:
                  self.pin_selector.addItem(str(plt_item))
              self.to_evaluate = QtWidgets.QLineEdit()
              self.pick_color = QtWidgets.QComboBox()
              for color in sorted(color_dict.values()):
                  self.pick_color.addItem(color)
```

```
            button_box = QtWidgets.QDialogButtonBox(QtWidgets.QDialogButtonBox.Ok |
QtWidgets.QDialogButtonBox.Cancel)
            button_box.rejected.connect(self.reject)
            button_box.accepted.connect(self.accept)

            main_layout = QtWidgets.QVBoxLayout()
            form_layout = QtWidgets.QFormLayout()
            form_layout.addRow(qlabel1, self.pin_selector)
            form_layout.addRow(qlabel2, self.to_evaluate)
            form_layout.addRow(qlabel3, self.pick_color)
            main_layout.addLayout(form_layout)
            main_layout.addWidget(button_box)
            self.setLayout(main_layout)

            self.exec_()

        def accept(self):
            # Get the user input and give it to form_to_notify
            pin_selected = self.pin_selector.currentText()
            eval_string = self.to_evaluate.text()
            color_selected = self.pick_color.currentText()
            # Handle adquired data to the PlotForm object
            self.plt_form.add_new_row(pin_selected, eval_string, color_selected)
            super().accept()


  class ErrorMessageWrapper:
      # A singleton approach to avoid bombing the user with
      # error windows
      instance = None

      def __init__(self, err_title, err_str):
          if not ErrorMessageWrapper.instance:
              ErrorMessageWrapper.instance                                        =
ErrorMessageWrapper.ErrorMessage(err_title, err_str)

      class ErrorMessage(QtWidgets.QErrorMessage):
          def __init__(self, err_title, err_str):
              QtWidgets.QErrorMessage.__init__(self)
              ErrorMessageWrapper.instance = self
              self.err_title = err_title
              self.err_str = err_str
              self.setWindowFlags(QtCore.Qt.CustomizeWindowHint                   |
QtCore.Qt.WindowCloseButtonHint)
              self.setWindowTitle(self.err_title)
              self.showMessage(self.err_str)
              self.exec_()

          def __del__(self):
              ErrorMessageWrapper.instance = None
```

7.4 CustomWidgets.py

```
  from PyQt5.QtWidgets import QWidget, QVBoxLayout, QLabel, QPlainTextEdit, \
      QPushButton, QTableWidget, QHBoxLayout, QTableWidgetItem, QDialog, QComboBox,
QLineEdit, QDialogButtonBox, \
      QFormLayout, QAbstractItemView, QHeaderView

  from PyQt5 import QtCore
  from utils.Forms import ErrorMessageWrapper
  import pyqtgraph as pg
  import time
  import logging


  class WidgetPlot(QWidget):
      def __init__(self, configuration_data, config_plt_data,
```

```
                    resource_dict_ref, user_dict_ref):
        QWidget.__init__(self)

        pg.setConfigOption('background', 'w')
        pg.setConfigOption('foreground', 'k')
        self.setLayout(QVBoxLayout())
        plot_widget = pg.PlotWidget(background='w')
        self.layout().addWidget(plot_widget)
        self.pltItem = plot_widget.getPlotItem()

        self.contained_plots = []
        self.resource_dict_ref = resource_dict_ref
        self.user_dict_ref = user_dict_ref
        # Set title and such from "meta"
        self.configuration_data = configuration_data

        for tmp in config_plt_data:
            plt_aux_tmp = self.PltAux(update_variable=tmp[0], math_expression=tmp[1],
color=tmp[2],
                                      plt_item=self.pltItem.plot())
            self.contained_plots.append(plt_aux_tmp)

    def new_data(self, data, timestamp):
        update_variable = data[0]
        value = data[1]

        for plt_aux in self.contained_plots:
            if plt_aux.update_variable == update_variable:
                tmp = self.resource_dict_ref.copy()
                tmp.update(self.user_dict_ref)
                # To avoid merging the dicts
                plt_aux.update(timestamp, value, tmp)

    def serialize(self):
        tmp = list()
        for plt_aux in self.contained_plots:
            tmp.append(plt_aux.serialize())
        return_dict = dict()
        return_dict['title'] = self.configuration_data['title']
        return_dict['pltAux_list'] = tmp

        return return_dict

    class PltAux:
        def __init__(self, update_variable, math_expression, color, plt_item):
            # Length must match
            self.limit = 500
            self.first_update = True
            self.t_ini = 0
            self.update_variable = update_variable
            self.curve = plt_item
            self.math_expression = math_expression
            self.color = color
            self.time_axe = []
            self.value_axe = []

        def update(self, ts, value, total_dict):
            if self.first_update:
                self.t_ini = time.time()
                self.first_update = False

            if len(self.time_axe) >= self.limit:
                self.time_axe.pop(0)
                self.value_axe.pop(0)
            # Eval and add to the axis
            try:
                if self.math_expression != "":
                    value = eval(self.math_expression.format_map(total_dict))
            except Exception as err:
```

```
                print('Error on eval, showing raw data')
                print(err)
                ErrorMessageWrapper(err.__class__, err)
            finally:
                self.time_axe.append(ts-self.t_ini)
                self.value_axe.append(value)
                self.curve.setData(self.time_axe, self.value_axe, pen=self.color)

        def serialize(self):
            save_dict = dict()
            save_dict['update_variable'] = self.update_variable
            save_dict['color'] = self.color
            if self.math_expression:
                save_dict['math_expression'] = self.math_expression
            else:
                save_dict['math_expression'] = ''

            return save_dict


  class CustomLogger(QWidget, logging.Handler):
      def __init__(self, log_id):
          QWidget.__init__(self)
          self.log_id = log_id

          logging.Handler.__init__(self)
          self.log = logging.getLogger(log_id)
          self.log.addHandler(self)

          base = QVBoxLayout()
          self.setLayout(base)

          self.loggerText = QPlainTextEdit()
          self.loggerText.setReadOnly(True)

          self.title = QLabel()
          self.title.setText('Logger')

          self.layout().addWidget(self.title)
          self.layout().addWidget(self.loggerText)

      def emit(self, record):
          msg = record.getMessage()
          self.loggerText.appendPlainText(msg)


  class DebugVarsTable(QWidget):
      def __init__(self, resource_dict):
          QWidget.__init__(self, parent=None)
          self.mainLayout = QHBoxLayout()
          self.mainLayout.setContentsMargins(0, 0, 0, 0)
          self.setStyleSheet('background-color:white')
          self.ArduinoTable = QTableWidget()  # VarName | Value | Adress | Type
          self._selected_row = None
          self.resource_dict = resource_dict

          self.ArduinoTable.setColumnCount(4)
          self.ArduinoTable.setRowCount(0)

          # Configure the table settings
          self.ArduinoTable.setHorizontalHeaderLabels(['Identificador', 'Valor',
                                                       'Dirección', 'Tipo de dato'])
          self.ArduinoTable.horizontalHeader().setSectionResizeMode(1,
QHeaderView.Stretch)
          self.ArduinoTable.horizontalHeader().setSectionResizeMode(1,
QHeaderView.Stretch)
          self.ArduinoTable.horizontalHeader().setSectionResizeMode(2,
QHeaderView.Stretch)
          self.ArduinoTable.horizontalHeader().setSectionResizeMode(3,
QHeaderView.Stretch)
```

```
          self.ArduinoTable.setEditTriggers(QAbstractItemView.NoEditTriggers)

          # Set up layout and add widgets
          self.setLayout(self.mainLayout)
          self.layout().addWidget(self.ArduinoTable)

      def new_arduino_data(self, data):
          append_row = self.ArduinoTable.rowCount()
          # Update table
          for row in range(0, append_row):
              item_tmp = self.ArduinoTable.item(row, 0)
              if item_tmp.text() == data['name']:
                  value_item = self.ArduinoTable.item(row, 1)
                  value_item.setText(str(data['value']))
                  return
          self.ArduinoTable.insertRow(append_row)
          var_name_item = QTableWidgetItem(str(data['name']))
          value_item = QTableWidgetItem(str(data['value']))
          addr_item = QTableWidgetItem(hex(data['addr']))
          data_type_item = QTableWidgetItem(str(data['data_type']))
          self.ArduinoTable.setItem(append_row, 0, var_name_item)
          self.ArduinoTable.setItem(append_row, 1, value_item)
          self.ArduinoTable.setItem(append_row, 2, addr_item)
          self.ArduinoTable.setItem(append_row, 3, data_type_item)

          # Update user_dict
          self.resource_dict[data['name']] = data['value']


  class UserVarsTable(QWidget):
      def __init__(self, user_vars):
          QWidget.__init__(self, parent=None)
          self.mainLayout = QHBoxLayout()
          self.mainLayout.setContentsMargins(0, 0, 0, 0)
          self.setStyleSheet('background-color:white')
          self.UserTable = QTableWidget()  # VarName | str(value) or math expression
          self._selected_row = None
          self.user_vars = user_vars

          self.UserTable.setColumnCount(2)
          self.UserTable.setRowCount(0)

          # Configure the table settings
          self.UserTable.setHorizontalHeaderLabels(['Parámetro', 'Valor'])
          self.UserTable.horizontalHeader().setSectionResizeMode(0,
QHeaderView.Stretch)
          self.UserTable.horizontalHeader().setSectionResizeMode(1,
QHeaderView.Stretch)
          self.UserTable.setEditTriggers(QAbstractItemView.NoEditTriggers)

          # Add event handlers
          self.UserTable.itemClicked.connect(self.selected_row)

          # Set of buttons for UserTable, use some sort of icons, +, crank, -
          button_container = QVBoxLayout()
          button_container.setContentsMargins(0, 0, 0, 0)
          self.add_button = QPushButton('Añadir')
          self.delete_button = QPushButton('Suprimir')

          self.add_button.clicked.connect(self.open_add_dialog)
          self.delete_button.clicked.connect(self.delete_from_user_vars)

          button_container.addWidget(self.add_button)
          button_container.addWidget(self.delete_button)

          # Set up layout and add widgets
          self.setLayout(self.mainLayout)
          self.layout().addWidget(self.UserTable)
          self.layout().addLayout(button_container)
```

```
    def open_add_dialog(self):
        try:
            self.AddUserVarMenu(user_table=self)
        except Exception as err:
            print(err)

    def delete_from_user_vars(self):
        if self._selected_row and bool(self.user_vars):
            actual_row = self._selected_row - 1
            key = self.UserTable.item(actual_row, 0).text()
            self.UserTable.removeRow(actual_row)
            del self.user_vars[key]
            self._selected_row = None

    def selected_row(self, item):
        self.UserTable.selectRow(item.row())
        self._selected_row = item.row() + 1  # Shenanigan
        print('Selected row: {}'.format(self._selected_row))

    def add_to_user_vars(self, key, value):
        if key not in self.user_vars.keys():
            # Check if it's an integer then a float
            try:
                float(value)
            except ValueError as err:
                # This aint a number, show error message
                ErrorMessageWrapper(err, 'No es un número')
            else:
                # Add to dictionary and update table
                self.user_vars[key] = value
                append_row = self.UserTable.rowCount()
                self.UserTable.insertRow(append_row)
                item_name = QTableWidgetItem(key)
                item_value = QTableWidgetItem(value)
                self.UserTable.setItem(append_row, 0, item_name)
                self.UserTable.setItem(append_row, 1, item_value)

    class AddUserVarMenu(QDialog):
        def __init__(self, user_table):
            QDialog.__init__(self)
            if not user_table:
                print('None de referencia')
            self.user_table = user_table
            self.setWindowFlags(QtCore.Qt.CustomizeWindowHint                    |
QtCore.Qt.WindowCloseButtonHint)
            self.setWindowTitle("Añadir parámetro")
            self.setFixedWidth(300)
            self.setFixedHeight(100)

            qlabel1 = QLabel("Nombre del parámetro:")
            qlabel2 = QLabel("Valor del parámetro:")

            self.var_name = QLineEdit()
            self.var_value = QLineEdit()

            button_box        =        QDialogButtonBox(QDialogButtonBox.Ok        |
QDialogButtonBox.Cancel)
            button_box.rejected.connect(self.reject)
            button_box.accepted.connect(self.accept)

            main_layout = QVBoxLayout()
            form_layout = QFormLayout()
            form_layout.addRow(qlabel1, self.var_name)
            form_layout.addRow(qlabel2, self.var_value)
            main_layout.addLayout(form_layout)
            main_layout.addWidget(button_box)
            self.setLayout(main_layout)

            self.show()
```

```
            self.exec_()

        def accept(self):
            key = self.var_name.text()
            value = self.var_value.text()
            if key and value and key != '' and value != '':
                self.user_table.add_to_user_vars(key, value)
            super().accept()
```

7.5 Communication.py

```
import bluetooth
import time


class ArduinoBoardBluetooth:
    def __init__(self, mac_addr,
                 timeout=3.0,
                 int_bytes=2,
                 long_bytes=4,
                 float_bytes=4,
                 double_bytes=4):

        self.mac_addr = mac_addr
        self.timeout = timeout

        self.int_bytes = int_bytes
        self.long_bytes = long_bytes
        self.float_bytes = float_bytes
        self.double_bytes = double_bytes

        # Create bluetooth socket to start connection
        self.bluetooth_socket = None
        self._is_connected = False
        self.open()

        # Limits for the board, rest of the __init__ is code from
        # Michael J. Harms code for PyCmdMessenger
        self.int_min = -2 ** (8 * self.int_bytes - 1)
        self.int_max = 2 ** (8 * self.int_bytes - 1) - 1

        self.unsigned_int_min = 0
        self.unsigned_int_max = 2 ** (8 * self.int_bytes) - 1

        self.long_min = -2 ** (8 * self.long_bytes - 1)
        self.long_max = 2 ** (8 * self.long_bytes - 1) - 1

        self.unsigned_long_min = 0
        self.unsigned_long_max = 2 ** (8 * self.long_bytes) - 1

        # Set to either IEEE 754 binary32 bit or binary64 bit
        if self.float_bytes == 4:
            self.float_min = -3.4028235E+38
            self.float_max = 3.4028235E+38
        elif self.float_bytes == 8:
            self.float_min = -1e308
            self.float_max = 1e308
        else:
            err = "float bytes should be 4 (32 bit) or 8 (64 bit)"
            raise ValueError(err)

        if self.double_bytes == 4:
            self.double_min = -3.4028235E+38
            self.double_max = 3.4028235E+38
        elif self.double_bytes == 8:
            self.double_min = -1e308
            self.double_max = 1e308
```

```
        else:
            err = "double bytes should be 4 (32 bit) or 8 (64 bit)"
            raise ValueError(err)

        # ------------------------------------------------------------------------
        # Create a self.XXX_type for each type based on its byte number. This
        # type can then be passed into struct.pack and struct.unpack calls to
        # properly format the bytes strings.
        # ------------------------------------------------------------------------

        INTEGER_TYPE = {2: "<h", 4: "<i", 8: "<l"}
        UNSIGNED_INTEGER_TYPE = {2: "<H", 4: "<I", 8: "<L"}
        FLOAT_TYPE = {4: "<f", 8: "<d"}

        try:
            self.int_type = INTEGER_TYPE[self.int_bytes]
            self.unsigned_int_type = UNSIGNED_INTEGER_TYPE[self.int_bytes]
        except KeyError:
            keys = list(INTEGER_TYPE.keys())
            keys.sort()

            err = "integer bytes must be one of {}".format(keys())
            raise ValueError(err)

        try:
            self.long_type = INTEGER_TYPE[self.long_bytes]
            self.unsigned_long_type = UNSIGNED_INTEGER_TYPE[self.long_bytes]
        except KeyError:
            keys = list(INTEGER_TYPE.keys())
            keys.sort()

            err = "long bytes must be one of {}".format(keys())
            raise ValueError(err)

        try:
            self.float_type = FLOAT_TYPE[self.float_bytes]
            self.double_type = FLOAT_TYPE[self.double_bytes]
        except KeyError:
            keys = list(self.FLOAT_TYPE.keys())
            keys.sort()

            err = "float and double bytes must be one of {}".format(keys())
            raise ValueError(err)
        pass

    def open(self):
        port = 1
        try:
            self.bluetooth_socket = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
            self.bluetooth_socket.settimeout(self.timeout)
            self.bluetooth_socket.connect((self.mac_addr, port))
            time.sleep(2.0)
        except bluetooth.btcommon.BluetoothError as err:
            # Error handler
            print(err)
        else:
            self._is_connected = True

    def read(self):
        # Must return a byte as ArduinoBoard does
        return self.bluetooth_socket.recv(1)

    def readline(self):
        # Must return a whole line
        # This method is never used by CmdMessenger class though
        try:
            raw_msg = self.bluetooth_socket.recv(1024)
            decoded_msg = raw_msg.decode('utf-8')[:-1]
        except UnicodeDecodeError as err:
            print(err)
```

```
            return None
        else:
            return decoded_msg

    def write(self, msg):
        self.bluetooth_socket.send(msg)

    def close(self):
        if self._is_connected:
            self.bluetooth_socket.close()
            self._is_connected = False

    @property
    def connected(self):
        return self._is_connected
```

7.6 PyGUIno.h

```
  #include <Arduino.h>
  #include "CmdMessenger.h"

  //Mensajes definidos entre el ordenador y Arduino
  enum available_commands{
  ack_start,
      request_pin, //PC will request a pin value, it will send it the value
      arduino_transmit_pin_value, //Arduino will transmit the pin int and the value
      request_debug_var_value, //PC will handle memory addr and tpye of a debug var
value
      answer_debug_var_value, //Answer to the previous msg
      arduino_transmit_debug_var, //Arduino will send to the pc a debug var
      // Solo nos interesa avisar de bytes entrantes y salientes
      arduino_byte_read_i2c,
      //Avisa al ordenador de que se ha leido un byte junto con la direccion que lo
envio
      // addr, byte
      arduino_byte_write_i2c,
      //Avisa al ordenador de que se envia uno o más bytes
      // addr, bytes
      arduino_spi_transfer
      //Debe serán o 16 bits o 32, con enviar dos uint16_t valdría
      // primero lo que Arduino enviará luego lo que Arduino ha recibido
  };


  typedef enum{
      bool_var,
      byte_var,
      char_var,
      float_var,
      double_var,
      int_var,
      long_var,
      short_var,
      u_int_var,
      u_short_var,
      u_long_var
  } data_type;


  void attach_callbacks(CmdMessenger &cmd);

  void on_request_pin(); // The pc will request a pin value

  void send_debug_var_value();
  void transmit_pin_value(int pin);
```

```
void add_update_debug_var(int data, bool var, const char name[]);
void add_update_debug_var(int data, byte var, const char name[]);
void add_update_debug_var(int data, char var, const char name[]);
void add_update_debug_var(int data, float var, const char name[]);
void add_update_debug_var(int data, double var, const char name[]);
void add_update_debug_var(int data, int var, const char name[]);
void add_update_debug_var(int data, long var, const char name[]);
void add_update_debug_var(int data, short var, const char name[]);
void add_update_debug_var(int data, unsigned char var, const char name[]);
void add_update_debug_var(int data, unsigned int var, const char name[]);
void add_update_debug_var(int data, unsigned short var, const char name[]);
void add_update_debug_var(int data, unsigned long var, const char name[]);
```

7.7 PyGUIno.cpp

```
/**
* Author: frarambra
* Website: https://github.com/frarambra
*
**/

#include "PyGUIno.h"
#include "CmdMessenger.h"
#include "Arduino.h"

CmdMessenger *pc_side;

//Tested, pc_side keeps the cmd address after function cast
void attach_callbacks(CmdMessenger &cmd){
    pc_side = &cmd;
    pc_side->attach(request_pin, on_request_pin);
    pc_side->attach(request_debug_var_value, send_debug_var_value);
    delay(5000);
    pc_side->sendCmd(ack_start, "Arduino has booted");
}

void transmit_pin_value(int pin){
    pc_side->sendCmdStart(arduino_transmit_pin_value);
    pc_side->sendCmdBinArg<int>(pin);
    int value = (pin < A0) ? digitalRead(pin) : analogRead(pin);
    pc_side->sendCmdBinArg<int>(value);
    pc_side->sendCmdEnd();
}

void on_request_pin(){
    int pin = pc_side->readBinArg<int>();
    int value = (pin < A0) ? digitalRead(pin) : analogRead(pin);
    pc_side->sendBinCmd(request_pin, value);
}

void actual_add_update(int data, int offset, void *addr, const char name[]){
    pc_side->sendCmdStart(arduino_transmit_debug_var);
    pc_side->sendCmdBinArg<int>(data); //Data type
    unsigned int tmp = (unsigned int) addr;
    pc_side->sendCmdBinArg<unsigned int>(tmp); //Addr
    pc_side->sendCmdArg(name); //Name
    for(int i=0; i< offset; i++){
        byte *tmp = (byte *)addr;
        pc_side->sendCmdBinArg<byte>(*tmp);// Bunch of bytes containing the value
        addr++;
    }
    pc_side->sendCmdEnd();
}

void add_update_debug_var(int data, bool var, const char name[]){
    actual_add_update(data, sizeof(var), &var, name);
}
```

```
void add_update_debug_var(int data, byte var, const char name[]){
    actual_add_update(data, sizeof(var), &var, name);
}

void add_update_debug_var(int data, char var, const char name[]){
    actual_add_update(data, sizeof(var), &var, name);
}

void add_update_debug_var(int data, float var, const char name[]){
    actual_add_update(data, sizeof(var), &var, name);
}

void add_update_debug_var(int data, double var, const char name[]){
    actual_add_update(data, sizeof(var), &var, name);
}

void add_update_debug_var(int data, int var, const char name[]){
    actual_add_update(data, sizeof(var), &var, name);
}

void add_update_debug_var(int data, long var, const char name[]){
    actual_add_update(data, sizeof(var), &var, name);
}

void add_update_debug_var(int data, short var, const char name[]){
    actual_add_update(data, sizeof(var), &var, name);
}

void add_update_debug_var(int data, unsigned int var, const char name[]){
    actual_add_update(data, sizeof(var), &var, name);
}

void add_update_debug_var(int data, unsigned short var, const char name[]){
    actual_add_update(data, sizeof(var), &var, name);
}

void add_update_debug_var(int data, unsigned long var, const char name[]){
    actual_add_update(data, sizeof(var), &var, name);
}

void send_debug_var_value(){
    //We expect the variable address and type
    int offset = pc_side->readBinArg<int>();
    unsigned int tmp = pc_side->readBinArg<unsigned int>();
    byte *addr = (byte *) tmp;
    pc_side->sendCmdStart(answer_debug_var_value);
    for(int i=0; i< offset; i++){
        pc_side->sendCmdBinArg<byte>(*addr);
        addr++;
    }
    pc_side->sendCmdEnd();
    //TODO: testing the function
}
```

7.8 PyGUInoWire.h

```
#include <Wire.h>
#include "CmdMessenger.h"
#include "Arduino.h"
#include "PyGUIno.h"



//La clase no heredará de Wire, sino que esta clase será utilizada
//Como si fuera un atributo más
```

```
  class PyGUInoWire {
      private:
          //Atributos
          CmdMessenger *cmd;


      public:
          //Atributos
          byte last_read_byte;
          uint16_t read_addr;
          uint16_t write_addr = 0;

          //Metodos

          PyGUInoWire(CmdMessenger &cmd); // Si hay una instacia de CmdMessenger la
usaremos
          PyGUInoWire(); // En caso de no haber instancia la creamos
          void begin(int address); //OK
          void begin();
          byte requestFrom(int address, int quantity);//OK
          byte requestFrom(int address, int quantity, bool stop);//OK
          void beginTransmission(int address);//OK
          byte endTransmission();         // 0 ok; 1 data too long to fit in transmit
buffer
          byte endTransmission(bool stop);// 2 recieved NACK on transmit of addres, 3
recv NACK on transmit data
                                          // 4 other error
          byte write(byte value);//OK
          byte write(const char name[]);//OK
          byte write(byte data[], int length);//OK
          byte available();//OK
          byte read();//OK
          void SetClock(int clockFrequency);
          //void onReceive(void (*function)(int));
          void onRequest(void (*function)(void));//ok
  };
```

7.9 PyGUInoWire.cpp

```
  #include "Arduino.h"
  #include "CmdMessenger.h"
  #include "PyGUInoWire.h"


  PyGUInoWire::PyGUInoWire(CmdMessenger &cmd){
      this->cmd = &cmd;
      this->cmd->sendCmd(ack_start, "Arduino has booted");

  }

  // Metodos que me asignan los valores de los atributos
  // write_addr y read_addr
  void PyGUInoWire::beginTransmission(int address){
      write_addr = (uint16_t) address;
      Wire.beginTransmission(address);
  }

  byte PyGUInoWire::requestFrom(int address, int quantity){
      read_addr = (uint16_t) address;
      return Wire.requestFrom(address, quantity);
  }

  byte PyGUInoWire::requestFrom(int address, int quantity, bool stop){
      read_addr = (uint16_t) address;
      return Wire.requestFrom(address, quantity, stop);
  }
```

```
byte PyGUInoWire::write(byte value){
    cmd->sendCmdStart(arduino_byte_write_i2c);
    cmd->sendCmdBinArg<uint16_t>(write_addr);
    cmd->sendCmdBinArg<byte>(value);
    cmd->sendCmdEnd();
    return Wire.write(value);
}

byte PyGUInoWire::write(const char data[]){

    int i = 0;
    cmd->sendCmdStart(arduino_byte_write_i2c);
    cmd->sendCmdBinArg<uint16_t>(write_addr);
    while(data[i]!='\0'){
        cmd->sendCmdBinArg<byte>(data[i]);
        i++;
    }
    cmd->sendCmdEnd();

    return Wire.write(data);
}

byte PyGUInoWire::write(byte data[], int length){

    int i = 0;
    cmd->sendCmdStart(arduino_byte_write_i2c);
    cmd->sendCmdBinArg<uint16_t>(write_addr);
    while(i<length){
        cmd->sendCmdBinArg<byte>(data[i]);
        i++;
    }
    cmd->sendCmdEnd();

    return Wire.write(data, length);
}

byte PyGUInoWire::read(){

    //Solo tenemos que devolver el byte leido, lo ideal sería tambien dar
    //La direccion del periferico que lo hace
    cmd->sendCmdStart(arduino_byte_read_i2c);
    cmd->sendCmdBinArg<uint16_t>(write_addr);//addr
    byte tmp = Wire.read();
    cmd->sendCmdBinArg<byte>(tmp);        //value
    cmd->sendCmdEnd();
    return tmp;
}

//Resto del archivo es basura para hacer patron fachada

void PyGUInoWire::begin(int address){
    address ? Wire.begin(address) : Wire.begin();
}

void PyGUInoWire::begin(){
    Wire.begin();
}

byte PyGUInoWire::endTransmission(){
    return Wire.endTransmission();
}

byte PyGUInoWire::endTransmission(bool stop){
    return Wire.endTransmission(stop);
}

byte PyGUInoWire::available(){
    return Wire.available();
}
```

```
void PyGUInoWire::SetClock(int clockFrequency){
    Wire.setClock(clockFrequency);
}

//void PyGUInoWire::onReceive(void (*function)(int)){
//    Wire.onReceive((*function)(int));
//}

void PyGUInoWire::onRequest(void (*function)(void)){
    Wire.onRequest(function);
}
```

7.10 PyGUInoSPI.h

```
#include <Arduino.h>
#include <SPI.h>
#include "CmdMessenger.h"


class PyGUInoSPI {
    private:
        CmdMessenger *cmd;

    public:
        PyGUInoSPI(CmdMessenger &cmd);
        void begin();
        void end();
        void beginTransaction(SPISettings settings);
        void endTransaction();
        uint8_t transfer(uint8_t value);//
        void transfer(void *buff, size_t count); //Parece que deja en buff el dato
        uint16_t transfer16(uint16_t value);
        byte read();
        void usingInterrupt(int interruptNumber);
        void setClockDivider(int clock_divider);

};
```

7.11 PyGUInoSPI.cpp

```
#include "Arduino.h"
#include "pyGUInoSPI.h"
#include "CmdMessenger.h"


PyGUInoSPI::PyGUInoSPI(CmdMessenger &cmd){
    this->cmd = &cmd;
}

void PyGUInoSPI::begin(){
    SPI.begin();
}

void PyGUInoSPI::end(){
    SPI.end();
}

void PyGUInoSPI::beginTransaction(SPISettings settings){
    SPI.beginTransaction(settings);
}

void PyGUInoSPI::endTransaction(){
    SPI.endTransaction();
}
```

```
uint8_t PyGUInoSPI::transfer(uint8_t value){
    uint8_t retorno;
    uint16_t tmp;

    cmd->sendCmdStart(arduino_spi_transfer);
    tmp = (uint16_t) value;
    cmd->sendCmdBinArg<uint16_t>(tmp);
    retorno = SPI.transfer(value);
    tmp = (uint16_t) retorno;
    cmd->sendCmdBinArg<uint16_t>(tmp);
    cmd->sendCmdEnd();

    return retorno;
}

uint16_t PyGUInoSPI::transfer16(uint16_t value){

    uint16_t tmp;
    cmd->sendCmdStart(arduino_spi_transfer);
    cmd->sendCmdBinArg<uint16_t>(value);
    tmp = SPI.transfer(value);
    cmd->sendCmdBinArg<uint16_t>(tmp);
    cmd->sendCmdEnd();

    return tmp;
}

byte PyGUInoSPI::read(){
    uint16_t tmp = (uint16_t) 0;
    cmd->sendCmdStart(arduino_spi_transfer);
    cmd->sendCmdBinArg<uint16_t>(tmp); // Tx
    tmp = (uint16_t) SPDR;
    cmd->sendCmdBinArg<uint16_t>(tmp);
    cmd->sendCmdEnd();

    return tmp;
}



void PyGUInoSPI::transfer(void *buff, size_t count){
    //A implementar
    SPI.transfer(buff, count);
}


void PyGUInoSPI::usingInterrupt(int interruptNumber){
    SPI.usingInterrupt(interruptNumber);
}

void PyGUInoSPI::setClockDivider(int clock_divider){
    SPI.setClockDivider(clock_divider);
}
```

8 ANEXO B: MANUAL DE USO DEL SOFTWARE

8.1 Instalación del software necesario

Para poder utilizar el software se requiere la instalación de la versión de Python 3.5 o superior, esta versión puede ser descargada de https://www.python.org/downloads/

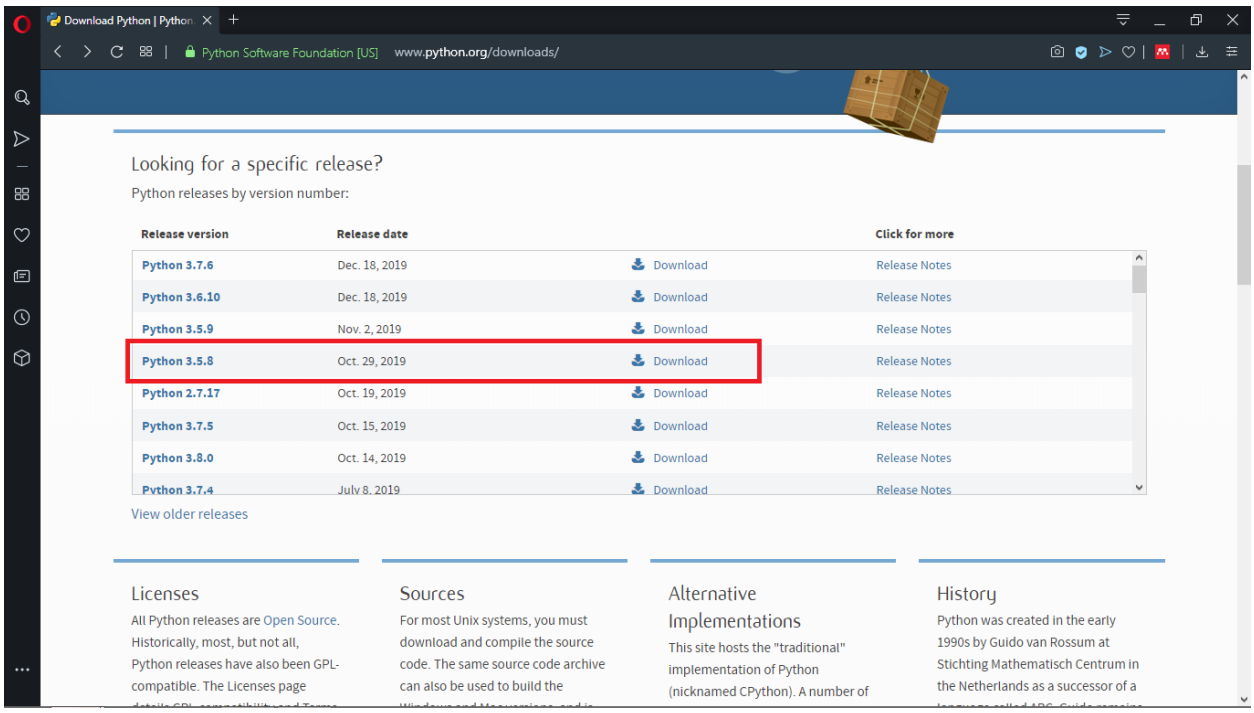


Figura 8-1 Sección de descargas de http://python.org

Una vez descargado e instalado podremos ejecutar archivos con la extensión *.py*. También debería estar disponible el comando *pip*, para la instalación de módulos de Python.

El software requiere del uso de los módulos PyQt5, pyBluez y PyCmdMessenger para el correcto funcionamiento. PyQt5 en especial requiere el uso de la versión 5.13.0. Para instalar los módulos deberá ejecutar las siguientes ordenes ejecutadas desde el terminal o en el caso de Windows desde el *cmd* o Windows Powershell con privilegios de administrador:

- *pip install PyQt5==5.13.0*
- *pip install PyCmdMessenger*

Estos dos comandos permitirán la instalación de 2 de los 3 módulos necesarios. Respecto al módulo bluetooth en caso de que el sistema operativo en el que se utlizará fuera MacOs o una distribución Linux solo debe ejecutar el comando:

- *pip install pybluez*

En el caso de Windows, el proceso de instalación del este módulo *pybluez* es algo más complejo pues requiere el uso del compilador adecuado para C++ que Microsoft incluye como parte de su "Visual Studio Build Tools 2019". Para su instalación diríjase a https://visualstudio.microsoft.com/es/downloads/ y descargue "Build Tools para Visual Studio 2019", en la figura 8-2 se observar donde se debe realizar la descarga.

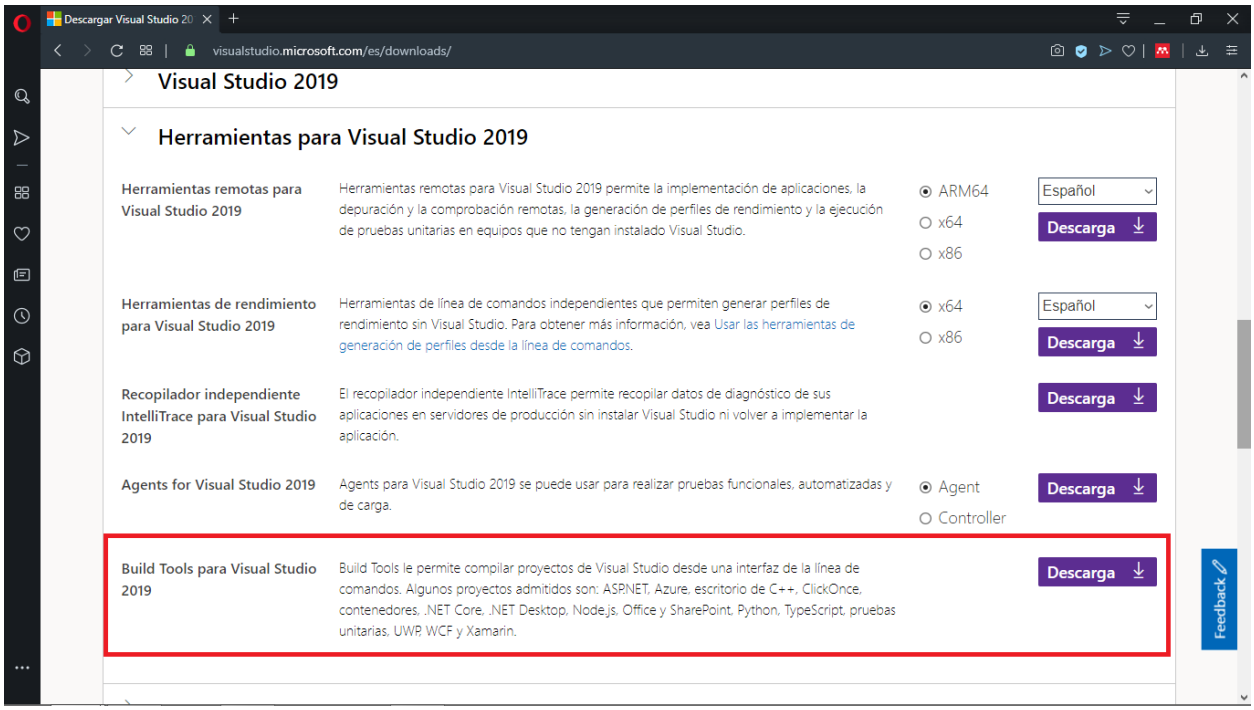


Figura 8-2 Zona de descarga de Microsoft

Tras descargar el archivo e instalarlo, debemos a su vez seleccionar el compilador de C++. Para ello vaya a la sección "Componentes individuales" y seleccione "Herramientas de compilación de MSVC v142 – VS 2019 C++ para x64/x86 (v14.23)" y proceda a su instalación. En caso de no encontrar tal entrada, cualquier tipo de compilador para x64 o x86 con versión superior o igual a la 14 debería ser suficiente. En la figura 8-3 se puede observar parte de las opciones que Microsoft ofrece.

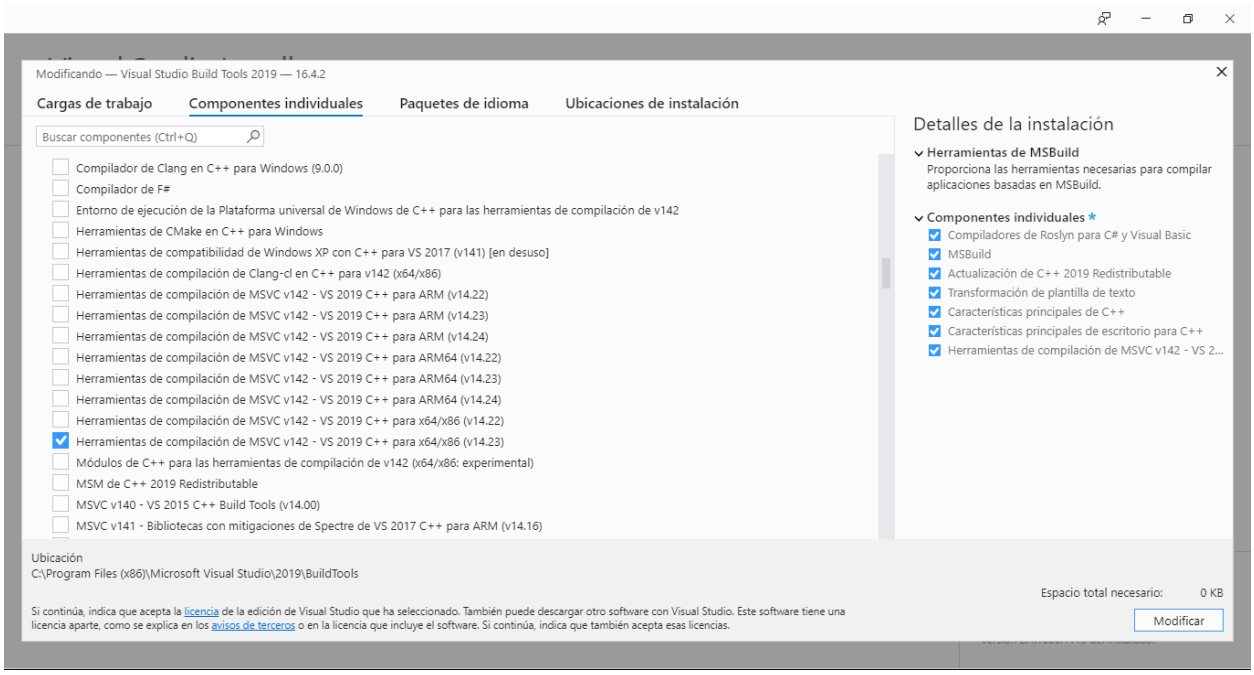


Figura 8-3 Selección del compilador necesario para la instalación

Ahora debería ser capaz de instalar el módulo *pybluez* ejecutando el siguiente comando desde la línea de comandos si este está siendo ejecutado con privilegios de administrador:

- *pip install –upgrade setuptools*
- *pip install pybluez*

Una vez todos los módulos necesarios para el funcionamiento han sido instalados, debe descargar de https://github.com/frarambra/pyGUIno el software que contiene el programa la interfaz y las librerías de Arduino. Para descargarlo pulse el botón "Clone or Download" y elija en el nuevo menú "Download ZIP".

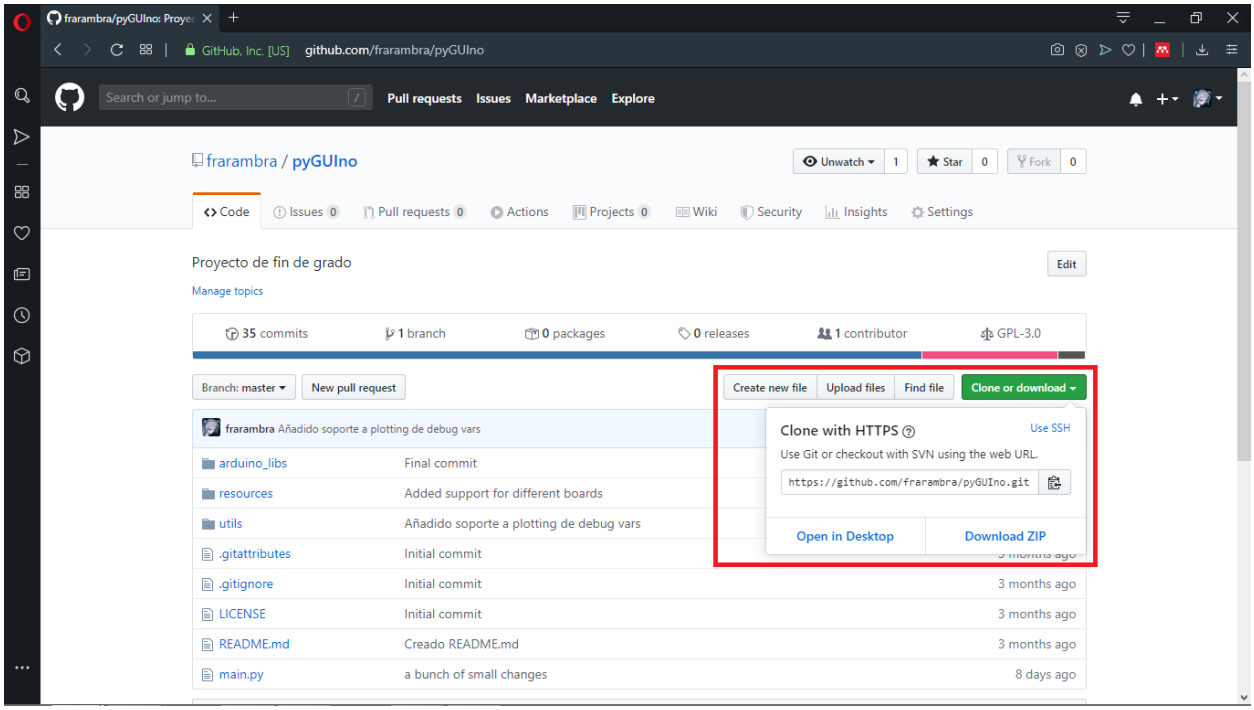


Figura 8-4 Repositorio con el software para la interfaz

Una vez descargado el archivo ZIP deberá descomprimirlo y mover los contenidos de la carpeta "arduino_libs" a la carpeta "Arduino/libraries", dentro de la carpeta que fue escogida en la instalación del programa Arduino IDE.

Tras todo este proceso podrá incluir en los programas para Arduino las librerías "*PyGUIno.h*", "*PyGUInoWire.h*" y "*PyGUInoSPI.h*".

8.2 Software para el ordenador

Para ejecutar el programa solo deberá hacer doble click en el archivo *main.py*. Tras esto se mostrará en la pantalla una ventana como la reflejada en la figura 8-5. La ventana está compuesta por una barra de herramientas, una zona de trabajo y una barra de estado.

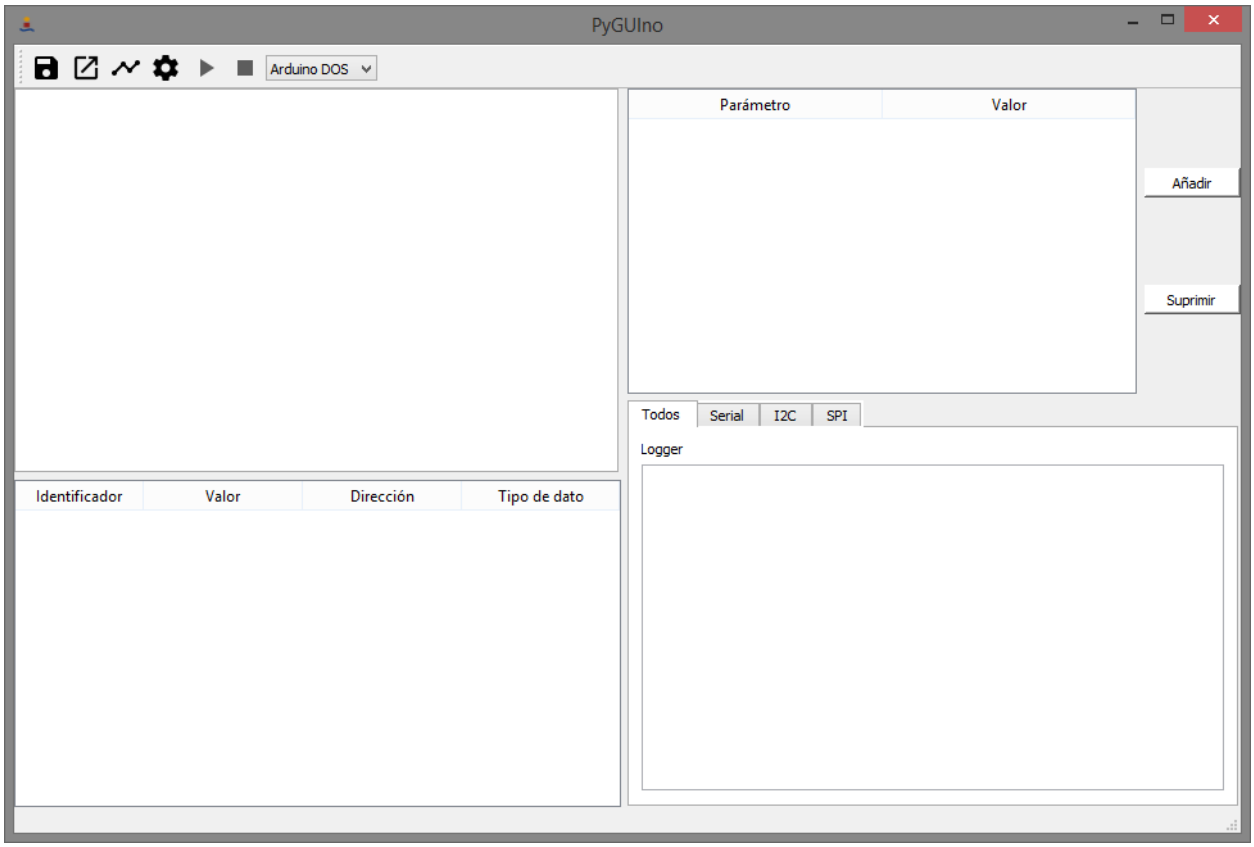


Figura 8-5 Pantalla del software creado

8.2.1 Zonas de trabajo

Los cuatro elementos de la zona de trabajo son: el área de gráficas, tabla de variables de usuario, tabla de variables de depuración y área de *loggers*. Cada área de la zona de trabajo es de tamaño variable, pudiendo el usuario ajustar el tamaño de cada zona desplazando el separador vertical u horizontal entre ellas.

A continuación, describimos cada uno de estos cuatro elementos:

- **Área de gráficas**: situada en la parte superior izquierda. Permite al usuario obtener una representación gráfica de los datos recuperados de Arduino. El usuario puede crear múltiples pestañas en cada una de las cuales podrá generarse una gráfica que a su vez podrá contener varias curvas. Las curvas podrán representar los datos originales que proceden de la placa o bien los datos que resulten de una transformación de los originales mediante una función matemática. Las curvas representadas podrán ser configuradas por el usuario para que tengan diversos colores. Las escalas para la correcta representación en la ventana de las curvas son ajustadas automáticamente por la aplicación, es decir, las gráficas son autoescalables.
- **Tabla de parámetros definidos por el usuario**: situada en la parte superior derecha, en esta tabla el usuario puede definir parámetros propios, que se pueden usar en las fórmulas matemáticas que se apliquen para la representación de las curvas. En esta zona, además, se encuentran dos botones que permiten la adición de parámetros a la tabla, así como su eliminación.
- **Tabla de variables de depuración**: esta tabla muestra información de variables de Arduino que el usuario indicó en el Sketch. Para cada variable que el usuario haya indicado, se muestran el identificador que el usuario le dio en el Sketch, su valor, su dirección de memoria y el tipo de dato almacenado en la memoria de Arduino. Dichas variables, permiten también su uso en las fórmulas matemáticas del área de gráficas.
- **Área de *loggers***: provee al usuario de la información sobre los mensajes que la placa Arduino recibe

de sus posibles periféricos, así como aquellos mensajes referentes a la comunicación entre la placa Arduino y el ordenador.

8.2.2 Barra de herramientas y barra de estado

La barra herramientas consta de 6 botones y cuadro de selección. La barra de herramientas permite ser ubicada a elección del usuario mediante su selección, desplazamiento y fijación de la posición final con el botón izquierdo del ratón. En la tabla continua se muestran las acciones para cada botón:

Tabla 8-1 Funcionalidad de cada botón de la barra de herramientas

Icono	Funcionalidad
	Guarda la configuración de las gráficas que el usuario haya definido previamente para cada pestaña en un archivo .json
	Carga la configuración de las gráficas almacenada en un archivo .json
	Lanza el cuadro de diálogo que permite al usuario configurar los parámetros de visualización de las gráficas.
	Abre el cuadro de diálogo para configurar los parámetros de la conexión con la placa.
	Inicia la recepción de mensajes y datos desde la placa.
	Detiene la recepción de mensajes y datos desde la placa.
	Permite seleccionar el modelo de placa de Arduino para adecuar la correcta interpretación de la longitud de los datos en bytes. Las opciones son modelos citados en la tabla 2-1

Además, mediante el uso de la barra de estado que se encuentra en el lado inferior de la ventana es posible obtener una pequeña descripción de la acción asignada a cada botón.

8.2.3 Configuración de la conexión y comienzo de la recepción de mensajes

Para configurar la conexión debe pulsarse el botón  "Conectar" de la parte superior izquierda de la barra de herramientas. Al pulsar este botón se abre el diálogo para configurar la conexión. Se puede seleccionar entre dos tipos de conexión, serie y Bluetooth.

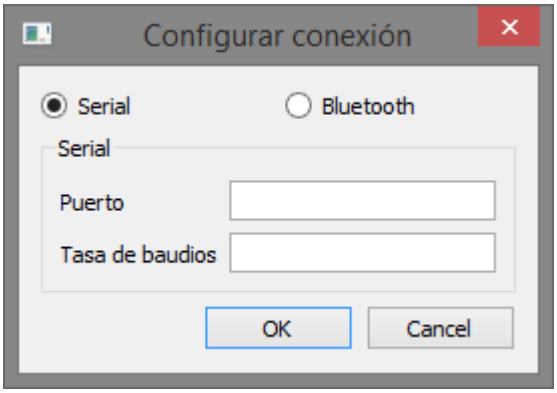


Figura 8-6 Diálogo para configurar la conexión mediante serie

En caso de selección de la conexión serie, se debe introducir el puerto y la tasa de baudios. En el caso de usar Windows, normalmente el puerto suele ser nombrado "*COM*", seguido de un número, por ejemplo, "*COM3*". En MacOS y distribuciones Linux normalmente el puerto suele ser nombrado "*dev/tty0*" o similar. Respecto a la tasa de baudios, normalmente suelen escogerse valores de 9600, 19200 baudios. No obstante, el valor de la tasa de baudios deberá ser el mismo que el especificado en el "*Sketch*" escrito en la memoria de Arduino.

En caso de seleccionar conexión Bluetooth se debe saber el nombre establecido para el dispositivo Bluetooth con el que se va a conectar con el fin de poder seleccionarlo dentro del dialogo para poder realizar la conexión. También deberá estar activada la conexión Bluetooth en el ordenador.

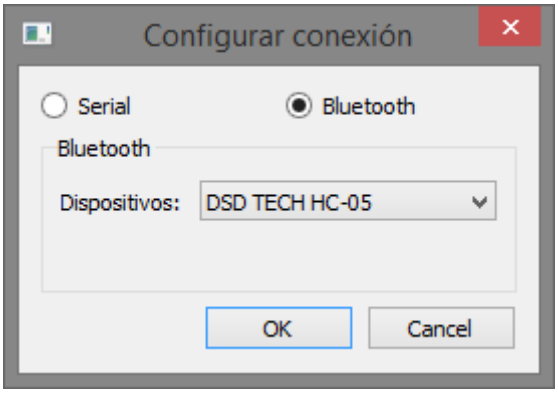


Figura 8-7 Diálogo para configurar la conexión Bluetooth

Una vez la conexión se ha establecido se habilitará el botón ▶ de recibir mensajes, que permite al programa comenzar la recepción de los mismos, actualizando la tabla de variables de depuración y gráficas que se hayan definido previamente.

8.2.4 Añadir una nueva gráfica

Para añadir una nueva gráfica deberá pulsar el botón "Añadir gráfica", esto abrirá el diálogo de la figura 8-8, que permite configurar la gráfica que será añadida. En el campo "Título" podrá escoger el nombre de la gráfica.

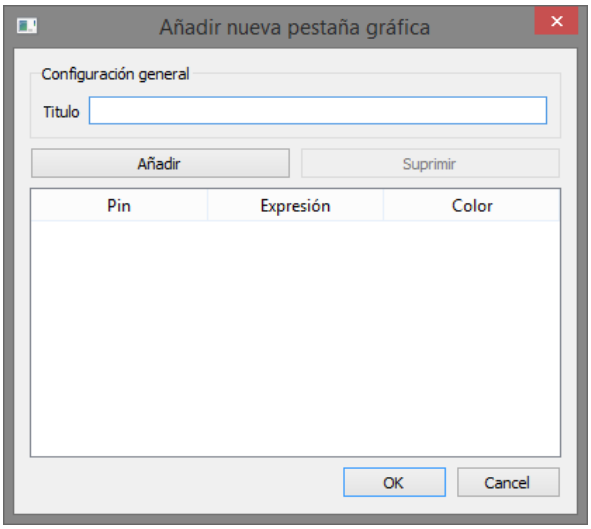


Figura 8-8 Diálogo para añadir gráficas

Puede observar que hay una tabla vacía en el cuerpo del diálogo. Esta tabla contendrá los parámetros de las curvas que serán representadas en la gráfica. Para añadir una entrada a la tabla pulse el botón "Añadir", que abrirá el diálogo de la figura 8-9.

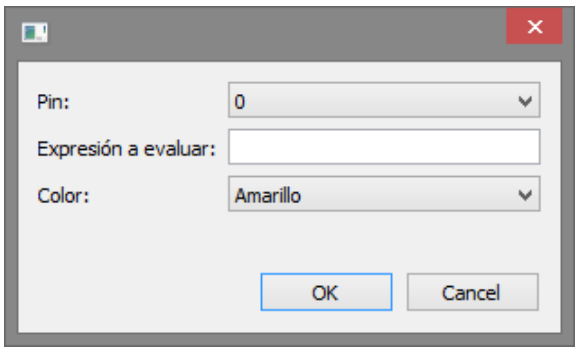


Figura 8-9 Diálogo para añadir parámetros de una curva

El diálogo se compone de 3 campos, pin, expresión a evaluar y color. A continuación, se explican en detalle cada uno de los campos:

- **Pin**: permite seleccionar el pin o variable de depuración con el cual la curva que se añadirá será actualizada.
- **Expresión a evaluar**: permite introducir una expresión matemática que será utilizada para calcular el valor asociado a la curva. Permite usar parámetros definidos en la tabla de parámetros, variables de depuración y pines. Para hacer uso de esta función, simplemente escriba entre {} el nombre de la variable a la que quiere referenciar. En caso de error en el cálculo o que este campo este vacío se representará el valor que el mensaje "*transmit_pin_value*" porta.
- **Color**: Permite la selección del color de la curva entre un conjunto de colores predefinidos.

REFERENCIAS

[1] T. Monjo Palau, A. Mora Carreño, M. Garreta Domingo, y J. Mora, «Interface Toolkit | Graphical user interface». [En línea]. Disponible en: http://disseny.recursos.uoc.edu/materials/interface-toolkit/es/graphical-user-interface/. [Accedido: 02-ene-2020].

[2] M.-C. Marcos, «HCI (human computer interaction): concepto y desarrollo», *El Prof. la Inf.*, vol. 10, n.º 6, pp. 4-16, 2001, doi: 10.1076/epri.10.6.4.8200.

[3] «Introduction - Material Design». [En línea]. Disponible en: https://material.io/design/introduction/. [Accedido: 02-ene-2020].

[4] J. Johnson y J. Johnson, «First Principles», *GUI Bloopers 2.0*, pp. 7-50, ene. 2008, doi: 10.1016/B978-012370643-0.50001-9.

[5] «About \ Wiring». [En línea]. Disponible en: http://wiring.org.co/about.html. [Accedido: 02-ene-2020].

[6] «Arduino - Products». [En línea]. Disponible en: https://www.arduino.cc/en/Main/Products. [Accedido: 02-ene-2020].

[7] Atmel, «ATmega328P 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash DATASHEET».

[8] Atmel, «SAM3X / SAM3A [DATASHEET]», 2015.

[9] Microchip, «SAM D21 Family Low-Power, 32-bit Cortex-M0+ MCU with Advanced Analog and PWM Features», 2018.

[10] B. Stroustrup, *The C++ programming language*. Addison-Wesley, 1997.

[11] «1. Extending Python with C or C++ — Python 3.5.9 documentation». [En línea]. Disponible en: https://docs.python.org/3.5/extending/extending.html. [Accedido: 11-ene-2020].

[12] «Philosophy — Kivy 1.11.1 documentation». [En línea]. Disponible en: https://kivy.org/doc/stable/philosophy.html. [Accedido: 02-ene-2020].

[13] «About Qt - Qt Wiki». [En línea]. Disponible en: https://wiki.qt.io/About_Qt. [Accedido: 02-ene-2020].

[14] «Installing — Matplotlib 3.1.2 documentation». [En línea]. Disponible en: https://matplotlib.org/users/installing.html. [Accedido: 11-ene-2020].

[15] «Installation — pyqtgraph 0.10.0 documentation». [En línea]. Disponible en: http://www.pyqtgraph.org/documentation/installation.html. [Accedido: 11-ene-2020].

[16] «Introduction — pyqtgraph 0.10.0 documentation». [En línea]. Disponible en: http://www.pyqtgraph.org/documentation/introduction.html. [Accedido: 02-ene-2020].

[17] «Fast Live Plotting in Matplotlib / PyPlot - Stack Overflow». [En línea]. Disponible en: https://stackoverflow.com/questions/40126176/fast-live-plotting-in-matplotlib-pyplot. [Accedido: 02-ene-2020].

[18] «harmsm/PyCmdMessenger: Python interface for CmdMessenger arduino serial communication library». [En línea]. Disponible en: https://github.com/harmsm/PyCmdMessenger. [Accedido: 02-ene-2020].

[19] «Threads Events QObjects - Qt Wiki». [En línea]. Disponible en: https://wiki.qt.io/Threads_Events_QObjects#Events_and_the_event_loop. [Accedido: 02-ene-2020].

[20] «Choosing a transport protocol». [En línea]. Disponible en: https://people.csail.mit.edu/albert/bluez-intro/x95.html. [Accedido: 02-ene-2020].

GLOSARIO

ARM:
Advanced RISC Machines

GUI:
Graphical User Interface

HCI:
Human-Computer Interaction

I2C:
Inter-Integrated Circuit

IDE:
Integrated Development Environment

IP:
Internet Protocol

Loggers:
Partes del software que captura mensajes

Script:
Conjunto de órdenes en un lenguaje de programación.

SPI:
Serial Peripheral Interface

Splitters:
Separadores

WIMP:
Windows Icons Menus Pointers