Proyecto Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica.

GUI para el control de fuentes de alimentación HP 66000A con BeagleBone Black

Autor:

Antonio Millán Díaz

Tutores:

Fernando Muñoz Chavero

Javier Galnares Arias

Dpto. Ingeniería electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Proyecto Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

GUI para el control de fuentes de alimentación HP 66000A con BeagleBone Black

Autor:
Antonio Millán Díaz

Tutores:
Fernando Muñoz Chavero – Profesor Titular
Javier Galnares Arias – Alter Technology

Dpto. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Proyecto Fin de Grado: GUI para el control de fuentes de alimentación HP 66000A con BeagleBone Black

Autor: Antonio Millán Díaz

Tutores: Fernando Muñoz Chavero
Javier Galnares Arias

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal

Agradecimientos

A mis tutores.

A mi hermana y a mis padres, por apoyarme incondicionalmente durante todos estos años de estudios.

Y a Alter Technology, por darme la oportunidad de realizar este proyecto junto a ellos.

Antonio Millán Díaz

Sevilla, 2019

Resumen

Este proyecto es un encargo de la empresa Alter Technology, para llevar a cabo una interfaz gráfica capaz de controlar varias fuentes de alimentación de la gama HP 66000A que tienen a su disposición en uno de sus laboratorios. Para ello se construirá un programa usando el lenguaje de programación Python, utilizando sus librerías: Tkinter y PyVisa.

La aplicación demandada, consistirá en una GUI capaz de modificar las salidas de voltaje e intensidad de los distintos módulos, además de representar gráficamente estos. Dicha representación, actualizará los datos en tiempo real, y se construirá utilizando una BeagleBone Black, acompañada de una pantalla táctil capacitiva.

Abstract

This project is an order from the company Alter Technology, who asked for a graphic interface that should able to control some power modules from the HP 66000A series which they own in one of their laboratories. In order to do so, a computer program is built, using the programming language Python, as well as its libraries: Tkinter and PyVisa.

The demanded job will be to build a GUI app intended to be able to modify the outputs of voltages and intensities coming from the different modules, as plotting them aswell. The given plot, which is to be implemented using a BeagleBone Black endowed with a capacitive touch screen, will have the data updated in real time.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

API	Application Programming Interface
BBB	BeagleBone Black.
DC	Direct Current.
GPIB	General Purpose Instrumentation Bus.
GUI	Graphical User Interface.
HP-IB	Hewlett-Packard Instrument Bus.
IEEE	Institute of Electrical and Electronics Ingenieers.
IP	Internet Protocol
MPS	Modular Power System.
OOP	Object Oriented Programming.
SBC	Single Board Computer.
SCPI	Standard Commands for Programmable Instruments.
S.O.	Sistema Operativo
USB	Universal Serial Bus
VISA	Virtual Instrument Software Architecture.

1 INTRODUCCIÓN

Actualmente, la gran mayoría de las aplicaciones que nos rodean ocultan su funcionamiento bajo una interfaz gráfica. El uso de interfaces gráficas de usuario estimula y motiva a las personas ajenas a la comunidad informática a utilizar programas y aplicaciones, que no podrían utilizar sin ellas.

Una interfaz gráfica, establece una comunicación entre la máquina y el usuario, sin necesidad de que este manifieste conocimientos de programación.

En los 60, el concepto de ordenador personal era impensable, los ordenadores eran máquinas complejas, y la mayoría no disponía de pantalla, y si lo hacían, lo más probable es que recibieran al usuario en código binario. Esto restringía el uso de las computadoras, las cuales sólo podían ser utilizadas por usuarios con conocimientos informáticos.

A mediados de los 70, surge la idea de GUI – *Graphical User Interface*, de la mano de "Alto", una computadora desarrollada por Xerox.

Figura 1-1. Computadora Alto, de Xerox.

Xerox, era una compañía dedicada a la fabricación de fotocopiadoras que, ante la necesidad de utilizar una computadora capaz de trabajar con sus impresoras, creó Alto. También desarrollaron SmallTalk, un lenguaje de programación orientado a objetos (OOP – Object Oriented Programming), cuya influencia llega hasta nuestros días a través de los lenguajes orientados a objetos actuales, como Python, Java o Ruby.

Con todo esto sobre la mesa, en 1983 llegaría "Lisa", un ordenador diseñado por Apple, que tomó varios de los conceptos desarrollados en Xerox. Y que, a pesar de su fracaso comercial, asentó el concepto de GUI, que a lo largo de la década evolucionaría de la mano de GEM, o Windows.

Gracias a la influencia de las interfaces gráficas, los ordenadores han mutado su comportamiento, dejando de ser máquinas de funcionalidad única o determinada, y alejadas de la sociedad, para convertirse en elementos con multitud de funcionalidades, fáciles de usar y cercanos al público general.

2 ENTORNO DE TRABAJO

En este capítulo se informará al lector acerca del ámbito de trabajo en el que se ha llevado a cabo este proyecto final de estudios. De esta forma, se proporcionarán datos sobre el lugar, la empresa, los elementos usados para su elaboración, y los conocimientos que se han adquirido previos a su desarrollo.

2.1 Alter Technology

Alter Technology es una agrupación resultado de la unión de varias empresas europeas, y perteneciente a TÜV NORD GROUP. Algunos miembros distinguidos de esta agrupación serían Hirex Engineering, Optocap, DMT, Tüvit, y Tüv Nord, entre otros. Alter Technology se dedica principalmente a la ingeniería, ensayo y aprovisionamiento de sistemas y componentes electrónicos, así como a la certificación de sistemas de alarma y detección de intrusos. Además, libra un importante papel en los proyectos espaciales europeos, y otros sectores altamente exigentes.

Figura 2-1. Logotipo de la compañía.

La compañía comenzó su andadura en 1986, con la fundación de Tecnologica, cuya sede se estableció en Madrid. Con el tiempo, adquiere e incorpora varias empresas, como Hirex Ingeneering, Alcatel, IGG o Tropel. Tecnologica puede considerarse como la semilla de la actual Alter Technology, que en 2011 se integra dentro de Tüv Nord Group, donde pasa a encabezar la nueva división del sector aeroespacial del grupo alemán. Desde entonces, Tüv Nord Group ha incorporado algunas otras empresas, como Optocap.

Por otro lado, en lo que respecta a Alter Technology, la compañía ha recibido varios premios desde su incorporación al grupo, tanto desde organismos públicos (como el Jurado Provincial de Sevilla o PCT Cartuja) como privados (el propio Tüv Nord Group). En la actualidad, está compuesta por 5 sedes, situadas en Sevilla, Madrid, Portsmouth (Reino Unido), San Petersburgo (Rusia) y Shangai (China). Contando con más de 250 empleados y más de 6000 metros cuadrados de laboratorios.

Con la ayuda de Alter Technology, Tüv Nord Group se dedica a proporcionar lo siguientes servicios:

- Desarrollo integral de componentes electrónicos bajo demanda, desde etapas iniciales de diseño y asesoramiento, hasta su propia elaboración y suministro, finalizando con los pertinentes análisis de calidad, donde destacan los tests de tipo eléctrico, físico-mecánico, medioambientales, y de radiación, entre otros.
- Ensamblado y encapsulado, de la mano de Optocap, con el objetivo de reducir gastos de desarrollo y fabricación, lo que permite a sus clientes facilitar la salida de sus productos al mercado.
- Labores de certificación, asegurando que sus productos cumplen las normativas y regulaciones de los organismos pertinentes.
- Prueba y certificación de sistemas de detección de intrusos, así como sistema de seguridad, alarmas y controles de acceso, de acuerdo con la normativa europea, y las regulaciones de tipo nacional o de características especiales (como la directiva CPD en el caso de los detectores de incendios).

- Asesoramiento en el ámbito de las aeronaves pilotadas de forma remotas (drones), donde también se elaboran testados y certificaciones.

La actividad principal de Alter Techonology consiste en colaborar con proyectos aeroespaciales, tanto europeos como no europeos. No obstante, también están presentes en otros mercados, como el de seguridad, transporte, emergencias, salud y automoción.

2.2 Laboratorio de climáticos

El desarrollo de este proyecto ha tenido lugar en el laboratorio de climáticos de Alter Technology, donde se llevan a cabo pruebas a diversos componentes electrónicos para su calificación para aplicaciones de espacio, así como para conocer su durabilidad, cualidades ante estrés térmico, y fiabilidad.

Algunos de los ensayos llevados a cabo en el laboratorio de ambientales son:

- **Baking:** Consiste en meter el componente sin polarización en un horno de temperatura controlada y humedad no controlada (estufa de laboratorio). Las temperaturas oscilan normalmente entre los 85 y los 125 ºC, y su duración suele ser de hasta 24 horas.
- **Burn-in:** Muy similar al anterior, pero difiere en que los dispositivos introducidos se encuentran encendidos, es decir, con polarización. Por ejemplo, para que un condensador se considere que está en burn-in y no en baking, debe estar excitado a su tensión nominal de funcionamiento. En función del componente, la duración oscila entre 168 y 240 horas.
- **Life-test:** Es un burn-in de larga duración. Este tipo de ensayos suelen oscilar entre 1000 y 3000 horas de duración. La razón de hacer un life-test para calificar el componente es que, del número de supervivientes tras 1000, 2000 o 3000 horas a la temperatura máxima de funcionamiento, se puede inferir el número de horas de vida o fiabilidad del componente, ya que hay una relación cuadráticamente inversa entre la temperatura de operación del dispositivo y la longevidad del mismo. Por ejemplo, si un dispositivo a 80 ºC supera las 3000 h, significa que a la mitad de temperatura durará 4 veces más, esto es, que a 40 ºC durará al menos 12000 h.
- **Thermal-shock:** Consiste en someter a un cambio brusco en la temperatura a los componentes, pasándolos de una cámara a la temperatura mínima de almacenamiento del dispositivo, a otra a la máxima de funcionamiento del componente, en un lapso de tiempo que no suele superar los 2 - 3 minutos.
- **Temperature cycling:** Similar al choque térmico, pero el cambio de temperatura no es abrupto, sino que se somete a una rampa suave inferior a 5 ºC/min. En estos ensayos el componente suele estar polarizado y se suele usar para calificaciones especiales.

2.3 Planteamiento del problema

El objetivo del presente proyecto es llevar a cabo una interfaz gráfica (GUI), que nos permita controlar fuentes de alimentación de la gama HP 66000A.

Estas fuentes de alimentación, son utilizadas en Alter Technology durante los ensayos en los que es necesario que los dispositivos se encuentren polarizados, y reciban tensión y corriente. Para controlar las fuentes, estas tienen un teclado que permite al operador comunicarse con ellas mediante comandos SCPI.

La necesidad de llevar a cabo una GUI, se debe a que resulta difícil y poco intuitivo modificar la salida de las fuentes haciendo uso del teclado, y más aún durante los ensayos.

Una vez que se ha entendido el problema, y antes de pasar a desarrollar cómo se ha solucionado, se informará al lector acerca de los distintos elementos que rodean a dichas fuentes de alimentación, y se aclararán algunos conceptos de importancia que han sido utilizados para llevar a cabo el trabajo.

2.3.1 Fuentes de alimentación HP 66000A.

Las fuentes HP 66000A se utilizan para proporcionar salidas de intensidad y voltaje DC, que pueden ser utilizadas por el usuario. Estos módulos de energía (power modules), cuentan con dos displays digitales que informan del nivel de tensión y de intensidad de la salida. Para utilizarlos, estos se introducen en la unidad central de módulos, o mainframe (MPS – Modular Power System).

La mainframe tiene capacidad para ocho módulos, que pueden conectarse a ella sin necesidad de cablear nada. En la siguiente figura, vemos un módulo de energía (verde) contenido en la mainframe (azul):

Figura 2-2. Módulo de energía dentro de una mainframe (MPS).

En función del modelo, las fuentes tendrán un rango de salidas de tensión y corriente distinto. Pero todas ofrecen la opción de proporcionar salidas en modo corriente constante (CC), o voltaje constante (CV). Los MPS, con sus distintos módulos, son muy prácticos y funcionales para poder generar un gran abanico de tensiones y corrientes.

Figura 2-3. Un MPS junto a su teclado.

2.3.2 Bus GPIB

Las fuentes se controlan utilizando un bus GPIB (General Purpose Instrumentation Bus). Este tipo de buses se utilizan para establecer comunicaciones entre varios instrumentos de medida.

Hewlett-Packard fabricó y diseño la herramienta, llamándola HP-IB (Hewlett-Packard Instrument Bus). A principios de los 70, la industria comenzó a implementar diversos duplicados del bus, convirtiéndose en uno de los más utilizados, razón por la que terminaría conociéndose por el nombre que le damos en la actualidad.

En la industria, es muy frecuente que los dispositivos de instrumentación utilicen un bus GPIB para comunicarse.

Utiliza un lenguaje basado en comandos, o mensajes: SCPI. Que utilizaremos para comunicarnos con las mainframes (siempre que estas estén físicamente conectadas al bus). De esta forma, podremos acceder y controlar los ocho módulos desde una misma dirección GPIB.

2.3.3 Comandos SCPI

Los módulos de la serie HP 66000A se programan mediante comandos SCPI (Standard Commands for Programmable Instruments). Un conjunto de órdenes comunes utilizadas sobre el bus GPIB para controlar las funcionalidades de instrumentos y dispositivos de características y construcciones completamente diferentes. Fueron desarrolladas a principios de la década de 1990 a partir de la norma IEEE 488.2.

La idea es controlar con el mismo comando funcionalidades comunes entre dos dispositivos distintos. Por ejemplo: utilizar el mismo comando para controlar la salida de un módulo HP 66000A, que para la salida de un instrumento totalmente distinto.

En nuestro caso, para controlar los módulos de energía, se han usado varios de los comandos SCPI que el manual de usuario de las fuentes HP 66000A recomendaba, para la realización de este proyecto, los más importantes son:

Tabla 2-1. Comandos SCPI.

Comando	Descripción
*IDN?	Pide al instrumento que se identifique.
MEAS:VOLT?	Mide la tensión a la salida.
MEAS:CURR?	Mide la intensidad a la salida.
VOLT:LEV?	Pregunta que nivel de tensión se estableció.
CURR:LEV?	Pregunta que nivel de intensidad se estableció.
VOLT:LEV X	Establece el nivel de tensión a x voltios.
CURR:LEV X	Establece el nivel de intensidad a x amperios.
OUTP:STAT?	Pregunta el estado de la salida: habilitada o inhabilitada.
OUTP:STAT ON	Habilita la salida.
OUTP:STAT OFF	Deshabilita la salida.

2.3.4 Ethernet Gateway

Con el objetivo de comunicarnos inalámbricamente con las mainframes, estas se han conectado a una pasarela de medios, o puerta de enlace.

Una puerta de enlace, o *Gateway* en inglés, permite conectar elementos o dispositivos pertenecientes a redes con arquitecturas y protocolos diferentes, en nuestro caso, GPIB a Ethernet. Gracias a la pasarela, podremos comunicarnos con la mainframe utilizando una dirección IP, a la cuál enviaremos comandos SCPI para controlar los módulos.

Figura 2-4. Gateway Agilent E5810B

2.3.5 VISA

A la hora de comunicarnos con instrumentos de medida, nos encontramos ante la presencia de varios tipos de protocolos, que se utilizan a través de distintas redes y sistemas.

Por ello, en la década de 1990 se desarrolló el estándar: Virtual Instrument Software Architecture (VISA), con el objetivo de facilitar esta tarea. VISA proporciona muchas facilidades a la hora de programar y solucionar problemas que usan sistemas de instrumentación, como GPIB.

VISA proporciona una interfaz de programación de aplicaciones (API), capaz de olvidar las diferencias existentes entre los sistemas de instrumentación, y unificando los métodos de acceso y comunicación a los distintos dispositivos independientemente de su bus de comunicaciones. El estándar, es utilizado por las aplicaciones de instrumentación más extendidas, como VisualBasic, Labview, o Matlab, aunque también esta presente en otros lenguajes de programación, como Python o C.

Como veremos en el capítulo 3, se utilizará la librería PyVisa de Python para llevar a cabo las comunicaciones entre la interfaz gráfica y los módulos de energía HP 66000A.

2.4 Componentes utilizados

2.4.1 BeagleBone Black

La BeagleBone Black es una placa (SBC – Single Board Computer), que cuenta con un procesador basado en ARM Cortex-A8. La BeagleBone Black (BBB), es un elemento fabricado por Circuitco LLC, cuya sede se encuentra en Texas (EE. UU).

Fue diseñada por Gerard Coley para la fundación BeagleBoard.org, que es una compañía de Michigan sin ánimo de lucro dedicada al diseño y uso de software y hardware libre en el entorno de la computación embebida. Su lanzamiento se llevó a cabo el 23 de Abril de 2013, a un precio de 45$.

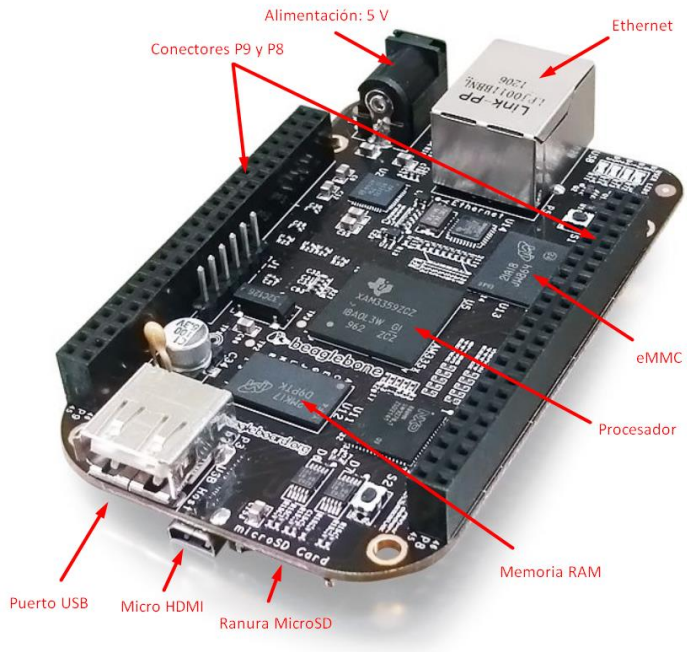

Figura 2-5. BeagleBone Black.

Es la última incorporación a la familia de placas SBC de las que dispone la fundación, entre las que se

encuentran: BeagleBoard, BeagleBoard-xM, BeagleBone, BeagleBone Black y BeagleBoard-X15. Para su construcción, se unieron empleados de Texas Instruments, DigiKey y la propia Circuitco. Entre los que se encontraba el propio portavoz de BeagleBoard, Jason Kridner.

A continuación, se recopilan en la siguiente tabla algunas de las características de la placa, proporcionando una descripción de alto nivel de los componentes que la forman.

Tabla 2-2. Características de la BeagleBone Black.

	Feature	
Processor	Sitara AM3359AZCZ100 1GHz, 2000 MIPS	
Graphics Engine	SGX530 3D, 20M Polygons/S	
SDRAM Memory	512MB DDR3L 606MHZ	
Onboard Flash	2GB, 8bit Embedded MMC	
PMIC	TPS65217C PMIC regulator and one additional LDO.	
Debug Support	Optional Onboard 20-pin CTI JTAG, Serial Header	
Power Source	miniUSB USB or DC Jack	5VDC External Via Expansion Header
PCB	3.4" x 2.1"	6 layers
Indicators	1-Power, 2-Ethernet, 4-User Controllable LEDs	
HS USB 2.0 Client Port	Access to USB0, Client mode via miniUSB	
HS USB 2.0 Host Port	Access to USB1, Type A Socket, 500mA LS/FS/HS	
Serial Port	UART0 access via 6 pin 3.3V TTL Header. Header is populated	
Ethernet	10/100, RJ45	
SD/MMC Connector	microSD , 3.3V	
User Input	Reset Button Boot Button Power Button	
Video Out	16b HDMI, 1280x1024 (MAX) 1024x768,1280x720,1440x900 w/EDID Support	
Audio	Via HDMI Interface, Stereo	
Expansion Connectors	Power 5V, 3.3V , VDD_ADC(1.8V) 3.3V I/O on all signals McASP0, SPI1, I2C, GPIO(65), LCD, GPMC, MMC1, MMC2, 7 AIN(1.8V MAX), 4 Timers, 3 Serial Ports, CAN0, EHRPWM(0,2),XDMA Interrupt, Power button, Expansion Board ID (Up to 4 can be stacked)	
Weight	1.4 oz (39.68 grams)	
Power	Refer to Section 6.1.7	

2.5.1 Pantalla gen4-4DCAPE-70CT

Para llevar a cabo este proyecto, se ha usado Debian, un sistema operativo libre, que se ejecuta en la placa mediante una tarjeta microSD. Para poder visualizar el escritorio de Debian, se ha utilizado una pantalla táctil capacitiva.

La pantalla pertenece a la gama gen4 LCD CAPE, fabricada por 4DSystems. Una empresa dedicada a la

elaboración de displays inteligentes y procesadores gráficos, y que se fundó en 1990. Esta gama, que está desarrollada específicamente para la BeagleBone Black, y que utiliza un adaptador para conectarse a ella,

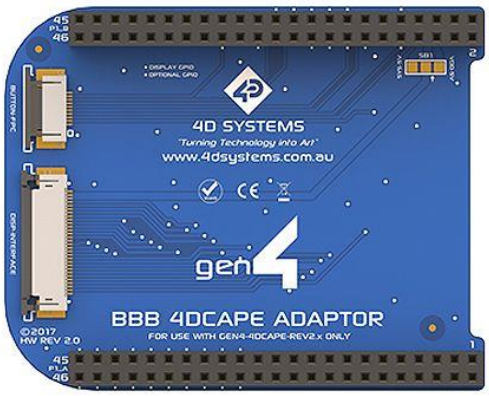

Figura 2-6. Adaptador gen4-4DCAPE para BBB.

posee un total de nueve clases de pantallas, todas pertenecientes a la serie 4DCAPE.

Para elaborar este proyecto final se ha utilizado una pantalla de siete pulgadas: gen4-4DCAPE-70CT, como la que se muestra en la siguiente figura:

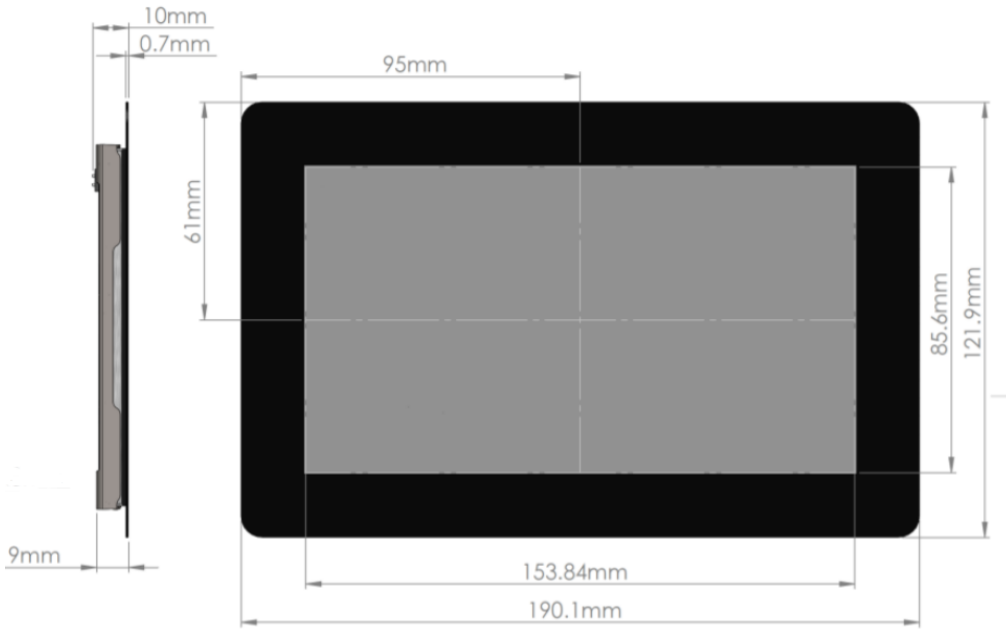

Figura 2-7. Pantalla gen4-4DCAPE-70CT.

3 PYTHON

En este capítulo se llevará a cabo una descripción del lenguaje de programación Python. No se pretende, en ningún caso, llevar a cabo un manual completo y exhaustivo del mismo, sino introducir al lector en el lenguaje de una forma simple y escueta, con el objetivo de ayudarle a comprender el trabajo realizado en el presente proyecto. Por tanto, quedarán fuera de este capítulo todos los contenidos sobre Python que no se hayan usado a la hora de elaborar el código final, o que se han considerado omitibles por tratar temas básicos y comunes a todos los lenguajes de programación, como por ejemplo: sintaxis, tipos de variables, operadores, condicionales, bucles, etc.

Además, se realizará una descripción de las librerías usadas para la realización del proyecto: Tkinter y PyVisa.

3.1 Principales características

Python es un lenguaje de programación creado por un grupo de programadores, al mando del holandés Guido van Rossum, en 1991. Su primera versión pública, fue la 0.9, ya que la versión 1.0 se retrasaría hasta 1994, mientras que las versiones 2.0 y 3.0 emergieron en los años 2000 y 2008, respectivamente. Su nombre, al contrario de lo que piensa la mayoría, no hace referencia a una pitón, sino que es un homenaje a los *Monty Python*, un grupo de humoristas británicos que era del agrado de Rossum.

Python destaca por ser un lenguaje de muy alto nivel, que cuenta con una gramática cómoda, sencilla, intuitiva, y legible. A continuación, se nombran algunas de sus características más notables:

- **Python es un lenguaje interpretado,** que ayuda a la experimentación y prueba de scripts. Cuenta con intérprete interactivo, y no es necesaria la compilación.
- **Es un lenguaje orientado a objetos,** beneficiándose de las características principales que esto conlleva: herencia, encapsulación, polimorfismo, etc.
- **Es de código abierto.**
- **Posee un tipado dinámico fuerte,** creando una distinción muy clara entre los diferentes tipos de variables, pero sin la necesidad de establecer dicho tipo al momento de la creación de la variable (dinamismo).
- **Es un lenguaje extensible,** que acepta la incorporación de nuevas funciones y módulos fácilmente.
- **Muy divulgado:** cuenta con una amplia gama de libros y publicaciones.
- **Gramática sencilla: s**uprime muchos de los símbolos usados por otros lenguajes, como por ejemplo el punto y coma. Lo que convierte a Python en un lenguaje ideal para iniciarse en la programación.

3.2 Algunos conceptos

En la jerga de Python, algunos de los conceptos tradicionales de programación han transformado su nombre, dando lugar a errores frecuentes entre los usuarios. En este apartado se ilustrará al lector en estos pequeños conceptos que pertenecen al lenguaje, y que se han usado para elaborar este proyecto final.

3.2.1 Métodos

Todos los lenguajes de programación de alto nivel utilizan el término: función. Como se sabe, una función es un fragmento de código, que se utilizará para interrumpir el flujo principal del programa y llevar a cabo la tarea descrita en su interior. Python utiliza las funciones de forma similar a como se haría en cualquier otro lenguaje: declaración, paso de parámetros, etc.

En Python, también existe un tipo de función denominada método. La principal característica de un método, y

lo que lo hace diferente a una función normal, es que su definición, se lleva a cabo en el interior de un objeto (una clase). Esto quiere decir, que un método es una tarea que solo podrá ser llamada por el objeto al que pertenezca, al contrario que una función normal, que puede ser llamada desde cualquier otro punto del código, incluida otra función.

3.2.2 Estructuras

Las estructuras de datos nos permiten almacenar variables dentro de una estructura, para acceder a ellas de forma ordenada. En Python existen varios tipos distintos, así que llevaremos a cabo una breve descripción de cada uno, nombrando sus principales diferencias con las estructuras clásicas de otros lenguajes de programación.

3.2.2.1 Listas

Una lista es el equivalente a un *array* tradicional, conocido habitualmente como vector por otros lenguajes de programación. La principal diferencia con el vector clásico, es que en Python podemos declarar una lista con tipos de variables distintos, de forma que en una misma estructura convivan enteros con flotantes, o con cadenas de caracteres. Además, estas pueden expandirse de forma dinámica, sin necesidad de llevar a cabo una reserva de memoria.

Para declarar una lista se usa el operador "[]" de la siguiente forma:

```
lista=[2,"Juan",4.56]
```

Figura 3-1. Ejemplo de lista.

También llevan asociados un conjunto de métodos que pueden usarse para llevar a cabo operaciones con ellas, siendo algunos de ellos:

- *Lista.append(elem)*: Añade *elem* al final de la lista.
- *Lista.insert(i, elem)*: Inserta *elem* en el posición *i* de la lista.

3.2.2.2 Tuplas

Una tupla es otra forma de crear conjuntos de variables en Python. Son muy parecidas a las listas, pero son inmutables, es decir, no podemos realizar cambios en ellas, tales como añadir elementos, eliminarlos o modificarlos, de forma que estarán restringidas a sólo lectura. Por tanto, no tienen cabida aquí métodos como *append()*, como los asociados a las listas que hemos nombrado anteriormente.

Las ventajas que nos ofrecen las tuplas con respecto a las listas, es que estas suelen ocupar menos espacio en memoria, y se almacenan de una forma más óptima. Esto convierte a las tuplas en estructuras bastante más rápidas que una lista.

Para declarar una tupla se utiliza el operador "()" de la siguiente forma:

```
tupla=(2,"Juan",4.56)
```

Figura 3-2. Ejemplo de tupla.

3.2.2.3 Diccionarios

Otra estructura de datos importante que ofrece Python son los diccionarios, cuya principal característica es la

forma en la que se acceden a los elementos que almacenan. En un diccionario, todos los elementos llevan asociados una clave, de forma que el orden de los elementos dentro de la estructura sea indiferente. Esto quiere decir que para acceder a los datos de un diccionario sólo es necesario proporcionar la clave asociada a dicho elemento.

Para declarar un diccionario se utiliza el operador "{}" de la siguiente forma:

```
diccionario={clave:elemento, "pera":"fruta", 2:"par", 3:"impar"}
```

Figura 3-3. Ejemplo de diccionario.

3.3 Lenguaje orientado a objetos

Python es un lenguaje de programación orientado a objetos (OOP). Este tipo de lenguajes, adquirieron un protagonismo mayor en la década de 1980, debido a la fama obtenida por C++, una extensión desarrollada para el lenguaje C, en 1979. Debido a las facilidades y características aportadas por este tipo de lenguajes, es también en este período cuando se empiezan a desarrollar las primeras interfaces gráficas de usuario, como *Windows 1.0*, una GUI del año 1985 que fue lanzada para sustituir a *MS-DOS*.

La principal filosofía de este paradigma de la programación es representar en el lenguaje de programación elementos que simulen objetos, tal y como los conocemos en la vida real. Estos objetos tendrán unas características, y unas funcionalidades determinadas.

Para ilustrar esta idea, supongamos que queremos implementar un objeto en Python. Por ejemplo:

Queremos representar un cofre. Entre sus atributos podríamos encontrar: dimensiones, volumen, color, etc. También tendrá unas funcionalidades propias de un cofre: podrá abrirse, cerrarse, llenarse, vaciarse, etc.

3.3.1 Clases

Una clase es la forma que tiene Python de representar objetos. Para definir una clase se utiliza la palabra reservada *class* seguida de un nombre que identificará al objeto. Por ejemplo, para el caso del cofre que hemos descrito anteriormente:

```
class cofre():
    def __init__(self):
        self.volumen=4
        self.largo=4
        self.ancho=2

    def llenar(self):
        #Al llenarlo el volumen disminuye:
        self.volumen = self.volumen-1
```

Figura 3-4. Ejemplo de clase.

El código anterior describe una clase llamada cofre, que tiene tres variables: volumen, largo, y ancho. Estas variables se encuentran dentro de lo que se denomina "constructor" de una clase. El constructor es lo primero que se ejecutará al crear un ejemplar de la clase cofre. En Python se usa la palabra reservada *def* seguida de *__init__():* para crear un constructor. También se puede apreciar que se ha definido una función dentro de cofre, llamada *llenar()*. Esta función, es una tarea asociada a la clase cofre, es decir, es un método de la clase cofre.

Generalmente las clases tendrán un constructor, encargado de definir e inicializar los elementos y

características que compondrán el objeto, y varios métodos encargados de llevar a cabo las tareas que dichos objetos pueden realizar.

La palabra reservada *self* hace referencia al ejemplar que recibirá *cofre()* cuando intentemos crear el objeto. Es similar a *this* usado por otros lenguajes OOP. Para entenderlo mejor, vamos a instanciar una clase cofre:

```
micofre = cofre()

micofre.largo=10
micofre.llenar()
```

Figura 3-5. Instancia de la clase cofre.

Con estas líneas de código estamos instanciando la clase cofre. Instanciar es crear un objeto de una clase definida previamente, en el ejemplo, estamos formando un objeto de tipo cofre, que guardaremos en la variable *micofre*. Desde esta variable, podremos cambiar las características de *micofre*, como el largo o el ancho, o realizar una tarea llamando a uno de sus métodos.

3.3.2 Beneficios de la orientación a objetos

Una vez que se ha ilustrado al lector acerca de la lógica existente tras un lenguaje de programación orientado a objetos, se va a enumerar una lista con las principales ventajas que estos nos ofrecen:

Modularización:

Cuando se desarrollan aplicaciones complejas, es habitual dividir el problema en varias tareas de dificultad menor. Gracias a los lenguajes orientados a objetos, pueden crearse módulos que lleven a cabo tareas simples, de forma que el problema final se resuelva al juntarlos.

Para ilustrar esta idea, supongamos que queremos desarrollar un programa en Python que simule la línea de producción de una planta que se dedica a comercializar refrescos. Podríamos dividir el problema en módulos simples (departamentos):

- Recepción de materia prima
- Elaboración
- Control de calidad
- Embotellamiento
- Envío del producto

Cada uno de estos módulos podría desarrollarse usando clases de Python como las que hemos visto, con cada clase determinada por unas características (tiempos, personal, coste, prioridad, etc), y unas funcionalidades.

La modularización es muy beneficiosa a la hora de modificar, corregir o reparar, un aplicación. Una solución correctamente modularizada, puede modificarse fácilmente sustituyendo un módulo por otro, y la detección de errores se transformará en una tarea mucho mas simple.

Encapsulamiento:

A partir de la idea de modularización, es inmediato pensar en el encapsulamiento. Con este término, se suele hacer referencia a la capacidad que tienen los lenguajes OOP para ocultar lo que un objeto guarda en su interior.

En el ejemplo de la fábrica anterior, los departamentos compartirán cierta información entre ellos, pero también ocultarán otra. Elaboración, por ejemplo, no compartirá con ningún otro departamento la receta del refresco, pues esta es información confidencial, y Elaboración será el único departamento que pueda acceder a esos datos. Dicho de otra forma, la receta sería un parámetro encapsulado en el interior del departamento.

En Python, si queremos encapsular una variable o un método dentro de una clase, debemos colocar dos guiones bajos delante del nombre.

```
class cofre():
    def __init__(self):
        self.__volumen=4
        self.largo=4
        self.ancho=2

    def llenar(self):
        #Al llenarlo el volumen disminuye:
        self.__volumen = self.__volumen-1
```

Figura 3-6. Encapsulación del volumen del cofre.

Si en el ejemplo de la clase *cofre()*, encapsulamos la variable volumen, esta ya no podrá ser modificada desde el exterior. Sin embargo, sí que podría disminuirse el volumen si llamamos al método *llenar()*, ya que cambiaría el valor de la variable desde el interior.

```
micofre.__volumen="23" ❌

micofre.llenar() ✔
```

Figura 3-7. Consecuencias de la encapsulación de variables

Herencia:

Puede darse el caso de que tengamos una clase creada y lista para usarse, pero que necesitemos hacerle una ligera modificación, para que nos sea de utilidad. Por ejemplo, supongamos que tenemos una clase que representa en Python un animal. Los objetos de la clase *animal()* tienen atributos como: tamaño, género, edad, etc. Y funcionalidades como: respirar, comer, correr, etc.

¿Qué pasaría si en el código quiere implementarse un león? Sin el concepto de herencia, tendríamos que volver a declarar una clase similar a *animal()*, pero que incluya elementos y funcionalidades típicas de un león como puede ser rugir. Dando lugar a dos clases prácticamente idénticas, diferenciándose únicamente en un elemento.

Gracias a la herencia, en Python se puede volver a utilizar el código de *animal()* para hacer que la clase *leon()* herede todos los atributos y comportamientos de *animal()*, y añada exclusivamente la funcionalidad de rugir. Para hacer esto, solo debemos pasarle la clase padre (o superclase), a la clase hija (o subclase) de la siguiente forma:

```
class leon(animal):
    def rugir(self):
        #Código para rugir
```

Figura 3-8. Herencia en Python.

3.4 Librería Tkinter

Actualmente, para desarrollar una interfaz gráfica (GUI) en Python, hay numerosas librerías que pueden usarse y que están a disposición del programador: wxPython, PyGTK, o Tkinter.

En Windows, y en la mayoría de las distros de Linux, la instalación de Python viene con Tkinter instalado de serie, convirtiendo la librería en un estándar del lenguaje para el desarrollo de interfaces gráficas. Trabaja con la biblioteca Tcl/Tk, que también es usada por otros lenguajes además de Python, como Perl o Ruby.

3.4.1 Creando una GUI

En la jerga informática, todo programador que se aventura por primera vez a comprender los secretos tras un lenguaje de programación se inicia con el clásico script: "Hola Mundo!". Pues bien, a la hora de comenzar a usar Tkinter, dicho script radica en abrir una simple ventana en el escritorio. Para conseguirlo, basta con las siguientes instrucciones:

```
from Tkinter import *
window=Tk()
window.mainloop()
```

Figura 3-9. Crear una ventana con Tkinter.

La primera línea de código importa la librería Tkinter, e informa a Python de que nuestro script usará dicha biblioteca. La siguiente línea es la encargada de crear la ventana raíz usando la clase *Tk()*, y guardándola en *window*. Y la última, es en realidad, la más importante, pues es la encargada de realizar un bucle infinito que actualice constantemente la GUI.

Cuando el flujo de ejecución del programa llega al método *mainloop()*, jamás sale de allí, pues este tiene un comportamiento similar a un bucle *while(1)*, en cuyo interior sólo se ejecutan instrucciones de actualización constante de la interfaz, por lo que, sin el método *mainloop()*, la aplicación quedaría congelada, y el usuario no podría interactuar con ella.

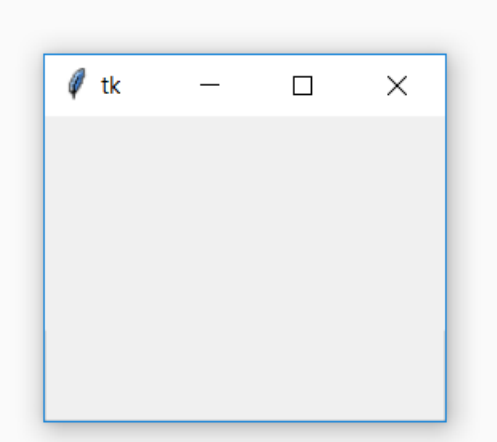

Figura 3-10. Ventana vacía.

3.4.2 Widgets

Ahora que sabemos cómo crear una ventana, sólo nos queda llenar esta con distintos tipos de elementos gráficos y funcionalidades. A estos elementos se les denomina: widgets[1]. La librería Tkinter pone a disposición del usuario varios widgets personalizables, que pueden usarse para crear GUIs de diversa índole. Tkinter utiliza la siguiente estructura para declarar un widget:

[1] La palabra widget deriva de "window-gadget". Aparato de ventana.

```
widget=Widget(contenedor, opciones)
```

Figura 3-11. Declaración general de un widget.

El contenedor hace referencia a la ventana o al frame donde se quiere situar el elemento, y las opciones dependen del widget en cuestión. En este proyecto se describirán los más importantes, y los que se han utilizado para llevar a cabo el trabajo:

Label:

Un label es un elemento estático dentro de una interfaz gráfica. Que se usa principalmente para "adornar" una interfaz usando imágenes, o para mostrar texto que no será variable. Podemos declarar un label de la siguiente forma:

```
from tkinter import *
window=Tk()

mostrar=Label(window, text="Soy un label")
mostrar.pack()

window.mainloop()
```

Figura 3-12. Declarar un label.

La variable *mostrar*, almacena en su interior un objeto de la clase label, que mostrará en pantalla el texto "Soy un label". También se hace una llamada al método *pack()*, este método coloca el widget en la ventana que tiene asociada. Es un método de posicionamiento, que trataremos con mayor profundidad en el siguiente apartado.

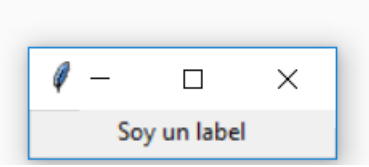

Figura 3-13. Ejemplo de label.

Si queremos mostrar una imagen, debemos cargarla primero utilizando la clase *PhotoImage()*, e indicando en las opciones que usaremos una imagen:

```
etsi=PhotoImage(file="etsilogo.png")
mostrar=Label(window, image=etsi)
mostrar.pack()
```

Figura 3-14. Otro ejemplo de label.

A continuación, se recopilan algunas de las principales opciones de configuración de la clase label:

Tabla 3-1. Opciones de un label.

Opciones	Descripción
text	Establece el texto que se mostrará.
image	Muestra una imagen.
bg	Color de fondo.
fg	Color del texto.
font	Tipo de fuente.
width	Ancho del widget.
height	Alto del widget.

Entry:

Un entry es una entrada de texto que sí puede ser modificada. Se utiliza principalmente para la comunicación con el usuario, permitiéndole ingresar y cambiar texto. Para declararlo instanciamos la clase *Entry()* de la siguiente forma:

```
texto_entry= StringVar()
mostrar=Entry(window, textvariable=texto_entry)
mostrar.pack()
```

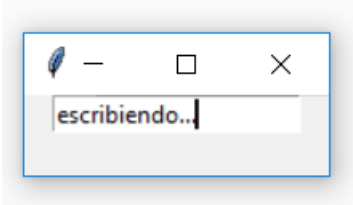

Figura 3-15. Ejemplo de entry.

Si queremos recuperar la información guardada en un entry, debemos usar una variable auxiliar que se asocie al contenido de la entrada de texto. Para ello, suele usarse la clase *StringVar()*, y usar los métodos *set()* y *get()* para modificar el interior del entry. Por ejemplo, para obtener lo que contiene, deberíamos escribir:

```
obtener_texto=texto_entry.get()
```

Figura 3-16. Cómo obtener el texto de un entry.

Al igual que ocurre con la clase *Label()*, un entry posee una lista de opciones con las que podemos configurar algunos elementos del widget. Algunas de ellas se recopilan en la siguiente tabla:

Tabla 3-2. Opciones de un entry.

Opciones	Descripción
textvariable	Asocia un *StringVar()* al contenido del entry.
bg (background)	Color de fondo.
fg (foreground)	Color del texto.
bd (borderwidth)	Grosor del borde.
font	Tipo de fuente.
width	Ancho del widget.
height	Alto del widget.

Button:

La clase *Button()* se utiliza para crear gráficamente un botón. Normalmente, los botones llevan asociados funciones que serán ejecutadas al pulsarlos. En Tkinter podemos declarar un botón de la siguiente forma:

```
def pulsado():
    print("Me has pulsado!")

mostrar=Button(window, text="Pulsa aquí", command=pulsado)
mostrar.pack()
```

Figura 3-18. Declarar un botón.

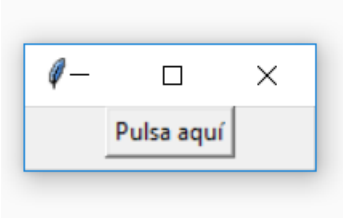

Figura 3-17. Ejemplo de botón.

Estas instrucciones, construyen un botón que escribe "Me has pulsado!" en la línea de comandos cada vez que se presiona el botón. Hay que mencionar, que en la opción *command,* se debe colocar el nombre de la función sin utilizar el operador "()". Esto se debe a que, si el intérprete lee los paréntesis, automáticamente entiende que debe ejecutar la función en ese momento, y no cuando se pulse el botón.

Pero entonces, ¿qué ocurre cuando queremos que esa función reciba algún parámetro? En ese caso estamos obligados a usar los paréntesis para indicar qué parámetro se pasa a la función, aunque podemos solucionar el problema usando el operador *lambda*:

```
def pulsado(text):
    print(text)

escribe = "Me has pulsado!"
mostrar=Button(window, text="Pulsa aquí", command=lambda: pulsado(escribe))
mostrar.pack()
```

Figura 3-19. Cómo declarar un botón cuya función recibe un parámetro.

El operador lambda se utiliza para declarar funciones anónimas. Este tipo de funciones son especiales, ya que se invocan como si toda la función se hubiese construido en la misma línea de ejecución en la que se encuentra el programa.

Como los demás widgets, la clase *Button()* también tiene un conjunto de opciones de configuración:

Tabla 3-3. Opciones de un botón.

Opciones	Descripción
text	Muestra texto sobre el botón.
command	Función llamada al presionar.
image	Cubre el botón con una imagen.
bg (background)	Color de fondo.
fg (foreground)	Color del texto.
bd (borderwidth)	Grosor del borde.
font	Tipo de fuente.
width	Ancho del widget.
height	Alto del widget.

Frame:

Un frame es un contenedor donde podemos insertar widgets. Como la clase *Frame()* funciona igual que un widget, podemos tener un frame insertado en otro frame, o en la ventana raíz. Para ilustrar la idea, representaremos el efecto Droste[2], usando frames de Tkinter:

```
uno=Frame(window, bg="blue",width=100,height=100)
uno.place(x=50,y=50)

dos=Frame(uno, bg="red",width=50,height=50)
dos.place(x=25,y=25)

tres=Frame(dos, bg="green",width=25,height=25)
tres.place(x=12,y=12)
```

Figura 3-21. Declaración de varios frames.

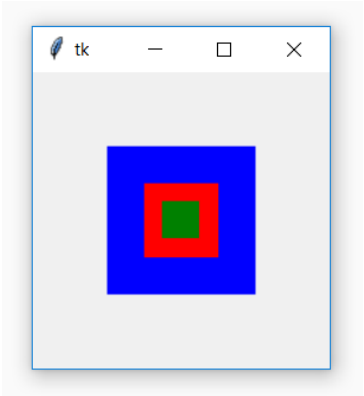

Figura 3-20. Efecto Droste con frames.

[2] El efecto Droste es la imagen que se obtiene del hecho enfrentar dos espejos.

Para llevarlo a cabo se ha usado *place()*, que es un método de posicionamiento de widgets y elementos de Tkinter, similar a *pack()*. En el siguiente apartado, veremos los principales tipos de métodos de posicionamiento.

3.4.3 Posicionamiento

Para colocar los widgets dentro de un contenedor, ya sea un frame o una ventana, existen varios métodos disponibles. Si no se posicionan haciendo uso de estos, los widgets no aparecerán en la interfaz, aunque estén declarados.

Pack:

Este método es ideal para interfaces simples, como cuadros de diálogo, ventanas emergentes de notificación, o formularios. El método coloca los widgets de forma que queden empaquetados, y luego adapta el contenedor al tamaño total que ocupan los widgets. Cuando se utiliza *pack()*, al igual que ocurría con los widgets, debemos utilizar las opciones de configuración que el método tiene asociadas:

Tabla 3-4. Opciones de *pack()*.

Opciones	Descripción
anchor	Lugar del widget usando: centro, derecha, izquierda, arriba o abajo.
expand	Expande el widget hasta ocupar el espacio disponible.
fill	Cubre todo el contenedor con el widget: horizontal, verticalmente o ambas.
padx	Margen externo horizontal.
pady	Margen externo vertical.
ipadx	Margen interno horizontal.
ipady	Margen interno vertical.

Place:

A diferencia de *pack()*, este método coloca los widgets usando coordenadas absolutas medidas en píxeles. Aunque puede llegar a ser tedioso, es el método que mejores resultados suele ofrecer. Es ideal para GUIs elaboradas en las que todo tiene que estar en un lugar bien especificado. Algunas de sus opciones principales son:

Tabla 3-5. Opciones de *place()*.

Opciones	Descripción
x	Posición horizontal de la esquina superior derecha del widget en píxeles.
y	Posición vertical de la esquina superior derecha del widget en píxeles.
relx	Posición horizontal de la esquina superior derecha del widget en p.u. (por unidad)
rely	Posición vertical de la esquina superior derecha del widget en p.u.
height	Alto del widget en píxeles
width	Ancho del widget en píxeles
relheight	Alto del widget en p.u.
relwidth	Ancho del widget en p.u.

Grid:

Esta forma de posicionar los elementos de una interfaz, se basa en fraccionar el contenedor en espacios que luego pueden ser ocupados por los widgets. Forma una matriz de huecos donde los elementos pueden ser colocados. Sus opciones básicas son:

Tabla 3-6. Opciones de *grid()*.

Opciones	Descripción
row	Fila que ocupará el widget.
column	Columnaque ocupará el widget.
rowspan	Número de filas que abarca el widget.
columnspan	Número de columnas que abarca el widget.
sticky	Dirección a la que aproximar el widget: Norte, sur, este u oeste.
padx	Margen externo horizontal.
pady	Margen externo vertical.
ipadx	Margen interno horizontal.
ipady	Margen interno vertical.

3.4.4 Eventos

Puede que la interfaz gráfica que estemos desarrollando necesite ejecutar ciertas tareas cuando tiene lugar un evento determinado. De forma similar al comportamiento que tenemos cuando se pulsa un botón.

En algunos casos, podríamos querer detectar en nuestra interfaz en qué punto de la ventana se ha pulsado para mostrar allí una imagen. O también podríamos necesitar saber cuándo se ha pulsado una tecla en el teclado del ordenador, y actuar en consecuencia.

Por ejemplo: realizar una interfaz gráfica que salga de la aplicación cuando se pulsa la tecla *Esc*.

Para realizar este tipo de tareas, todos los widgets de Tkinter pueden hacer uso del método *bind()*. Con él, podemos asociar un evento determinado a un widget de la aplicación, y actuar en consecuencia. La forma de utilizar este método es la siguiente:

```
widget.bind(evento, funcion_evento)
```

Figura 3-22. Declaración general de un evento.

La instrucción de arriba, asocia a la variable *widget*, la tarea *funcion_evento()*, que se realizará siempre que tenga lugar el evento definido en *evento*.

Las variables como *evento* son cadenas de caracteres con un formato determinado. Estas cadenas, dan información acerca del evento que se quiere registrar. En la siguiente tabla se muestran algunos de los formatos más usados:

Tabla 3-7. Lista de eventos.

Eventos	Descripción
<Button-1>	Click izquierdo del ratón (click derecho = <Button-2>).
<Double-Button-1>	Doble click izquierdo.
<Enter>	El cursor entra y se posiciona sobre el widget.
<Leave>	El cursor sale del widget.
<Key>	Se ha pulsado alguna tecla.
<B1-Motion>	Se arrastra el cursor con el botón izquierdo pulsado.
<ButtonRelease-1>	Guarda las posiciones del cursor.
<Configure>	Detecta si el widget cambia su tamaño o forma.
<FocusIn>	Detecta si el objetivo del teclado es el widget.

Todos estos eventos, tienen asociados unos atributos, que almacenan cierta información en función del tipo de evento que se quiera registrar:

Tabla 3-8. Lista de atributos de los eventos.

Atributo	Descripción
widget	Indica widget que generó el evento.
x, y	Posición del cursor en píxeles, en función del widget.
x_root, y_root	Posición del cursor en píxeles, en función de la ventana raíz.
char	Cadena de caracteres con el valor de la tecla. Sólo para eventos de teclado.
num	El número identificativo del botón. 1- Botón izquierdo. 2- Derecho. 3- Rueda.
width, height	Nuevo tamaño de widget. Sólo evento <Configure>.
type	Tipo del evento.

Por ejemplo, imaginemos que estamos desarrollando una aplicación gráfica que permite al usuario colocar en un lienzo, formas geométricas, ya sean círculos, triángulos, o incluso imágenes.

Esto puede llevarse a cabo creando un frame que represente el lienzo de la aplicación, y a continuación, asociarle como evento el botón izquierdo del ratón, de forma que podamos detectar cuándo estamos pulsando sobre el frame.

Luego, en la función que manejará el evento, se creará un label con la imagen que queramos colocar en el lienzo. Un programa de estas características puede llevarse a cabo de la siguiente manera:

```
from tkinter import *

def mostrar_imagen(evento):
    posx=evento.x #Posicion x donde se hizo click
    posy=evento.y #Posicion y donde se hizo click

    Label(lienzo, image=etsi).place(x=posx,y=posy)
```

```
window=Tk()
window.config(bg="grey")
window.geometry("500x500")
window.resizable(0,0)

#Cargamos la imagen que vamos a usar:
etsi=PhotoImage(file="etsilogo.png")


lienzo=Frame(window,width=400, height=400)
lienzo.place(x=50,y=50)

#Button-1 es el evento asociado al botón izquierdo del ratón
lienzo.bind("<Button-1>", mostrar_imagen)



window.mainloop()
```

Figura 3-23. Ejemplo de un programa usando eventos.

El ejemplo anterior, crea un lienzo en una ventana, e inserta el logotipo de la escuela cada vez que pulsamos en él. En la siguiente figura veremos el resultado cuando llevamos a cabo su ejecución:

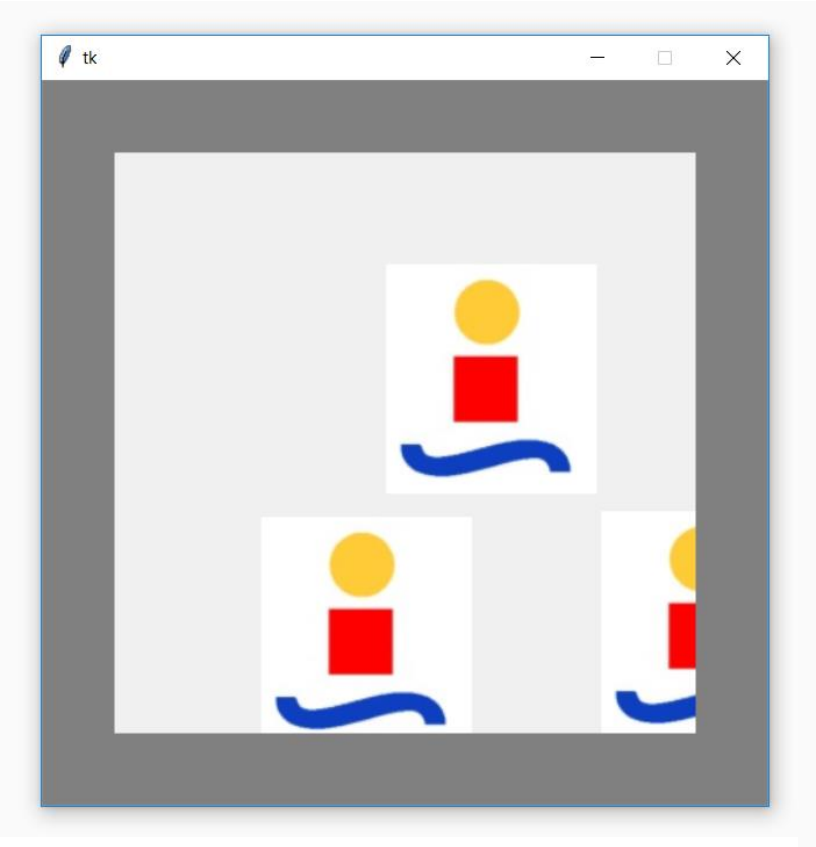

Figura 3-24. Ejecución del programa anterior.

3.4.5 Cómo ejecutar una función en background

Ahora que conocemos gran parte del funcionamiento de una interfaz gráfica desarrollada con Tkinter, es posible que el lector esté preguntándose, cómo se puede llevar a cabo una función que se ejecute simultáneamente con la interfaz, de forma que no la interrumpa.

Como sabemos, una interfaz programada con la librería Tkinter, no es más que un conjunto de declaraciones de widgets, colocados en una ventana, que se actualizan constantemente a través de un bucle infinito al que el código accede a través del método *mainloop()*.

Pues bien, en el momento en que el flujo del programa ingresa a dicho bucle, esta a merced de la interfaz, lo cual quiere decir que solo ejecutará funciones, clases, y código que sea llamado a través de los elementos de la interfaz, como pueden ser los botones, o los eventos. Todo aquello que se encuentre debajo de la instrucción *mainloop()* nunca se ejecutará, pues el flujo del programa nunca llegará allí.

Una posible solución, sería dividir el programa en dos hilos de ejecución en paralelo. Uno de ellos llevaría a cabo las labores de declaración y de actualización de la interfaz, y el otro se dedicaría a operar la tarea en background y luego actualizar las variables necesarias. Pues bien, esto no es compatible con Tkinter, ya que es una librería no *thread-safe*.

Para solucionar el problema, la librería pone a disposición un método especial que permite al usuario llamar a una función cada cierto tiempo, en función de lo definido por el programador. Hablamos del método *after()* de Tkinter, que podríamos usar de la siguiente forma:

```
from tkinter import *
import random

window=Tk()

def cambiar_color():
    color=random.choice(["green","blue","red","black","white"])
    mostrar.config(bg=color)

    window.after(1000,cambiar_color)


mostrar=Frame(window, bg="blue",width=200,height=200)
mostrar.pack()

cambiar_color()

window.mainloop()
```

Figura 3-25. Programa que cambia el color de un frame aleatoriamente.

Como vemos, el programa anterior dispone de una función llamada *cambiar_color()* que se dedica a cambiar el color de un frame que ya hemos inicializado. Si omitiésemos la línea que hace uso del método *after()*. Sólo se cambiaría el color una vez, justo antes de entrar al *mainloop()*.

Sin embargo, tal y como esta programado el script, cuando el flujo de programa ejecuta por primera vez la función *cambiar_color()*, esta queda programada para ejecutarse de nuevo 1000 milisegundos más tarde, gracias al método *after()*.

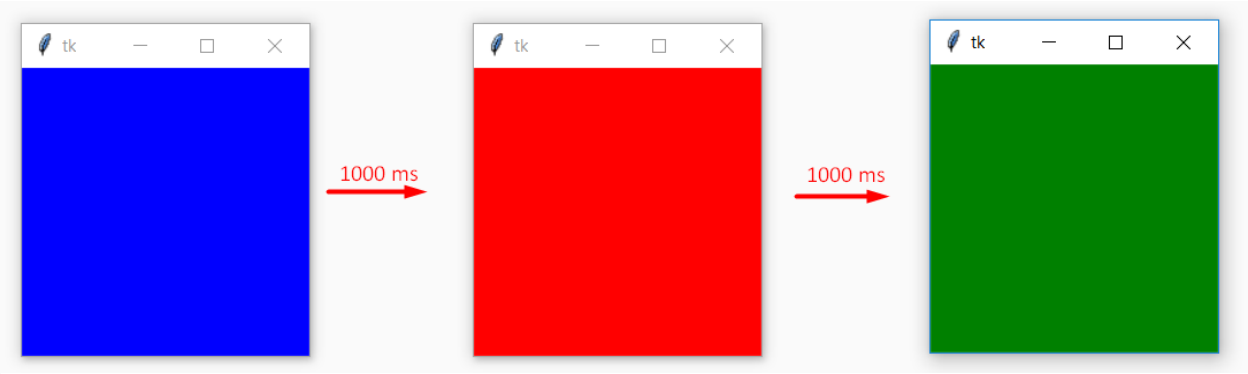

Figura 3-26. Funcionamiento del ejemplo anterior.

De esta forma, podemos llevar a cabo de una forma sencilla tareas en background, que no se ejecutan al pulsar ningún botón, o gracias a ningún evento. Además, este método nunca interrumpirá tareas de actualización llevadas a cabo por el bucle *mainloop()*, ya que el programa sólo entrará a ejecutar estas funciones una vez que el bucle de actualización se encuentre liberado para llevar a cabo las tareas.

Esto, como es lógico, provoca que el tiempo configurado por el usuario para repetir la función, sea en realidad el tiempo mínimo en el que la función volverá a repetirse. En nuestro ejemplo, 1000 ms es el tiempo mínimo que deseamos esperar entre llamada y llamada, por lo que habrá veces que el intervalo sea de 1002 ms, o de 1007 ms en otra.

Por lo tanto, no es recomendable usar esta solución para implementar aplicaciones con especificaciones de tiempo muy restringidas. Para este tipo de tareas, sí regresaríamos a la idea original de usar varios hilos. De forma que se llevaría a cabo la realización de colas que compartan información entre el hilo de la interfaz, y los hilos de las tareas en background. Estas colas sí son *thread-safe*, por lo que podríamos usar el método *after()* sólo para obtener o modificar la información de dichas colas, haciendo que estas funcionen de intermediario entre los hilos.

En definitiva, debemos tener cuidado si vamos a usar este método para llevar a cabo en su interior tareas muy laboriosas, puesto que tal y como está planteado su funcionamiento, los resultados no siempre son los deseados. Si las tareas incluidas en este tipo de funciones tardan mucho en regresar al bucle de actualización de la interfaz, esta no tendrá más remedio que congelarse momentáneamente.

3.4.6 Niveles superiores

Para finalizar, veremos cómo crear ventanas nuevas a partir de otras ya existentes. Hasta ahora, todos los ejemplos utilizaban una sola ventana raíz para llevar a cabo sus diversas funciones. La forma de fabricar dicha ventana la vimos en el apartado 3.4.1.

Tkinter utiliza una clase distinta de *Tk()* para crear ventanas de un orden superior a la raíz, llamada *Toplevel()*. Estas ventanas superiores estarán asociadas a otras ventanas "padres", de forma que, si se destruye la raíz, caerán todas y cada una de las ventanas "hijas" asociadas.

Para ilustrarlo, se ha creado un programa en Tkinter que construye una ventana superior a partir de una raíz, cada vez que se pulsa un botón.

También se ha utilizado el método *destroy()*, para añadir al programa una forma alternativa de destruir las ventanas diferente del botón "cerrar" que ofrecen los S.O.

```
from tkinter import *

def construir_toplevel():
    hijo=Toplevel(raiz)

    Label(hijo, text="Soy un hijo").pack()
    salir=Button(hijo,text="Destruir",command=hijo.destroy).pack()

raiz=Tk()
raiz.config(bg="grey")

boton=Button(raiz,text="Crear hijo",command=construir_toplevel).pack()

raiz.mainloop()
```

Figura 3-28. Interfaz capaz de crear niveles superiores.

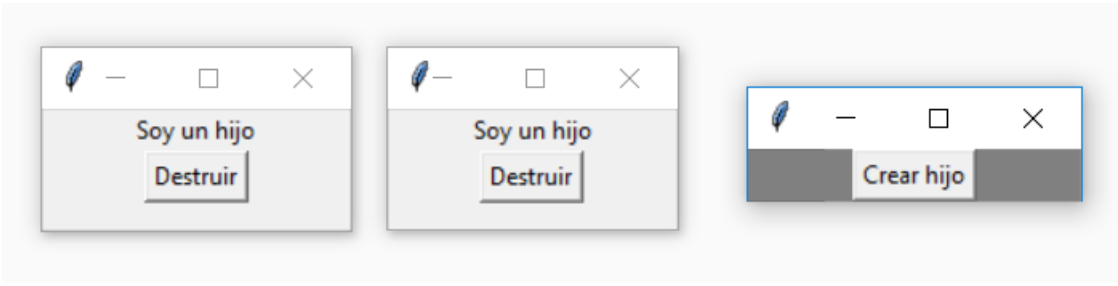

Figura 3-27. Ejecución del ejemplo anterior.

3.5 Librería PyVISA

Como adelantábamos en el capítulo 2, PyVISA es una librería disponible para el lenguaje de programación Python, que nos permite llevar a cabo comunicaciones con todo tipo de dispositivos de medida independientemente de la interfaz que utilicen.

3.5.1 Inicialización

Para utilizar la librería, lo primero que debemos hacer es importarla, y luego instanciar una clase de tipo *ResourceManager()*:

```
import pyvisa as visa

rm=visa.ResourceManager()
#rm=visa.ResourceManager("@py")
```

Figura 3-29. Inicializando PyVisa.

Si lo llamamos sin argumentos, PyVISA entenderá que quiere usarse el backend[3] por defecto: NI. Sin

[3] En diseño de software el front-end es la parte del software que interactúa con los usuarios y el back-end es la parte que procesa la entrada desde el front-end.

embargo, suelen utilizarse dos backends principalmente: El backend por defecto de PyVISA, que utiliza la librería NI, o el backend proporcionado por PyVISA-py, que es una implementación dentro de Python de la librería VISA, incluyendo "@py" como argumento de *ResourceManager()*.

Una vez inicializado el *ResourceManager()* podemos utilizar la librería PyVISA para crear una lista de los recursos o dispositivos a los que tenemos acceso:

```
import pyvisa as visa

rm=visa.ResourceManager()

print(rm.list_resources())
```

Figura 3-30. Crear una lista de recursos.

3.5.2 Establecer conexión con un dispositivo

Una vez que conocemos la dirección y el tipo de dispositivo al que vamos a conectarnos, podremos utilizar el método *open_resource()* para establecer una conexión entre el código y el dispositivo, que podremos usar para enviar mensajes:

```
import pyvisa as visa

rm=visa.ResourceManager()

modulo=rm.open_resource("TCPIP0::192.168.30.174::gpib0,3,1")
```

Figura 3-31. Establecer comunicación con un dispositivo.

El argumento que pasamos a *open_resource()* es una cadena de caracteres que aporta información acerca del dispositivo, y que representa la dirección del mismo. Estas cadenas se obtienen normalmente usando el método *list_resources()* primero.

Esta cadena de caracteres depende de la interfaz que este usando el dispositivo. En la siguiente tabla se recopila el tipo de nomenclatura que pueden presentar. Los elementos entre paréntesis son opcionales:

Tabla 3-9. Direcciones en función de la interfaz usada por el dispositivo.

Interfaz	Sintaxis de la cadena
ENET-Serial INSTR	ASRL[0]::host address::serial port::INSTR
GPIB INSTR	GPIB[board]::primary address[::secondary address][::INSTR]
GPIB INTFC	GPIB[board]::INTFC
PXI BACKPLANE	PXI[interface]::chassis number::BACKPLANE
PXI INSTR	PXI[bus]::device[::function][::INSTR]
PXI INSTR	PXI[interface]::bus-device[.function][::INSTR]
PXI INSTR	PXI[interface]::CHASSISchassis number::SLOTslot number[::FUNCfunction][::INSTR]

PXI MEMACC	PXI[interface]::MEMACC
Remote NI-VISA	visa://host address[:server port]/remote resource
Serial INSTR	ASRLboard[::INSTR]
TCPIP INSTR	TCPIP[board]::host address[::LAN device name][::INSTR]
TCPIP SOCKET	TCPIP[board]::host address::port::SOCKET
USB INSTR	USB[board]::manufacturer ID::model code::serial number[::USB interface number][::INSTR]
USB RAW	USB[board]::manufacturer ID::model code::serial number[::USB interface number]::RAW
VXI BACKPLANE	VXI[board][::VXI logical address]::BACKPLANE
VXI INSTR	VXI[board]::VXI logical address[::INSTR]
VXI MEMACC	VXI[board]::MEMACC
VXI SERVANT	VXI[board]::SERVANT

En el ejemplo anterior, se ha utilizado la dirección IP: 192.168.30.174. En esta dirección se encuentran conectadas una pareja de mainframes.

En el segundo capítulo de este documento, se habló de que los mainframes estaban conectados a un gateway que convertía la interfaz GPIB a Ethernet. Por lo tanto, cuando llamamos al método *open_resource()* y le pasamos la cadena: "TCPIP0::192.168.30.174::gpib0,3,1", se establece una conexión vía Ethernet a dicha dirección IP, que a su vez, también puede entenderse como una dirección GPIB una vez se cruza el gateway.

Más allá de la puerta de enlace (gateway), estamos por lo tanto, situados en una dirección GPIB que contiene dos mainframes, con ocho módulos cada una. Si queremos conectarnos sólo a una de ellas, debemos seguir avanzando a través del GPIB, por ello la cadena sigue con: gpib0,3,1.

El primer número hacer referencia al [board], que la mayoría de veces se queda por defecto con un valor de cero, como ocurre al usar “TCPIP0” en la cadena.

El segundo número indica a qué mainframe vamos a conectarnos, que debido a cómo ha llevado a cabo la configuración Alter Technology, puede ser: un 1 para la mainframe superior, o 3, para la mainframe inferior.

Y el último, puede variar de 0 a 7, e indica el módulo concreto al cuál podemos enviar comandos SCPI (de ahí el nombre dado a la variable *modulo*).

Write

Es un método que se utiliza para enviar al dispositivo mensajes. Para el caso del módulo del ejemplo, podríamos usar:

```
modulo.write("OUTP:STAT OFF")
```

Figura 3-32. Ejemplo de write.

Este mensaje envía el siguiente comando SCPI: “OUTP:STAT OFF”, que indica al módulo que inhabilite su salida.

Read

Funciona igual que *write()*, pero sirve para recibir mensajes del dispositivo. Ejemplo:

```
modulo.read()
```

Figura 3-33. Ejemplo de read.

Query

Es el equivalente a realizar un *write()* seguido de un *read()*. Envía un mensaje al dispositivo, y permanece a la espera de respuesta. En el siguiente ejemplo, le pedimos al módulo que nos responda cuál es el voltaje medido en su salida:

```
modulo.query("MEAS:VOLT?")
```

Figura 3-34. Ejemplo de query.

Esta instrucción podría sustituirse por la siguiente:

```
modulo.write("MEAS:VOLT?")
modulo.read()
```

Figura 3-35. Equivalente a un query.

4 DESARROLLO DE LA APLICACIÓN

En este capítulo se profundizará en la interfaz gráfica confeccionada en este proyecto final. Para ello se informará al lector acerca del funcionamiento de la aplicación, y posteriormente, se realizará un estudio del código fuente empleado, haciendo especial énfasis en los apartados convenientes.

4.1 Funcionamiento

En este apartado no se pretende llevar a cabo un manual de instrucciones sobre la forma en que trabaja la interfaz, más bien, se intenta dar a conocer el funcionamiento básico de la aplicación, y de algunas de sus características principales:

4.1.1 Pantalla de inicio

Una vez abierta, veremos dos mainframes actualizando sus valores constantemente. Los datos se actualizan en cascada desde los módulos de la izquierda hasta los módulos de la derecha, con una duración total menor de 10 segundos. La aplicación nos dará la posibilidad de editar cualquiera de las dos unidades de módulos, o de salir de la aplicación.

Figura 4-1. Pantalla principal.

La mayor parte del tiempo la aplicación se encontrará en este primer nivel[4], informando al usuario del estado de los módulos. Cuando se necesite llevar a cabo cambios en alguno de ellos, el usuario podrá pulsar en uno de los dos botones *Editar* y la aplicación lo llevará al segundo nivel: el editor.

4.1.2 Editor de módulos

En este nivel, la ventana luce de forma similar a como lo hacía la pantalla principal, ya que muestra en la zona superior un mainframe como los que se han usado allí, sin embargo, el elemento de la zona inferior, aunque parece un mainframe, no lo es, ya que es un apartado gráfico creado para modificar los módulos contenidos en la mainframe superior.

[4] El primer nivel hace referencia a la ventana raíz de la aplicación, siendo los niveles siguientes hijos de este.

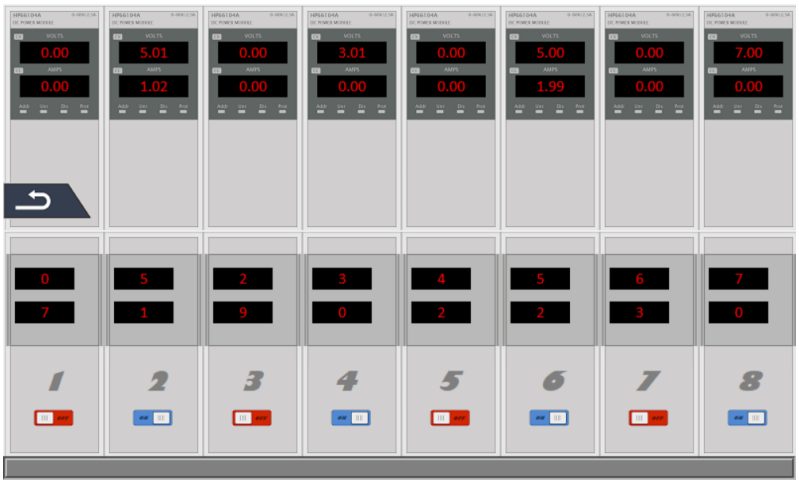

Figura 4-2. Editor de módulos.

Previamente, en el primer nivel, se eligió cuál de las dos mainframes presentadas iba a ser modificada (usando los botones *Editar*). En este segundo nivel la interfaz crea una copia en tiempo real de la mainframe elegida, y la sitúa en la zona superior. De forma que el usuario pueda visualizar la mainframe a la vez que esta se modifica.

El editor se construirá utilizando un botón On/Off para cada módulo, y dos displays. El botón habilitará o deshabilitará la salida según su estado, mientras que los displays mostrarán a qué nivel se pondrá la salida en caso de que esta esté habilitada.

Por lo tanto, la edición siempre se llevará a cabo en dos pasos, de forma que, para establecer un voltaje de salida, se procedería de la siguiente forma:

- **Paso 1**. Establecer el nivel de salida deseado, y que aparezca en el display.
- **Paso 2**. Poner el switch en On, esto hará que se ponga la salida al nivel indicado en el display.

El funcionamiento es simple, si la salida está On, el módulo establecerá como valores de tensión y corriente los indicados por los displays. Si está Off, la salida será cero.

Como la aplicación está preparada para funcionar en una pantalla táctil, para escribir en los displays (o los entry) es necesario que aparezca un teclado auxiliar cada vez que toquemos en ellos.

Para que al momento de aparecer no dificulte la edición, cuando pulsamos en un entry de uno de los cuatro primeros módulos, el teclado se situará sobre los cuatro últimos, y viceversa.

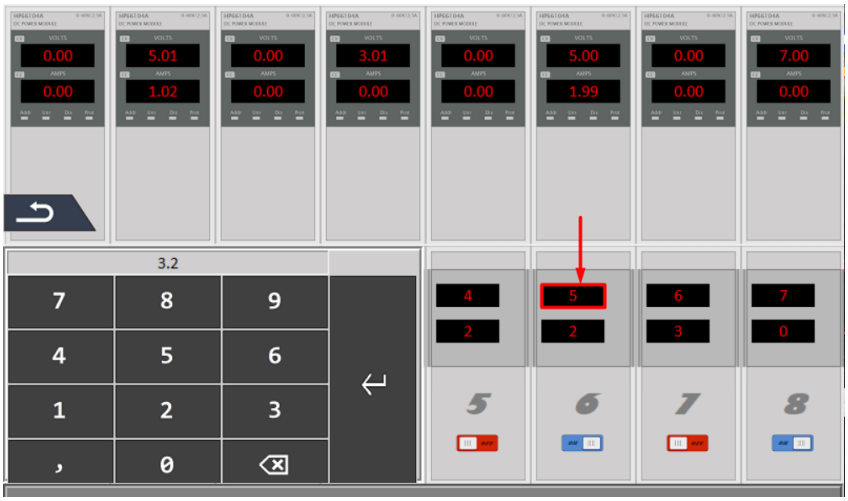

Figura 4-3. Posición del teclado al tocar el entry indicado en rojo.

Como veremos en los próximos apartados de este capítulo, a efectos de código, esta operación para crear el teclado se ha llevado a cabo creando una nueva ventana (nivel tres).

4.2 Código Python

Ahora hablaremos de cómo se ha podido implementar la interfaz, analizando los puntos clave del código en Python. Se realizará una breve descripción de los objetos que se han definido, y que conforman la GUI, como se han usado, y sus principales características.

Puede que, durante los próximos apartados, el lector necesite consultar el código total de la aplicación, por lo que es conveniente recordar que está disponible en el anexo de este documento.

4.2.1 Primer nivel (pantalla de inicio)

Empezaremos creando el primer nivel de la interfaz. Si recordamos lo que se dijo en el apartado anterior, la primera ventana de nuestra GUI, mostrará dos unidades de módulos (dos mainframes), que deben actualizarse en concordancia con la unidad física real.

Para llevar a cabo la tarea de una forma ordenada, vamos a crear una clase en Python que represente a una mainframe, de forma que lo único que tengamos que hacer desde el flujo principal del programa sea crear dos ejemplares de dicha clase.

4.2.1.1 Clase: *mainframe()*

En resumen, el trabajo a realizar consiste en llevar a cabo la declaración de una clase que represente gráficamente a una mainframe (o MPS). A dicha clase se le ha llamado: *mainframe()*, y es uno de los elementos fundamentales que conformarán nuestra interfaz gráfica.

La misión de *mainframe()* es crear gráficamente una unidad de módulos, que cuente con entradas de texto (entry) que se actualicen constantemente en función de los datos de la mainframe real. Su constructor, posee cuatro parámetros de entrada: *master, posx, posy* y *dir_mod*:

- **master:** Será la ventana de Tkinter donde se va a construir la mainframe. En este caso, la ventana de la pantalla principal, contenida en la variable *window*.
- **posx:** Posición *x* de la esquina superior izquierda de la mainframe en la interfaz.
- **posy:** Posición *y* de la esquina superior izquierda de la mainframe en la interfaz.
- **dir_mod:** Lista 1x8 de direcciones VISA. Apuntan a los módulos de la mainframe física.

En su interior, cuenta con una variable consistente en una matriz 2x8 de identificadores, a la cual se ha llamado *identificadores*. Cada identificador tendrá vinculado uno de los displays de voltaje y corriente de los módulos. Para crear el identificador se ha usado una cadena que contiene una letra[5] y un número.

De esta forma, "*v2"* haría referencia al display de voltaje del módulo tres (ya que se empieza a contar desde el cero), mientras que "*c4"* se correspondería con la entrada de corriente del módulo cuatro.

[5] Letra v para el voltaje, y c para la corriente.

Figura 4-4. Mainframe mostrando el identificador correspondiente de cada display.

Estos identificadores se comparten a varias funciones frecuentemente, pero son usados principalmente por dos diccionarios creados dentro de la clase *mainframe()*:

- **entry:** Que estará a cargo de los displays, tanto de voltaje como de intensidad. Y asociará a cada display una variable de tipo *StringVar()*. Para saber a qué modulo se está enlazando las cadenas de caracteres se utiliza la lista *identificadores,* de forma que: dado un identificador, se recibirá una *StringVar()* asociada a la entry correspondiente. Por ejemplo, para modificar el voltaje del módulo 5 se haría lo siguiente:

```
self.entry["v4"].set("5.00")
```

Figura 4-5. Formato de escritura en un objeto de tipo *mainframe()*.

- **modulo:** Con este diccionario, podremos solicitar información a la MPS real. De forma similar a como se explicó durante el capítulo 3.5. Por ejemplo, para medir el voltaje de salida del módulo 5 (que en Python sería el 4, ya que se empieza a contar desde cero) se haría lo siguiente:

```
volt_salida=self.modulo[4].query("MEAS:VOLT?")
```

Figura 4-6. Cómo comunicarse con el módulo real.

Método: *actualizar()*

El objetivo de *mainframe()* es poder actualizar constantemente los displays de voltaje y corriente de sus módulos. Para ello, hace uso de su método interno: *actualizar(),* el cual requiere de los parámetros: *master*, y *col*. Y que será llamado por primera vez desde el constructor de la clase *mainframe()*, de la siguiente forma:

```
self.actualizar(master,0)
```

Figura 4-7. Primera llamada a *actualizar()* desde *mainframe()*.

Por lo tanto, cada vez que se llame a la función *actualizar()*, esta debe enviar una petición a la mainframe

física, y pedirle el dato de una de sus salidas, ya sea de voltaje, o de corriente. A continuación, se realiza una breve descripción de los parámetros de la función:

- **master:** Es la ventana de Tkinter donde se ha construido la mainframe. Es necesario que *actualizar()* la reciba, para poder utilizar el método *after()*, que se explicó en el capítulo 3.
- **col:** Hace referencia a la columna en la que se encuentra el módulo que queremos actualizar.

En función de los parámetros que se le han pasado, *actualizar()* preguntará a uno de los módulos acerca de su tensión, o su corriente, mediante el método *query()* de la librería PyVISA. Por ejemplo, si se quiere preguntar por el voltaje:

```
respuesta=self.modulo[col].query("MEAS:VOLT?")
respuesta="{:3.2f}".format(float(respuesta))
```

Figura 4-8. Uso de *query()* dentro de *actualizar()*.

La información recibida, se guarda en la variable *respuesta*, y luego se modifica en la siguiente instrucción para poder incluirla al entry correspondiente, de forma que sólo tenga dos cifras decimales.

Una vez se actualiza el entry correspondiente, la función se llamará a sí misma en intervalos de 1000 ms, mediante la siguiente instrucción:

```
master.after(1000,self.actualizar,master,col+1)
```

Figura 4-9. Uso de *after()* para ejecutar *actualizar()* en background.

Esta línea de código se encuentra al final de la función *actualizar()*, y es una instrucción que hace uso del método *after()*, propio de la biblioteca Tkinter, y del que se habló en el tercer capítulo. Además, a la hora de dejar programada la próxima llamada a la función, se utilizarán los siguientes parámetros: *master*, y *col+1*.

Esto quiere decir que, como el método debe preparar los parámetros que se pasarán la próxima vez, dentro del mismo deben realizarse tareas de preparación para dichas variables. Como es el caso de la variable *col*, la cuál hay que resetear cuando llega a la última columna de módulos:

```
if col==len(self.identificadores[1]):
    col=0
```

Figura 4-10. Preparación de *col* para poder ser utilizada dentro de *actualizar()*.

En resumen, gracias al método *actualizar()*, podremos actualizar continuamente la mainframe recorriendo todos los módulos. El flujo de actualización del método presenta un trazado como el que se muestra en la siguiente figura:

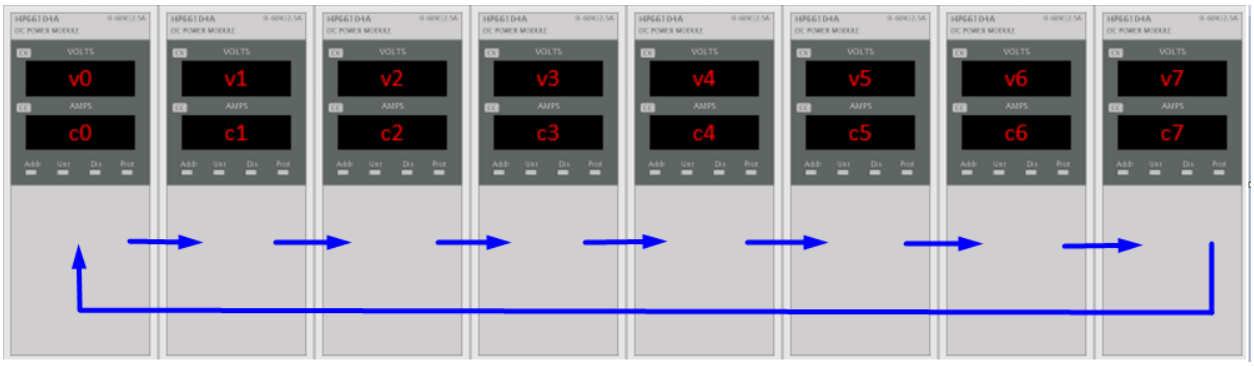

Figura 4-11. Flujo de actualización.

4.2.1.2 Otros elementos:

Vamos a describir ahora algunos de los elementos que son utilizados también por la pantalla principal (primer nivel):

Clase: *b_edit()*

Este tipo de clase surge de la necesidad de crear botones que transporten la GUI desde la pantalla principal, a la pantalla de edición. En el apartado anterior, a estos se les denominó botones *Editar*.

Su construcción es simple, y se puede dividir su funcionamiento en dos partes:

1. La declaración de un botón utilizando la librería Tkinter.
2. La construcción de un método asociado a dicho botón que transporte la GUI desde el primer nivel al segundo.

Figura 4-12. Botón *Editar*.

La clase *b_edit()* recibe los siguientes parámetros:

- **master:** Es la ventana de Tkinter donde se va a construir el botón, y en la que se encuentra el mainframe. Como es la pantalla principal, se recibe *window*.
- **mainframe:** Es el objeto de tipo *mainframe()* que querremos editar una vez que pasemos al nivel siguiente de la interfaz.

La declaración del botón es algo que ya se ha visto en el capítulo anterior, por lo que pasaremos a describir el método que tiene asociado: *go()*.

Como vemos en la siguiente figura, *go()* realiza dos operaciones: declara una nueva ventana a partir de master, y construye el segundo nivel utilizando la clases *editor_mainframe()*, *b_salir()*, y la función *barra()*.

```
def go(self,master,mainframe):

    window2=Toplevel(master)          #A partir de la ventana en la que estamos, creamos un nivel superior.
    window2.overrideredirect(1)       #Pantalla completa
    window2.geometry("800x480+0+0") #Tamaño y colocación en la pantalla.
    window2.resizable(0,0)            #No podremos modificar las dimensiones de la ventana window2.

    #CONSTRUCCION 2ºNIVEL:
    m3=editor_mainframe(window2, mainframe, 0, 0)   #Construimos un editor de mainframe.
    barra(window2)                                  #Llenamos el hueco inferior.
    b_salir(window2,m3,i_atras)                     #Construimos un boton para regresar al primer nivel.
```

Figura 4-13. Método *go()* para llegar al nivel de edición.

Clase: *b_salir()*:

Al igual que *b_edit()*, se utiliza para representar un botón. En este caso con el funcionamiento contrario. La clase *b_salir()*, será utilizada para transportar la GUI de los niveles superiores a los inferiores. Más concretamente, podrá transportar al usuario desde el segundo hasta el primer nivel (botón *Atrás*), o desde el primer nivel a salir por completo de la GUI (botón *Salir*). Recibe como parámetros:

- **master:** Es la ventana de Tkinter de la que se quiere salir. Pueden ser *window* para salir de la GUI, o *window2* para cerrar el editor y volver a la pantalla principal.
- **elemento:** Es un elemento gráfico cuyo frame interno usaremos para construir el botón. Pueden ser objetos de tipo *mainframe()* y *editor_mainframe()*.
- **foto:** Contiene la imagen de fondo, que es diferente si se implementa el botón *Salir* o *Atrás*.

Como es lógico, el *command* asociado al botón será: *master.destroy()*.

Figura 4-14. Botón *Salir*.

Figura 4-15. Botón *Atrás*.

Función: *barra()*:

Esta función ejecuta una serie de instrucciones de la librería Tkinter para rellenar una parte de la pantalla que no es ocupada por la interfaz. La ejecutan tanto la pantalla principal como la pantalla de edición.

Figura 4-16. Elemento aportado por *barra()*, marcado en azul.

Con todas estas clases y métodos esto sobre la mesa, para crear la pantalla principal solo seran necesarias las siguientes instrucciones:

```
#Construimos una mainframe en la zona superior de la ventana:
m1=mainframe(window, 0, 0, direcciones1)

#Construimos una mainframe en la zona inferior de la ventana:
m2=mainframe(window, 0, 228, direcciones2)

#Rellenamos el resto:
barra(window)

#Boton para salir de la aplicación:
b_salir(window,m2,i_exit)

#Boton Editar para m1 (mainframe superior)
b_edit(window,m1)

#Boton Editar para m2 (mainframe inferior).
b_edit(window,m2)
```

Figura 4-17. Construcción de la pantalla principal.

4.2.2 Segundo nivel (editor de módulos)

En este punto, hemos conseguido crear en su totalidad la pantalla principal (el primer nivel), que consiste en llamar dos veces a la clase *mainframe()*. Una vez tocamos en uno de los botones *Editar,* la interfaz mostrará una pantalla nueva: el editor de módulos.

En este segundo nivel se debe poder visualizar una copia de la mainframe que hayamos seleccionado, y un display que nos permita editar y modificar el voltaje o la corriente de sus módulos. Para llevar a cabo esta misión haremos uso de una nueva clase: *editor_mainframe()*.

4.2.2.1 Clase: *editor_mainframe()*

El trabajo de *editor_mainframe()* es realizar una copia de uno de los MPS creados en el nivel uno, mostrando en sus entradas de voltaje y corriente los mismos valores, y además, crear un editor, que utilice una serie de botones, y un teclado, para poder editar la mainframe seleccionada.

La clase *editor_mainframe()* recibe los parámetros : *master, mainframe, posx* y *posy*:

- **master:** Será la ventana de Tkinter donde se va a construir el editor. En el caso del segundo nivel, se pasará la ventana *window2*.
- **mainframe:** Es el objeto tipo *mainframe()* que va a copiarse y editarse.
- **posx:** Posición *x* de la esquina superior izquierda en la interfaz.
- **posy:** Posición *y* de la esquina superior izquierda en la interfaz.

Creando la copia:

Para crear la copia de la unidad seleccionada, se podría instanciar una clase *mainframe()*, como las que hemos usado en el primer nivel, aunque no es recomendable. Ya que, si se hiciera, convivirían dos objetos tipo *mainframe()* exactamente iguales, ambos solicitando información a la MPS real, es decir, solicitando el mismo

dato por duplicado.

Por ello, es importante aclarar que dentro de *editor_mainframe()* se realizará una copia de un objeto *mainframe()* ya existente, para evitar el envío innecesario de comandos SCPI a la MPS real.

Para construir la copia se utiliza la variable *copias,* la cuál será un diccionario de variables de tipo *StringVar()*, cada una de ellas con un entry asociado, al igual que ocurría en la clase *mainframe()* con la variable *entry.*

De esta forma, para crear la copia solo debemos cargar el mismo fondo, e igualar las cadenas de caracteres de *copias* a las cadenas de la variable *entry* creadas en *mainframe()*:

```
self.copias[i]=mainframe.entry[i]
```

Figura 4-18. Usar las cadenas de *mainframe()* en la copia construida en *editor_mainframe()*.

Creando el editor:

Para crear el editor, se cargará una imagen de fondo distinta, con el objetivo de que el usuario final pueda distinguir bien el editor.

Figura 4-19. Fondo del editor.

Se han usado las siguientes variables para su construcción:

- **entry:** Es un diccionario, que se utilizará de forma similar al diccionario del mismo nombre: *entry*, que usábamos en la clase *mainframe()*. Este diccionario de cadenas tendrá asociados los display del editor.
- **boton_hab:** Es una lista que contiene objetos de tipo *b_onoff()*, es decir los botones que se ocupan de la habilitación/deshabilitación de la salida del módulo.

El editor tiene asociado un método mediante el cual inicializa todas sus variables, llamado *cargar()*. De forma que, tras su ejecución, los botones On/Off muestren el mismo estado que tienen en la MPS física, y los displays construidos muestren el último valor establecido para las salidas (el paso 1 definido en el capítulo 4.1.2).

Además, a todos estos entrys (o displays), se les asociará un evento, de forma similar a como se llevó a cabo en el capítulo 3.4. Para que cada vez que se pulse en ellos, se construya un teclado.

Para ello, se ha utilizado el método *bind()*, y se han asociado los entry a la clase *teclado()*:

```
self.disp.bind("<Button-1>",lambda event,master=master,
    editor=self,mainframe=mainframe,identificador=i:teclado(master,editor,mainframe,identificador))
```

Figura 4-20. Evento asociado a la variable *disp*. La cuál contiene un widget de tipo entry.

Método: *cargar()*

Este método es una de las funcionalidades del editor, y consiste en inicializar todas las variables relativas a *editor_mainframe()*. Como, por ejemplo, los botones On/Off, o el contenido de los displays.

Tiene un comportamiento similar al método *actualizar()*. Con la diferencia de que *cargar()* sólo recorrerá las columnas de módulos una única vez. Los parámetros que recibe son: *master*, *mainframe*, y *col*.

- **master:** Es la ventana de Tkinter donde se ha construido la mainframe. Es necesario que *actualizar()* la reciba, para poder utilizar el método *after()*, que se explicó en el capítulo 3.
- **mainframe:** Es el objeto que representa la MPS que se pretende editar.
- **col:** Hace referencia a la columna en la que se encuentra el módulo que queremos actualizar.

Este método realizará las tres siguientes operaciones;

1. Preguntará a la mainframe física por los niveles establecidos para la salida de los módulos (paso 1), utilizando para ello el método *query()*. Por ejemplo, para conocer el nivel de voltaje:

```
respuesta=mainframe.modulo[col].query("VOLT:LEV?")
```

Figura 4-21. El método *cargar()* preguntando al módulo por su nivel de voltaje.

2. Inicializará los botones On/Off, llamando al método *inicializar()* definido dentro de los botones, y que explicaremos más tarde:

```
self.boton_hab[col].inicializar(self,mainframe,col)
```

Figura 4-22. El método *cargar()* inicializando el botón On/Off de la columna *col*.

3. Y, por último, nos aseguramos de que sólo ejecutaremos el método *cargar()* una vez por cada módulo. Para ello, introduciremos la instrucción *after()* dentro de un condicional *If*, de la siguiente forma:

```
if col<max_col-1: #Sólo seguimos hasta llegar al modulo 7.
    master.after(100,self.cargar,master,mainframe,col+1)
```

Figura 4-23. Uso del método *after()* en *cargar()*.

4.2.2.2 Otros elementos:

Clase: *b_onoff()*

Esta clase se utiliza para crear botones On/Off, que permitan habilitar o deshabilitar la salida del módulo (paso 2). Recibe los siguientes parámetros:

- **editor:** Recibe un objeto del tipo *editor_mainframe()*, que es el elemento sobre el que se creará el botón.
- **mainframe:** Recibe el objeto *mainframe()* que se está editando, es decir, aquel cuyas salidas

se habilitarán o deshabilitarán utilizando el botón.

- **col:** Posición del módulo que será controlado por el botón.
- **posx:** Posición x donde dibujar el botón.
- **posy:** Posición y donde dibujar el botón.

La clase *b_onoff()* utiliza la variable interna: *disponible*. Una variable de tipo booleano que valdrá *True* cuando el estado es On, y *False* cuando el estado es Off. Además, tiene asociados dos métodos: *switching()*, e *inicializar()*.

Figura 4-24. Formatos del botón On/Off.

Método: *inicializar()*

Este método se llamará desde *cargar()*, y se encargará de inicializar el botón On/Off. Para ello, el método *inicializar()* le pregunta a la MPS física a través de la siguiente instrucción:

```
estado=mainframe.modulo[col].query("OUTP:STAT?")
```

Figura 4-25. Pregunta al módulo nº *col*, por el estado de habilitación de su salida.

Método: *switching()*

Es el método que se ejecutará cada vez que pulsemos en el botón On/Off. Se encarga de alternar entre los estados On y Off. Para ello, se llevan a cabo las siguientes instrucciones:

```
if self.disponible==True:
    #Deshabilitamos la salida, y reconfiguramos el boton a OFF:
    mainframe.modulo[col].write("OUTP:STAT OFF")
    self.btn.config(text='False', image=i_off)
    self.disponible=False

else:
    #Habilitamos la salida, y reconfiguramos el boton a ON:
    mainframe.modulo[col].write("OUTP:STAT ON")
    self.btn.config(text='True', image=i_on)
    self.disponible=True
```

Figura 4-26. Interior del método *switching()*.

Estas instrucciones pueden traducirse de la siguiente manera:

Si al pulsar sobre el botón, el estado es On (*disponible* = *True*):

1. Deshabilita la salida del módulo físico
2. Reconfigura el botón y carga la imagen con el formato Off.
3. Cambia a Off la variable *disponible*.

Si al pulsar sobre el botón, el estado es Off (*disponible* = *False*):

1. Habilita la salida del módulo físico.
2. Reconfigura el botón y carga la imagen con el formato On.
3. Cambia a On la variable *disponible*.

4.2.2.3 Clase: *teclado()*

Esta clase se ejecutará cada vez que se pulse en uno de los displays del editor, construyendo un objeto que represente a un teclado numérico. Recibe los siguientes parámetros:

- **master:** Recibe la ventana donde se encuentra el editor: *window2*.
- **editor:** Recibe un objeto del tipo *editor_mainframe()*, que contiene los displays que se van a editar utilizando este teclado.
- **mainframe:** Recibe el objeto *mainframe()* que representa el MPS que se va a modificar.
- **identificador:** Indica qué modulo se modificará, y qué salida: voltaje o corriente.

Para construir el teclado, se construye una ventana de un nivel superior a partir de *master*, llamada *window3*, utilizando el método *Toplevel()*. En esta nueva ventana, se construirá un teclado numérico, y se podrán establecer nuevos valores de tensión y corriente para los módulos.

Cada botón del teclado tendrá asignado dos cosas: una imagen y una clave. La imagen se utiliza para cubrir el botón con ella, y la clave contiene una cadena de caracteres con el número que se asocia a la tecla, excepto los botones *intro* y *borrar*.

Una vez que se construyen las teclas, cada vez que son pulsadas, se ejecutará el método *escribir()*, que colocará las claves en un entry auxiliar, al que se ha llamado *consola*.

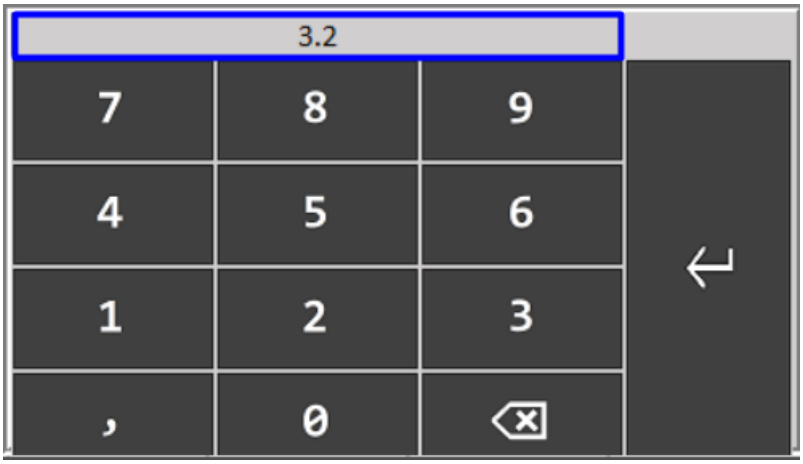

Figura 4-27. El entry *consola* marcado en azul en el teclado.

Método: *escribir()*

Recibe los siguientes parámetros:

- **valor:** Contiene la clave de la tecla.

- **editor:** Recibe un objeto del tipo *editor_mainframe()*.
- **mainframe:** Recibe el objeto *mainframe()* que contiene el MPS que se va a modificar.
- **identificador:** Indica qué modulo se modificará, y qué salida: voltaje o corriente.

El método *escribir()* llevará a cabo varias tareas en función de la tecla pulsada:

1. Si se pulsa *borrar*: eliminará el último valor insertado en la *consola*.
2. Si se pulsa *intro*: cerrará el teclado y enviará el número que se haya escrito en *consola* al módulo, para que este modifique su nivel de voltaje o corriente.

Tiene la siguiente estructura de código:

```
#Si se pulsa intro:
if valor=="intro":
    #Si el entry es de voltaje:
    if identificador[0]=="v":
        #Establecemos un nuevo valor de salida:
        mainframe.modulo[int(identificador[1])].write("VOLT:LEV "+self.consola.get())
        editor.entry[identificador].set(self.consola.get())

    #Si el entry es de corriente:
    else:
        try:
        #Establecemos un nuevo valor de salida:
        mainframe.modulo[int(identificador[1])].write("CURR:LEV "+self.consola.get())
        editor.entry[identificador].set(self.consola.get())

    #Siempre que pulsemos intro, destruiremos el teclado:
    self.window3.destroy()

#Si se pulsa borrar:
elif valor=="*":
    self.consola.set(self.consola.get()[:-1]) #Elimina el último caracter de la consola

#Para cualquier otro boton:
else:
    self.consola.set(self.consola.get()+valor) #Añade la clave del boton a la consola
```

Figura 4-28. Método *escribir()*.

Como vemos, se actuará de forma distinta, dependiendo de la variable *valor*, la cual contiene una cadena de caracteres que informa de qué tecla se ha pulsado, por ejemplo: “intro” equivale a la tecla *intro*, y “*” a la tecla *borrar*.

4.3 Construcción

Para finalizar, una vez que tenemos desarrolladas todas las definiciones anteriores, tendremos a nuestra disposición, las herramientas necesarias para construir la aplicación demandada.

Para inicializarla, sólo se necesitan treinta y cuatro instrucciones en total, de las cuales, veinte pertenecen a la

inicialización de las imágenes de las que hace uso la interfaz.

Esto quiere decir que, gracias a todas las funciones, métodos y clases anteriores, podemos lanzar una interfaz completamente funcional, tan sólo con catorce líneas.

Una vez que se ejecutan estas catorce instrucciones, el código entra al bucle *mailoop()* de Tkinter, y no vuelve a salir jamás (a no ser que cerremos la GUI). Por lo tanto, una vez que se llega al bucle, son los objetos declarados a lo largo de este capítulo, los que se encargan de ejecutar unas u otras instrucciones, en función de lo que el usuario esté comunicando a la aplicación a través de la interfaz.

Son los objetos los que deciden que tareas llevar a cabo, interaccionando los unos con los otros.

```
#Iniciamos el programa:
window=Tk()
window.overrideredirect(1) #Pantalla completa

#Creamos el resource manager:
rm=visa.ResourceManager("@py")

#Inicializamos las listas de direcciones:
direcciones1 = ["gpib0,1,0","gpib0,1,1","gpib0,1,2","gpib0,1,3","gpib0,1,4","gpib0,1,5","gpib0,1,6","gpib0,1,7"]
direcciones2 = ["gpib0,3,0","gpib0,3,1","gpib0,3,2","gpib0,3,3","gpib0,3,4","gpib0,3,5","gpib0,3,6","gpib0,3,7"]

#Construimos la pantalla principal:
m1=mainframe(window, 0, 0, direcciones1)    #Construye una mainframe en la zona superior de la ventana.
m2=mainframe(window, 0, 228, direcciones2)  #Construye una mainframe en la zona inferior de la ventana.
barra(window)                               #Rellena la zona inferior.

b_salir(window,m2,i_exit)                   #Boton para salir de la aplicación.
b_edit(window,m1)                           #Boton Editar de m1.
b_edit(window,m2)                           #Coton Editar de m2.

#Establecemos el tamaño y no dejamos que este pueda modificarse:
window.geometry("800x480+0+0")
window.resizable(0,0)

#Comienza el bucle...
window.mainloop()
```

Figura 4-29. Las catorce líneas que inicializan la interfaz.

5 CONCLUSIÓN

En este proyecto final de grado, se demuestra que es posible llevar a cabo el control de las fuentes de alimentación de la gama HP 66000A, mediante el uso de una interfaz gráfica desarrollada en el entorno de programación: Python.

A partir de aquí, se pueden seguir añadiendo funcionalidades y subsistemas a la GUI: como visualizar la temperatura de la sala o de los hornos, realizar un registro de las operaciones realizadas con los MPS, o incluso hacer que la GUI pueda de decidir desconectar ciertos elementos ante fallos (por ejemplo: un aumento considerable de la temperatura).

La tarjeta BeagleBone Black, es un SBC que ha sido capaz de procesar la aplicación desarrollada, defendiendo su postura de mantenerse como la alternativa principal a la Raspberry Pi.

Debido a su enorme popularidad, y su característica de código abierto, en este proyecto queda manifestado que Python es un lenguaje capaz de proporcionar a los usuarios librerías que transforman grandes problemas, en problemas sencillos.

Se han utilizado, desarrollado y aprendido diversos conocimientos en el ámbito de la ingeniería durante la realización del trabajo, que me han llevado a la conclusión de que, aunque todo puede encontrarse en Internet, sólo encuentran aquellos que saben cómo buscar.

Para finalizar, durante la ejecución del proyecto se ha deducido que la programación orientada a objetos es muy beneficiosa a la hora de llevar a cabo interfaces de usuario, y es que, gracias a la OOP, el programa desarrollado en este documento sólo necesita de catorce instrucciones para inicializarse. Teniendo en cuenta esto, y algunos datos históricos, no es una imprudencia concluir que:

No es una coincidencia que ambas (OOP y GUI) nacieran y evolucionaran de forma simultánea en Xerox, en los años 70. Debido a las características que este tipo de programación podía ofrecer, la orientación a objetos resultó de vital importancia en el desarrollo de las interfaces de usuario, dando lugar a aplicaciones más cercanas al público general, tal y como las conocemos hoy en día.

REFERENCIAS

[1] «Alter Technology,» TÜV Nord Group, [En línea]. Available: https://www.altertechnology-group.com. [Último acceso: Agosto 2019].

[2] «PyVisa,» [En línea]. Available: https://pyvisa.readthedocs.io/en/latest/index.html. [Último acceso: Agosto 2019].

[3] Fundación BeagleBoard, «BeagleBoard.org,» [En línea]. Available: http://beagleboard.org/about. [Último acceso: Agosto 2019].

[4] U. barbarapvn, «Historia del Software: GUI (Graphical User Interface),» *hipertextual,* 2012.

[5] «Python,» Python Software Foundation, [En línea]. Available: https://docs.python.org/2/. [Último acceso: Agosto 2019].

[6] «effbot.org,» [En línea]. Available: https://effbot.org/tkinterbook/. [Último acceso: Agosto 2019].

[7] «Stackoverflow,» [En línea]. Available: https://es.stackoverflow.com/questions/34247/tkinter-seguir-utilizando-programa-mientras-se-ejecuta-un-while.

[8] «Stackoverflow,» [En línea]. Available: https://stackoverflow.com/questions/18499082/tkinter-only-calls-after-idle-once/38817470#38817470.

[9] M. Buttu, El gran libro de Python, Marcombo, 2016.

[10] Keysight, «HP Series 661xxA MPS Power Modules: User´s Guide,» 2001.

[11] Keysight, «HP Series 661xxA MPS Power Modules: Programming Guide,» 2000.

[12] 4. Systems, «Gen4 LCD Capes – BeagleBone Black,» 2017.

[13] Gerald Coley, Beagleboard.org, «BeagleBone Black: System Reference Manual,» 2013.

ANEXO

Se incluye el código Python realizado en este proyecto:

```
from Tkinter import *
import pyvisa as visa

class mainframe():
#CLASS: Construye una mainframe que se actualiza constantemente
    def __init__(self, master, posx, posy, dir_mod):
        #master = la ventana donde queremos colocar el mainframe.
        #posx = la posicion x donde queremos colocar el mainframe.
        #posy = la posicion y donde queremos colocar el mainframe.
        #dir_mod = lista son las direcciones de los modulos.

        self.fr=Frame(master) #Creamos un frame donde insertaremos todos los elementos de la mainframe.
        self.fr.place(x=posx,y=posy,width=800,height=228)

        self.entry={} #Diccionario de variables StringVar() desde donde modificaremos los entry.
        self.modulo={} #Diccionario donde guardaremos las conexiones establecidas vía VISA con los módulos.
        self.identificadores = [["v0","v1","v2","v3","v4","v5","v6","v7"],["c0","c1","c2","c3","c4","c5","c6","c7"]]

        Label(self.fr, image=i_mainframe).place(relx=0,rely=0) #Establece el fondo del objeto dentro de la GUI

        #Bucle que recorre los identificadores: y=numero de fila, x=numero de columna, i=identificador
        for y, fila in enumerate(self.identificadores,1):
            for x, i in enumerate(fila):
                #Voltaje:
                if y==1:
                    try:
                        self.modulo[x]=rm.open_resource("TCPIP0::192.168.30.172::"+dir_mod[x])  #Abre una conexión con el
                                                                                                #módulo de la columna x.
                        self.entry[i]=StringVar(value="--") #Se construye una variable de texto que luego se asociará al
                                                            #entry de voltaje i="vx"
                    except:
                        self.modulo[x]=False #Guardamos False para saber que no se ha conseguido establecer conexión.
                        self.entry[i]=StringVar(value="Not Found")#En caso de no abrir la conexión, informaremos del error.
                    Entry(self.fr,textvariable=self.entry[i],font=("Calibri",12),fg="red",bg="black",borderwidth=0,
                        justify="center").place(y=38,x=17+x*100,width=70,height=22)

                #Corriente:
                else:
                    self.entry[i]=StringVar(value="--") #Se construye una variable de texto que luego se asociará
                    #                                   al entry de corriente i="cx"
                    Entry(self.fr,textvariable=self.entry[i],font=("Calibri",12),fg="red",bg="black",borderwidth=0,
                        justify="center").place(y=38+34,x=17+x*100,width=70,height=22)

        #Llama por primera vez a actualizar, que se repetirá cada 1000 ms.
        self.actualizar(master,0)   #Pasamos la ventana raíz y la columna de inicio desde la que empieza
                                    #la actualización de modulos (columna = 0).

    def actualizar(self,master,col):
    #METODO: Actualiza los entry de la clase mainframe para que coincidan con los de la mainframe real
        #master = la ventana en la que se encuentra el editor.
        #col = numero que indica el módulo que vamos a actualizar.

        #En caso de llegar al ultimo modulo de la mainframe, reinicializamos la columna.
        if col==len(self.identificadores[1]):
            col=0

        #Preguntamos voltaje:
        try:
            respuesta=self.modulo[col].query("MEAS:VOLT?")
            respuesta="{:3.2f}".format(float(respuesta))
        except:
            if self.modulo[col]==False:
                respuesta="Not Found"
            else:
                respuesta="err_update"
        self.entry[self.identificadores[0][col]].set(respuesta)

        #Preguntamos corriente:
        try:
            respuesta=self.modulo[col].query("MEAS:CURR?")
            respuesta="{:3.2f}".format(float(respuesta))
        except:
            if self.modulo[col]==False:
                respuesta="Not Found"
            else:
                respuesta="err_update"
        self.entry[self.identificadores[1][col]].set(respuesta)

        #Aumentamos la columna y repetimos la tarea dentro de 1000 ms
        master.after(1000,self.actualizar,master,col+1)
```

```
class editor_mainframe():
#CLASS: Construye un editor de mainframe
    def __init__(self, master, mainframe, posx, posy):
        #master = la ventana donde queremos colocar el mainframe.
        #posx = la posicion x donde queremos colocar el editor.
        #posy = la posicion y donde queremos colocar el editor.
        #mainframe = contiene una objeto de la clase mainframe, del que se creará una copia.

        self.identificadores = [["v0","v1","v2","v3","v4","v5","v6","v7"],["c0","c1","c2","c3","c4","c5","c6","c7"]]

        ############################    COPIA DE UNA MAINFRAME    ##############################################
        self.fr=Frame(master) #Creamos un frame donde incluiremos todos los elementos de la copia del mainframe
        self.fr.place(x=posx,y=posy,width=800,height=228)

        self.copias={} #Diccionario de variables StringVar()

        Label(self.fr, image=i_mainframe).place(relx=0,rely=0) #Establece el fondo del objeto dentro de la GUI

        for y, fila in enumerate(self.identificadores,1):
            for x, i in enumerate(fila):
                #Voltaje:
                if y==1:
                    self.copias[i]=mainframe.entry[i] #En los entry de la copia guardamos lo mismo que en el original.
                    Entry(self.fr,textvariable=self.copias[i],font=("Calibri",12),fg="red",bg="black",borderwidth=0,
                        justify="center").place(y=38,x=17+x*100,width=70,height=22)

                #Corriente:
                else:
                    self.copias[i]=mainframe.entry[i] #En los entry de la copia guardamos lo mismo que en el original.
                    Entry(self.fr,textvariable=self.copias[i],font=("Calibri",12),fg="red",bg="black",borderwidth=0,
                        justify="center").place(y=38+34,x=17+x*100,width=70,height=22)


        ###################################   EDITOR   ###########################################################
        self.fr2=Frame(master) #En este frame, colocaremos todos los elementos del editor.
        self.fr2.place(x=posx,y=posy+228,width=800,height=228)

        self.entry={} #Diccionario de variables StringVar()
        self.boton_hab=[] #Lista que contendrá los botones On/Off

        Label(self.fr2, image=i_editor).place(relx=0,rely=0) #Establece el fondo del objeto dentro de la GUI

        for y, fila in enumerate(self.identificadores,1):
            for x, i in enumerate(fila):
                #Voltaje:
                if y==1:
                    self.entry[i]=StringVar(value="--")
                    self.disp=Entry(self.fr2,textvariable=self.entry[i],font=("Calibri",12),fg="red",bg="black",
                        borderwidth=0,justify="center")
                    self.disp.place(y=38,x=17+x*100,width=70,height=22)

                    #Si detectamos click izquierdo en el entry, crearemos un teclado:
                    self.disp.bind("<Button-1>",lambda event,master=master,
                        editor=self,mainframe=mainframe,identificador=i:teclado(master,editor,mainframe,identificador))

                #Corriente:
                else:
                    self.entry[i]=StringVar(value="--")
                    self.disp=Entry(self.fr2,textvariable=self.entry[i],font=("Calibri",12),fg="red",bg="black",
                        borderwidth=0,justify="center")
                    self.disp.place(y=38+34,x=17+x*100,width=70,height=22)

                    #Si detectamos click izquierdo en el entry, crearemos un teclado:
                    self.disp.bind("<Button-1>",lambda event,master=master,
                        editor=self,mainframe=mainframe,identificador=i:teclado(master,editor,mainframe,identificador))

                    #Para acabar, creamos los botones On/Off, insertandolos en la lista boton_hab
                    self.boton_hab.append(b_onoff(self, mainframe, x, 30+x*100, 180))

        #Llama por primera vez a cargar, que se repetirá cada 100 ms, pero sólo recorrerá los entry una vez.
        self.cargar(master,mainframe,0) #Pasamos la ventana raíz, la mainframe que estamos editando, y la columna
                                        #de inicio desde la que empieza la actualización de modulos (columna = 0).

    def cargar(self,master,mainframe,col):
    #METODO: Carga en los entry del editor los valores de salida configurados en la mainframe física
        #master = la ventana en la que se encuentra el editor.
        #mainframe = objeto de la clase mainframe a través del cuál preguntaremos al mainframe real.
        #col = numero que indica el módulo que vamos a cargar.
```

```
        #Cargamos en el entry el ultimo voltaje establecido:
        try:
            respuesta=mainframe.modulo[col].query("VOLT:LEV?")
            respuesta="{:3.2f}".format(float(respuesta))
        except:
            if mainframe.modulo[col]==False:
                respuesta="X"
            else:
                respuesta="err_charge"
        self.entry[self.identificadores[0][col]].set(respuesta)

        #Cargamos corriente establecida:
        try:
            respuesta=mainframe.modulo[col].query("CURR:LEV?")
            respuesta="{:3.2f}".format(float(respuesta))
        except:
            if mainframe.modulo[col]==False:
                respuesta="X"
            else:
                respuesta="err_charge"
        self.entry[self.identificadores[1][col]].set(respuesta)

        #Inicializamos el boton on/off del modulo:
        self.boton_hab[col].inicializar(self,mainframe,col)

        #Aseguramos que solo preguntamos una vez por cada columna
        max_col=len(self.identificadores[1])
        if col<max_col-1: #Sólo seguimos hasta llegar al modulo 7.
            master.after(100,self.cargar,master,mainframe,col+1)

class teclado():
#CLASS: Construye un teclado que será usado por del editor de mainframe cuando se pulsen sus entry.
    def __init__(self,master,editor,mainframe,identificador):
        #master = la ventana desde la cual se llamó al teclado.
        #editor = contiene un objeto de la clase editor_mainframe, que solicitó la creación de un teclado.
        #mainframe = contiene una objeto de la clase mainframe, el cual esta siendo editado.
        #identificador = El identificador del entry que activó la construcción del teclado

        #Creamos una lista 4x3 con las imagenes de cada tecla:
        self.teclas=[[siete,ocho,nueve],[cuatro,cinco,seis],[uno,dos,tres],[coma,cero,borrar]]
        #Creamos una lista 4x3 con los valores en formato string de cada tecla:
        self.clave=[["7","8","9"],["4","5","6"],["1","2","3"],[".","0","*"]]

        self.window3=Toplevel(master)       #Creamos un nivel superior para el teclado a partir de master
        self.window3.overrideredirect(1)    #Ejecuta la ventana sin los bordes del S.O.(Sólo interior de la ventana).
                                            #Útil para pantalla completa
        self.window3.config(relief="groove",bd=4,bg="#d0cece")

        if int(identificador[1])>3:
        #Si hemos tocado en un entry de los modulos 4, 5, 6 o 7:
            self.window3.geometry("400x225+0+230")      #Coloca el teclado sobre los modulos 0, 1, 2 y 3.
        else:
        #Si hemos tocado en un entry de los modulos 0, 1, 2 o 3:
            self.window3.geometry("400x225+401+230")    #Coloca el teclado sobre los modulos 4, 5, 6 y 7.

        #Encima del teclado crearemos una consola que mostrará lo que se está escribiendo:
        self.consola=StringVar(value="")
        Entry(self.window3,textvariable=self.consola,font=("Calibri",12),fg="black",bg="#d0cece",borderwidth=1,
            justify="center").grid(row=0, column=0, columnspan=4,sticky="nsew")

        #Bucle que recorre las teclas: y=numero de fila, x=numero de columna, tecla=imagen de la tecla.
        for y, fila in enumerate(self.teclas):
            for x, tecla in enumerate(fila):
                key=self.clave[y][x] #Hacemos una copia de la clave correspondiente.

                #Vamos declarando los botones y situandolos en el lugar correspondiente:
                Button(self.window3,image=tecla,borderwidth=0,bg="#d0cece",cursor="hand2",command=lambda valor=key,
                    editor=editor,mainframe=mainframe,identificador=identificador:
                    self.escribir(valor,editor,mainframe,identificador)).grid(row=y+1, column=x)

        #Añadimos el botón intro:
        Button(self.window3,image=intro, borderwidth=0,bg="#d0cece",cursor="hand2",command=lambda valor="intro",
            editor=editor,mainframe=mainframe,identificador=identificador:
            self.escribir(valor,editor,mainframe,identificador)).grid(row=1,column=4,rowspan=4)

    def escribir(self,valor,editor,mainframe,identificador):
    #METODO: Da soporte al comportamiento del teclado, en función de la tecla pulsada.
        #valor = informa de la tecla pulsada.
        #editor = contiene un objeto de la clase editor_mainframe, que solicitó la creación del teclado.
        #mainframe = contiene una objeto de la clase mainframe, el cual esta siendo editado.
        #identificador = contiene el identificador del entry que activó la construcción del teclado

        #Si se pulsa intro:
        if valor=="intro":
            #Si el entry es de voltaje:
            if identificador[0]=="v":
                try:
                    #Establecemos un nuevo valor de salida:
                    mainframe.modulo[int(identificador[1])].write("VOLT:LEV "+self.consola.get())
                    editor.entry[identificador].set(self.consola.get())
                except:
                    editor.entry[identificador].set("err_send")
```

```
            #Si el entry es de corriente:
            else:
                try:
                    #Establecemos un nuevo valor de salida:
                    mainframe.modulo[int(identificador[1])].write("CURR:LEV "+self.consola.get())
                    editor.entry[identificador].set(self.consola.get())
                except:
                    editor.entry[identificador].set("err_send")

            #Siempre que pulsemos intro, destruiremos el teclado:
            self.window3.destroy()

        #Si se pulsa borrar:
        elif valor=="*":
            self.consola.set(self.consola.get()[:-1]) #Elimina el último caracter de la consola

        #Para cualquier otro boton:
        else:
            self.consola.set(self.consola.get()+valor) #Añade la clave del boton a la consola


class b_onoff():
#CLASS: Construye un boton de encendido/apagado
    def __init__(self, editor, mainframe, col, posx, posy):
        #editor = contiene el objeto de la clase editor, desde donde se solicitó la creación del boton On/Off.
        #mainframe = contiene un objeto de la clase mainframe, al que pertenece el botón.
        #col = contiene la columna del módulo al que pertenece el botón.
        #posx = posicion x del boton dentro del editor.
        #posy = posicion y del boton dentro del editor.

        #Creamos el boton, que será inicializado a OFF hasta que se llame al método inicializar:
        self.btn=Button(editor.fr2,text="False",image=i_off,borderwidth=0,bg="#d0cece",cursor="hand2",
            command=lambda:self.switching(mainframe,col))
        self.btn.place(x=posx, y=posy)

        self.disponible=False #Indica el estado del boton: inicializado a OFF hasta que se llame al método inicializar.

    def switching(self,mainframe,col):
    #METODO: Intercambia los estados ON y OFF cuando se pulsa el botón.
        #mainframe = contiene una objeto de la clase mainframe, al que pertenece el botón.
        #col = contiene la columna del módulo al que pertenece el botón.

        #Si al momento de pulsar, el estado es ON:
        if self.disponible==True:
            try:
                #Deshabilitamos la salida, y reconfiguramos el boton a OFF:
                mainframe.modulo[col].write("OUTP:STAT OFF")
                self.btn.config(text='False', image=i_off)
                self.disponible=False
            except:
                print("err_switch")

        #Si al momento de pulsar, el estado es OFF:
        else:
            try:
                #Habilitamos la salida, y reconfiguramos el boton a ON:
                mainframe.modulo[col].write("OUTP:STAT ON")
                self.btn.config(text='True', image=i_on)
                self.disponible=True
            except:
                print("err_switch")

    def inicializar(self,editor,mainframe,col):
    #METODO: Pregunta al mainframe real en qué estado se encuentra el mainframe: si tiene la salida ON, u OFF.
        #editor = contiene el objeto de la clase editor, desde donde se solicitó la creación del boton On/Off.
        #mainframe = contiene una objeto de la clase mainframe, al que pertenece el botón.
        #col = contiene la columna del módulo al que pertenece el botón.

        try:
            #Preguntamos a la mainframe física en que estado se encuentra la salida:
            estado=mainframe.modulo[col].query("OUTP:STAT?")
        except:
            estado=False
            editor.entry["v"+str(col)].set("err_inic")
            editor.entry["c"+str(col)].set("err_inic")

        #Si la mainframe física esta ON:
        if estado==True:
            #También nos ponemos a ON:
            self.btn.config(text='True', image=i_on)
            self.disponible=True
        #Si la mainframe física esta OFF:
        else:
            #También nos ponemos a OFF:
            self.btn.config(text='False', image=i_off)
            self.disponible=False
```

```
class b_salir():
#CLASS: Construye un boton para salir de la aplicacion, o retroceder a un nivel inferior.
    def __init__(self, master, elemento, foto):
        #master = la ventana en la que se construirá el boton.
        #mainframe = el mainframe al que pertenece el boton.
        #foto = contiene la imagen de fondo del boton, que podran ser i_exit o i_atras.

        self.salida=Button(elemento.fr,image=foto,borderwidth=0,bg="#d0cece",cursor="hand2",
            command=lambda:self.salirapp(master))
        self.salida.place(x=0,y=180)

    def salirapp(self, master):
    #METODO: Destruye la ventana cuando se pulsa el boton.
        #master= ventana en la que se encuentra el boton.
        master.destroy()

class b_edit():
#CLASS: Construye un boton para avanzar desde el primer nivel al segundo: hacia el editor de mainframe.
    def __init__(self,master,mainframe):
        #master = la ventana en la que se construirá el boton.
        #mainframe = el mainframe donde se construirá boton.

        self.editar=Button(mainframe.fr,image=i_edit,borderwidth=0,bg="#d0cece",cursor="hand2",
            command=lambda:self.go(master,mainframe))
        self.editar.place(y=150,x=653)

    def go(self,master,mainframe):
    #METODO: Se encarga de crear el nivel superior cuando se pulsa el boton.
        #master = la ventana en la que se construirá el boton.
        #mainframe = el mainframe donde se construirá el boton.

        window2=Toplevel(master)        #A partir de la ventana en la que estamos, creamos un nivel superior.
        window2.overrideredirect(1)     #Pantalla completa
        window2.geometry("800x480+0+0") #Tamaño y colocación en la pantalla.
        window2.resizable(0,0)          #No podremos modificar las dimensiones de la ventana window2.

        m3=editor_mainframe(window2, mainframe, 0, 0)   #Construimos un editor de mainframe.
        barra(window2)                                  #Llenamos el hueco inferior.
        b_salir(window2,m3,i_atras)                     #Construimos un boton para regresar al primer nivel.

def barra(master):
#FUNCION: Construye una barra que cubre la zona inferior de la pantalla indicada
    #master = la ventana cuya zona inferior vamos a cubrir.
    barra=Frame(master)
    barra.place(rely=0.95,width=800,height=24)
    barra.config(relief="groove",bd=4,bg="grey")

#*****************************************************************************************************************
#                          FIN DE DEFINICIONES DE CLASES, METODOS Y FUNCIONES
#_____________________________________________________________________________________________________________



#Iniciamos el programa:
window=Tk()
window.overrideredirect(1) #Pantalla completa

#--------------CARGANDO IMAGENES-----------------------------------#
#Fondos:
i_mainframe=PhotoImage(file="mod2.png")     #Imagen Mainframe
i_editor=PhotoImage(file="emptymod.png")    #Imagen Editor mainframe

#Botones:
i_on=PhotoImage(file="on.png")              #Imagen boton ON
i_off=PhotoImage(file="off.png")            #Imagen boton OFF
i_exit=PhotoImage(file="salir.png")         #Imagen boton salir
i_edit=PhotoImage(file="b1.png")            #Imagen boton editar
i_atras=PhotoImage(file="atras.png")        #Imagen boton atras

#Para el teclado:
uno=PhotoImage(file="1.png")                #Imagen tecla 1
dos=PhotoImage(file="2.png")                #Imagen tecla 2
tres=PhotoImage(file="3.png")               #Imagen tecla 3
cuatro=PhotoImage(file="4.png")             #Imagen tecla 4
cinco=PhotoImage(file="5.png")              #Imagen tecla 5
seis=PhotoImage(file="6.png")               #Imagen tecla 6
siete=PhotoImage(file="7.png")              #Imagen tecla 7
ocho=PhotoImage(file="8.png")               #Imagen tecla 8
nueve=PhotoImage(file="9.png")              #Imagen tecla 9
cero=PhotoImage(file="0.png")               #Imagen tecla 0
coma=PhotoImage(file="coma.png")            #Imagen tecla coma
borrar=PhotoImage(file="borrar.png")        #Imagen tecla borrar
intro=PhotoImage(file="intro2.png")         #Imagen tecla intro
#-----------------------------------------------------------------#

#Creamos el resource manager:
rm=visa.ResourceManager("@py")
```

```
#Inicializamos las listas de direcciones:
direcciones1 = ["gpib0,1,0","gpib0,1,1","gpib0,1,2","gpib0,1,3","gpib0,1,4","gpib0,1,5","gpib0,1,6","gpib0,1,7"]
direcciones2 = ["gpib0,3,0","gpib0,3,1","gpib0,3,2","gpib0,3,3","gpib0,3,4","gpib0,3,5","gpib0,3,6","gpib0,3,7"]

#Construimos la pantalla principal:
m1=mainframe(window, 0, 0, direcciones1)    #Construye una mainframe en la zona superior de la ventana.
m2=mainframe(window, 0, 228, direcciones2)  #Construye una mainframe en la zona inferior de la ventana.
barra(window)                               #Rellena la zona inferior.

b_salir(window,m2,i_exit)                   #Boton para salir de la aplicación.
b_edit(window,m1)                           #Boton Editar de m1.
b_edit(window,m2)                           #Coton Editar de m2.

#Establecemos el tamaño y no dejamos que este pueda modificarse:
window.geometry("800x480+0+0")
window.resizable(0,0)

#Comienza el bucle...
window.mainloop()
```