Proyecto Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Diseño e implementación de una interfaz gráfica para actuar sobre bombas peristálticas y monitorizar en tiempo real la temperatura.

Autor: Alberto Bautista Ferrete
Tutor: Ramón de Jesús Risco Delgado

Dpto. de Física Aplicada III
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Proyecto Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Diseño e implementación de una interfaz gráfica para actuar sobre bombas peristálticas y monitorizar en tiempo real la temperatura.

Autor:
Alberto Bautista Ferrete

Tutor:
Ramón de Jesús Risco Delgado
Profesor titular

Dpto. de Física Aplicada III
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Proyecto Fin de Grado: Diseño e implementación de una interfaz gráfica para actuar sobre bombas peristálticas y monitorizar en tiempo real la temperatura.

Autor: Alberto Bautista ferrete

Tutor: Ramón de Jesús Risco Delgado

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

A mi familia

A mis maestros

Agradecimientos

Los estilos adoptados por nuestra Escuela y utilizada en este texto es una versión y adaptación a Word® del la versión L^AT_EX que el Prof. Payán realizó para un libro que desde hace tiempo viene escribiendo para su asignatura. Por ello, la Escuela le está agradecida. Por otro lado, la adaptación se hizo sobre un formato que el prof. Aguilera arregló, basándose en su tesis doctoral. Su aportación ha sido muy relevante para que este formato vea la luz. Esta adaptación la llevamos a cabo el alumno Silvio Fernández, becario del Centro de Cálculo, y yo mismo, sobre un trabajo preliminar del alumno Julián José Pérez Arias.

A esta hoja de estilos se le incluyó unos nuevos diseños de portada. El diseño gráfico de las portadas para proyectos fin de grado, carrera y máster, está basado en el que el prof. Fernando García García, de la Facultad de Bellas Artes de nuestra Universidad, hiciera para los libros, o tesis, de la sección de publicación de nuestra Escuela. Nuestra Escuela le agradece que pusiera su arte y su trabajo a nuestra disposición.

Juan José Murillo Fuentes

Subdirección de Comunicaciones y Recursos Comunes

Sevilla, 2013

Resumen

Con el siguiente trabajo se pretende ayudar y facilitar la experimentación con las bombas peristálticas de interés en Medicina y Biotecnología mediante una interfaz gráfica al personal del laboratorio de fisica.

Para llevarlo a cabo, se propone como solución el diseño y construcción de una placa electrónica capaz de recopilar un amplio espectro de temperaturas y, a su vez, actuar sobre el caudal de una sustancia líquida que fluye por una serie de bombas peristálticas.

Por último y ligado con lo anterior, se implementrá un software con una interfaz gráfica que facilite la actuación sobre las bombas, la monitorización de las temperaturas y el cambio de parámetros.

Abstract

In our school there are a considerable number of documents, many teachers and researchers. Our students also contribute to this production through its work in order of degree, master's theses. The aim of this material is easier to edit these documents at the same time promote our corporate image, providing visibility and recognition of our Center.

... -translation by google-

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

°C	Grados Celsius.
Cm	Centímetros.
CPU	Central Processing Unit (Unidad central de procesamiento).
cs	Centésimas de segundo.
GHz	Gigaherzio.
GPIO	General Purpose Input/Output (Entrada/Salida de Propósito General).
GUI	Graphical User Interface (Interfaz gráfica de usuario).
LAN	Local Area Network (Red área local).
mA	Miliamperio.
ml	Mililitro.
mV	Milivoltios.
Mbps	Megabit por segundo.
PCB	Printed Circuit Board (Placa de circuito impreso)
PID	Control Integral proporcional y derivativo
RPI3	Raspberry Pi 3.
SO	Sistema Operativo.
SPI	Serial Peripheral Interface (bus serial).
USB	Universal Serial Bus (bus universal en serie).

1 INTRODUCCIÓN

Esto es una cita al principio de un capítulo.

- El autor de la cita -

5.1 Crioperservación

La crioperservación es la técnica por la cual se disminuye a muy baja temperatura (entre -80ºC y -196ºC) células, tejidos, órganos y cuerpos, deteninendo su actividad metabólica pero no provocando su muerte.

Las células y tejidos que han sido criopreservados mantienen sus propiedades tras la descongelación, pero actualmente los órganos presentan la formación de hielo cuando cuando se criopreservan y en su posterior congelación. Esta formación de hielo produce dos problemas principales:

- El agua está presente en el cuerpo de los animales y como no puede ser de otra manera también en sus órganos. El agua de estos órganos se cristaliza por la parte extracelular y no por la parte intracelular, produciendo un desequilibrio osmótico aumentando la concentración de sales disueltas al disminuir la cantidad de agua líquida en ellos. Este problema pone en peligro la viabilidad de las células de los órganos a causa de la deshidratación.
- Durante la congelación se produce la formación de cristales de hielo. El hielo al ocupar un mayor volumen que el agua líquida provoca la presión sobre las células y causa un daño irreparable sobre la estructura interna de las células.

Una posible solución radiica en el uso de crioprotectores. Su funcionalidad es que no se produzca cristales de hielo durante el enfriamiento. Los crioprotectores tienen la habilidad de unirse al agua y la capacidad de reducir los efectos toxicos de las altas concentraciones de sales ya que estos sustituyen el agua de la célula y consiguen mantener el equilibrio osmótico en ella.

Los crioprotectores son muy útiles en células aisladas o en tejidos, pero en los órganos existen diferentes células y cada una tiene sus propias propiedades de crioperservción por lo que utilizar un único crioprotector límita la recuperación de todas las células que forman los órganos.

5.2 Objetivo

En el siguiente proyecto se tiene como objetivo principal ayudar a conseguir la crioperservación de órganos mediante la actualización de los medios electrónicos e informáticos que se disponen en el laboratorio. Para ello, se ha pensado como solución la implementación de una interfaz gráfica en la que facilite el trabajo al/los usuario/s que realicen la experimentación en el laboratorio.

Lo primero que se debe realizar es un estudio del funcionamiento de los equipos con los que se realiza la experimentación. Concretamente los equipos sobre los que se va a implementar la interfaz grafica son unas bombas peristálticas con las siguientes características:

2 HARDAWARE

No es posible hacer realidad la monotorización (recopilación de información del entorno para su análisis) y el control (acciones de respuesta tras el análisis) sin la combinación y comunicación entre los distintos elementos físicos que conforman el hardware del proyecto.Tampoco es posible realizar un prototipo de PCB y que cumpla con las especificaciones la primera vez que se diseña. Por ello en los siguientes apartados y subapartados se pueden ver los diferentes componentes utilizados, la justificación de su uso y los pasos que se llevaron a cabo en el diseño, modelado, rectificaciones y montaje de la placa PCB antes de su correcto funcionamiento.

2.1 Componentes

En los apartados que vienen a continuación se describen los diferentes elementos físicos utilizados para poder hacer realidad la monotorización y actuación de las bombas peristálticas, así como la justificación de su uso y el coste económico de cada uno

2.1.1 Raspberry Pi 3: modelo B+

La comunicación entre el medio físico y el usuario para la recopilación de datos y la toma de decisiones se debe llevar a cabo mediante un ordenador. Para no tener siempre este dispositivo conectado cada vez que se requiera la utilización de las bombas peristálticas se ha optado por la utilización de una Raspberry pi. Concretamente se ha optado por el modelo B+ de la Raspberry Pi 3 (ver figura 2.1) No es más que un micro computador pensado para no tan altas prestaciones como los ordenadores que se pueden ver hoy en día. En él se puede cargar un sistema operativo desde la tarjeta Micro SD[1], donde previamente se ha instalado dicho sistema operativo.

La Raspberry Pi 3 B+ para que funcione debe de estar alimentada con un cargador de 2500 mA y 5V

Entre sus especificaciones resaltan: un procesador más potente que funciona a 1.4 Ghz, Bluetooth 4.2, 40 pines para la comunicación con el exterior y la posibilidad de trabajar comodamente vía Wi-Fi. El resto de prestaciones técnicas aparecen en el siguiente listado:

- CPU + GPU: Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz
- RAM: 1GB LPDDR2 SDRAM
- Wi-Fi + Bluetooth: 2.4GHz y 5GHz IEEE 802.11.b/g/n/ac, Bluetooth 4.2, BLE
- Ethernet: Gigabit Ethernet sobre USB 2.0 (300 Mbps)
- GPIO de 40 pines
- 4 puertos USB 2.0
- Puerto CSI para conectar una cámara.
- Puerto DSI para conectar una pantalla táctil
- Salida de audio estéreo y vídeo compuesto
- Power-over-Ethernet (PoE)

Su precio es asequible para las buenas funcionalidades que ofrece. El coste de esta ha sido de 83,01€

[1] La Raspberry Pi 3 B+ no trae memoria. Se la ha de incorporar mediante una Micro SD cuyo modelo y características aparecen en subapartado 2.1.1.1

Figura 2.1: Raspberry Pi 3 B+

2.1.1.1 Micro SD

Como se ha mencionado en el apartado anterior hace falta insertar en la Raspberry una memoria para poder cargar el sistema operativo y poder trabajar en el proyecto.

Se ha optado por una tarjeta de memoria de 32 GB micro SD con una velocidad de lectura/escritura de hasta 100 MB/s para disparar y transferir rápido entre sus características más llamativas.

El coste de esta ha sido de 19 €.

Figura 2.2: Micro SD

2.1.2 Módulo MCP4725

El módulo MCP4725 (figura 2.2) es un convertidor de digital a analógico (DAC) de baja potencia, alta precisión, canal único y salida de voltaje de 12 bits con memoria no volátil (EEPROM). Es decir, mediante señales digitales de la Raspberry se puede actuar sobre el voltaje (analógico) y manipular las velocidades de las bombas peristálticas; es decir, se puede actuar sobre el caudal de dichas bombas.

El MCP4725 se controla por I2C por lo que es sencillo realizar su lectura. Dispone de dos posibles direcciones, que se elige mediante la conexión del pin ADDR.

También incorpora una memoria EEPROM que permite que mantenga el nivel de tensión incluso después de un corte de alimentación.

La tensión de alimentación del MCP4725 es de 2.7V a 5.5. La tensión máxima que puede proporcionar es Vcc. Alimentado a una tensión de 5V sus 4096 niveles (12 bits) suponen una precisión de, aproximadamente, 1mV.

Se ha comprado 2 módulos, siendo el precio toal por los dos de 8.79 €

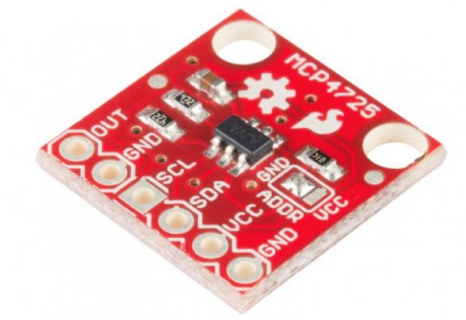


Figura 2.3: Módulo MCP4725

2.1.3 Módulo rele de 2 canales

Estos interruptores permiten activar o desactivar otros componentes de alto voltaje o amperaje, como pueden ser las bombas peristálticas, con una pequeña señal proporcionada por la Raspberry Pi.

El módulo (ver figura 2.3) posee 2 relés de alta calidad, capaces de manejar cargas de hasta 250V/10A. Cada canal posee aislamiento eléctrico por medio de un optoacoplador y un led indicador de estado. Su diseño facilita el trabajo con Raspberry Pi. Este modulo Relay activa la salida normalmente abierta (NO: Normally Open) al recibir un "0" lógico (0 Voltios) y desactiva la salida con un "1" lógico (5 voltios).

Entre las cargas que se pueden manejar tenemos: bombillas de luz, luminarias, motores AC (220V), motores DC, solenoides, electroválvulas, calentadores de agua y una gran variedad de actuadores más. Se recomienda realizar y verificar las conexiones antes de alimentar el circuito, también es una buena practica proteger el circuito dentro de un case.

Entre sus especificaciones técnicas se encuentran:

- Voltaje de Operación: 5V DC
- Señal de Control: TTL (3.3V o 5V)
- Nº de Relays (canales): **2 CH**
- Modelo Relay: SRD-05VDC-SL-C
- Capacidad máx: 10A/250VAC, 10A/30VDC
- Corriente máx: 10A (NO), 5A (NC)
- Tiempo de acción: 10 ms / 5 ms
- Para activar salida NO: 0 Voltios
- Entradas Optoacopladas
- Indicadores LED de activación

Se ha utilizado un único módulo, siendo el precio por la compra de 3 de ellos de 9 €.

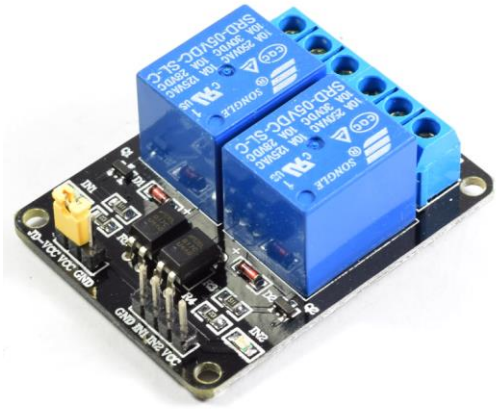


Figura 2.4: Módulos 2 relés

2.1.4 Buzzer

El buzzer (figura 2.4) es un instrumento que emite señales acústicas para dar un aviso. Se trata de un electroimán que hace oscilar una pequeña lámina de metal. Se utilizará para emitir una señal de alarma en caso de que se den unas condiciones adecuadas.

Especificaciones técnicas:

- Voltaje de Operación: 3.3V - 5V DC
- Tipo: Piezo eléctrico pasivo
- Incluye el transistor S8550
- Pines: VCC, GND y Señal

Se ha utilizado un único módulo, siendo el precio por 5 de ellos de 8 €

Figura 2.5: Buzzer

2.1.5 Termopar Adafruit MAX31856 tipo K

Un termopar[2] es transductor que ofrece como salida una diferencia de potencial (del orden de los mV) en funcion de la diferencia de la diferencia de temperatra que se produce entre uno de los extremos de ≪medida≫ y otro de ≪referencia≫

El amplificador para termopar MAX31856 (ver fifura 2.5) digitaliza la señal de cualquier tipo de termpopar. Los datos de salida se ofrecen en grados Celsius (°C). Posee una resolución de temperaturas de 0.0078125 °C, exibe una precisión de medición de ±0.15 % y permite lecturas comprendidas entre -210 °C y 1800 °C (dependiendo del tipo de termopar: K, J, N, R, S, T, E y B), rango más que suficiente para las temperaturas que se manejan en este proyecto. Las entradas de termopar están protegidas contra sobretensiones de hasta ±45V. Sus características más relevantes son:

- Proporciona lecturas de temperatura de termopar de alta precisión.
  - Incluye corrección automática de linealización para 8 tipos de termopares.
  - ± 0.15% (máx., -20 ° C a + 85 ° C) Termopar de escala completa y error de linealidad.
  - Resolución de temperatura de termopar de 19 bits, 0.0078125 ° C
- La compensación interna de la unión en frío minimiza los componentes del sistema
  - ± 0.7 ° C (máx., -20 ° C a + 85 ° C) precisión de unión fría
- La protección de entrada de ± 45V proporciona un rendimiento robusto del sistema.
- El filtrado de rechazo de ruido de 50Hz / 60Hz mejora la información de pedido del rendimiento del sistema
- Paquete TSSOP de 14 pines

Se ha hecho uso de tres módulos amplificadores de termopar MAX31856 junto a sus respectivas sondas de temperatura para este proyecto. El motivo radica en la neceidad de monotorizar la termperatura del

[2] Para más información sobre los termopares se recomienda visitar la web número… de la bibliografía

fluido ...

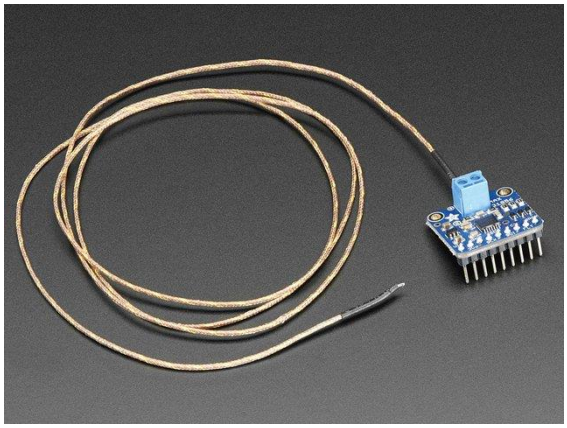


Figura 2.6: Amplidficador de termpopr MAX31856 y sonda de temperatura

2.1.6 Bidireccional

Un convertidor de niveles lógicos bidireccional es capaz de bajar y subir señales a diferentes voltajes. Como la Raspberry que se ha utilizado trabaja a un nivel lógico de 3.3 V y algunos de los módulos utilizados requieren un nivel lógico de 5 V, surge la necesidad de utilizar este dispositivo electrónico para convertir las señales de 3.3 V a 5 V y así conseguir que funcionen todos los módulos.

Se ha hecho uso de un único módulo siendo el coste por cinco de ellos de 7€.

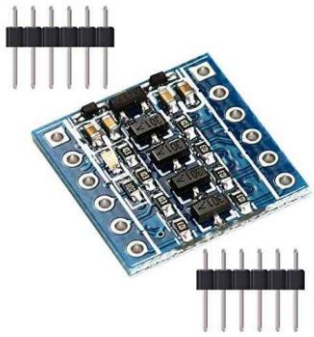


Figura 2.7: Convertidor de nivel lógico bidireccional de 3.3V a 5V

2.2 Fabricación de PCB

2.2.1 Puebas iniciales

El primero de los pasos a realizar es la comprobación del correcto funcionamiento de todos los elementos antes descritos (MCP4725, módulo de dos relés, termopar MAX31856…). Para ello, y mediante la Raspberry Pi 3 B+, una placa protoboard y diferentes cables de conexión, se realizaron diferentes programas[3] en Pythonv[4] (ver figura 2.7). En el capítulo 3 y en el apéndice B se hace mayor hincapié en todo lo relacionado con la programación y elección de pines.

[3] Para ver con más detalle los programas realizados consulte el Apendice B: Códigos
[4] Python es el lenguaje de programación utilizado como se describe en el capítulo 3: Software

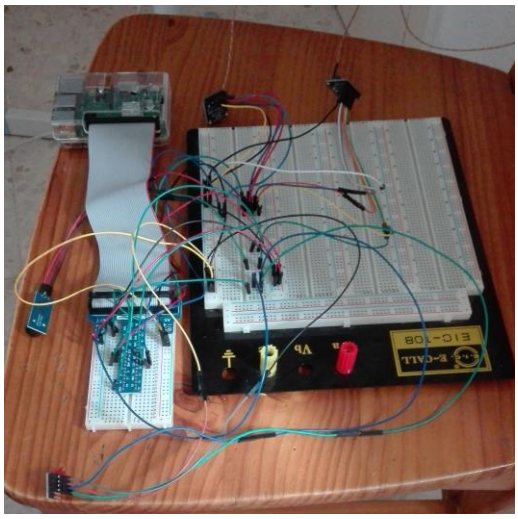


Figura 2.8: Pruebas iniciales con protoboard.

2.2.2 Diseño

El siguiente paso consiste en la elaboración del prototipo de placa mediante el programa de diseño EAGLE®[5].Los requisitos fundamentales que debe cumplir el diseño son los mostrados a continuación:

- Minimizar las dimensiones de la placa para que cupiesen todos los módulos utilizados, así como una zona reservada para los 40 pines de la Raspberry Pi 3 B+ y dos bornas de salida para los dos MCP4725.
- Diseñar la huella de cada componente manteniendo las dimensiones originales y la distancia entre pines.
- Orientar los módulos de los termopares hacia el exterior de la placa para manipular las sondas cómodamente.
- Reservar sobre la parte externa de la placa una zona para los 40 pines de la Raspberry PI 3 B+ para que el fajín de pines no envuelva al resto de módulos y facilite la visión y manipulación de estos.
- Realizar la elaboración de la PCB a doble cara facilitando que ninguna de las pistas se pueda cruzar.

Teniendo en cuenta lo anterior mencionado se procedió a medir con el calibre el ancho, largo y distancia entre pines de cada componente para posteriormente dimensionar la placa con una superficie de 10x10 cm en la que entrasen todos los módulos que se utilizan (ver figura 2.8)

[5] EAGLE es el programa de diseño utilizado para realizar la PCB. Se describe con más detalle en el capítulo 3: Software y en el apéndice A: Planos

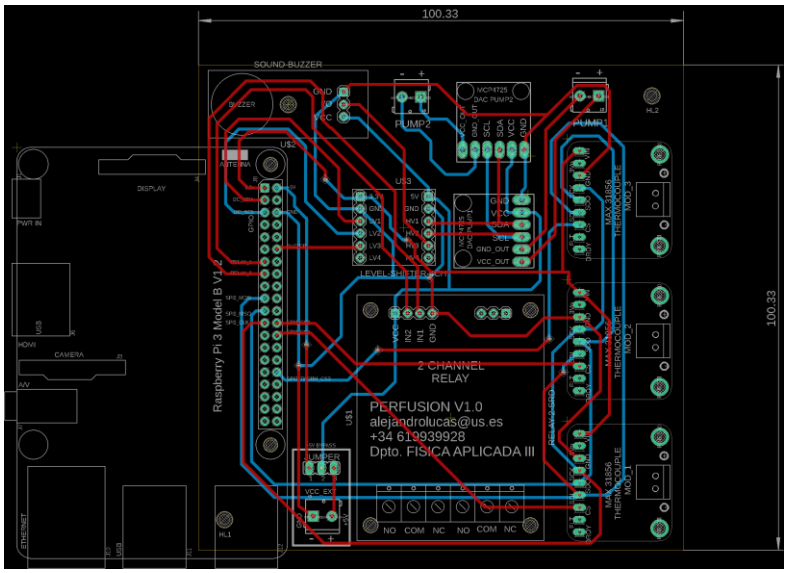


Figura 2.9: Imagen del diseño de la placa

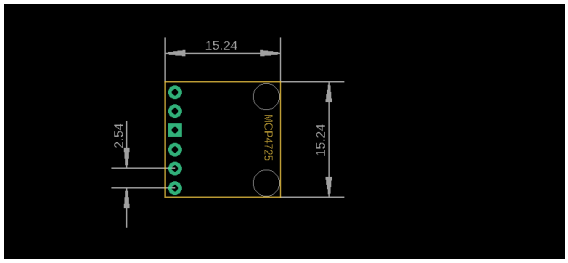


Figura 2.10: Huella del componente MCP4725

2.2.3 Elaboración del prototipo

Tras comprobar el correcto funcionamiento de cada componente y realizar el diseño mediante EAGLE se pocede a forjar en el laboratorio y de forma casera la placa, para posteriormente verificar su buen funcionamiento y enviarla a una empresa para fabricarla de forma más precisa y mejorando la calidad que la elaborada manualmente.

El procedimiento que se ha llevado a cabo junto a los materiales necesarios han sido los siguientes:

- **Impresión:** una vez se haya hecho el diseño se imprime sobre papel vegetal la cara superior (top) y la cara inferior (bottom) de la placa. Es importante, a la hora de imprimir, que el fotolito tenga las mismas dimensiones que el diseño.

Figura 2.11: Papel vegetal

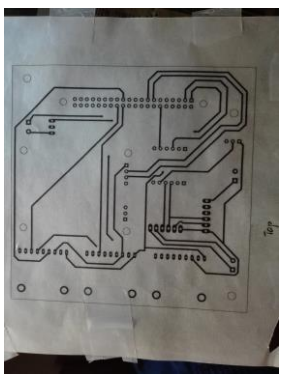


Figura 2.12: Cara superior del deseño sobre papel vegetal

- **Mecanizado:** sabiendo las dimensiones de la PCB (10x10 cm) se corta un trozo de placa fotosensible del mismo tamaño.

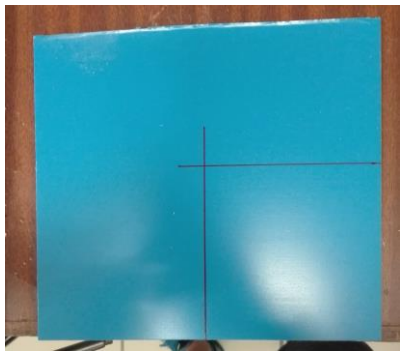


Figura 2.13: Placa fotosensible antes del corte

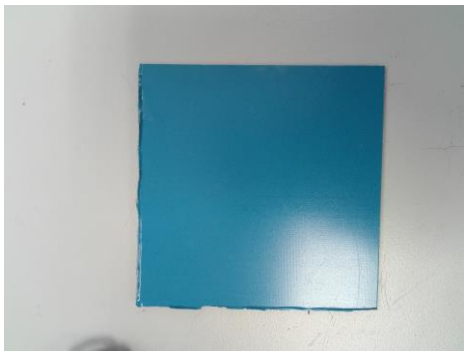


Figura 2.14: Placa fotosensible después del corte

- **Insolación:** las placas fotosensibles disponen de un plástico protector que evita que la placa se exponga a la luz y quede inutilizable. Retirado el plástico hay que colorar las hojas de impresión sobre la placa y asegurarse que las caras top y bottom del papel vegetal queden bien alineadas conforme al borde la placa anteriormente cortada. Este es un paso importante por lo que se recomienda tener cautela y fijar bien el papel vegetal a la placa. Posteriormente se introduce el conjunto en la insoladora donde se realiza el vaciado e insolación durante un tiempo aproximado de un minuto.

Figura 2.15: Insoladora

- **Revelado:** Se introduce la PCB en un recipiente de agua con sosa caustica. Poco a poco se verá como las pistas aparecen sobre la placa. Remover bien el recipiente ayudará a que el revelado salga mejor. Tras esto hay que lavar bien la placa con agua para limpiarla de la sosa caustica.

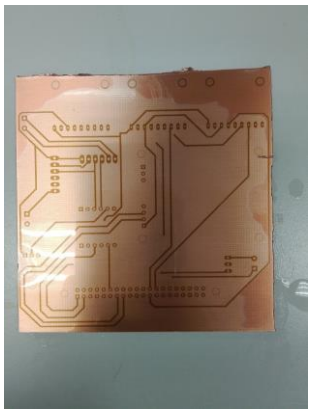


Figura 2.16: Imagen PCB tras el revelado

- **Ataque químico:** este es el paso más peligroso por lo que se recomienda estar bien protequido con gafas, guantes, bata y mascarilla debido a los vapores que el ateque desprende. Se llena un recipitante con ácido clorhídrico hasta que la placa quede sumergida junto a 15-20 ml de agua oxigenada. Pasado

un par de minutos se puede sacar la PCB del recipiente (**evitando el contacto directo con el ácido**) y revisar que que se ha oxidado toda la parte de la PCB donde no hay pistas. Por último, limpiamos la placa nuevamente con agua y la secamos bien.

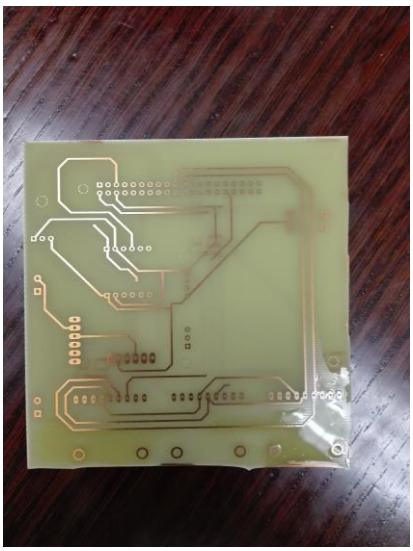


Figura 2.17: Fifura de la PCB tras el ataque químico

- **Taladrado:** con un taladro y una broca de 0.8 mm se realizan todos los agujeros dondese introducirán los pines de los componentes y las vías. Hay que tener cuidado a la hora de taladrar ya que la broca es tan fina que puede llegar a romperse fácilmente.
- **Soldadura con estaño:**

2.2.4 Montaje de componentes y verificación de funcionamiento

Realizada la PCB se coloca cada uno de los componentes sobre los pines de la placa siguiendo la orientación del diseño. Luego se comprueba el correcto funcionamiento de cada módulo mediante un programa[6] de verificación realizado en Python.

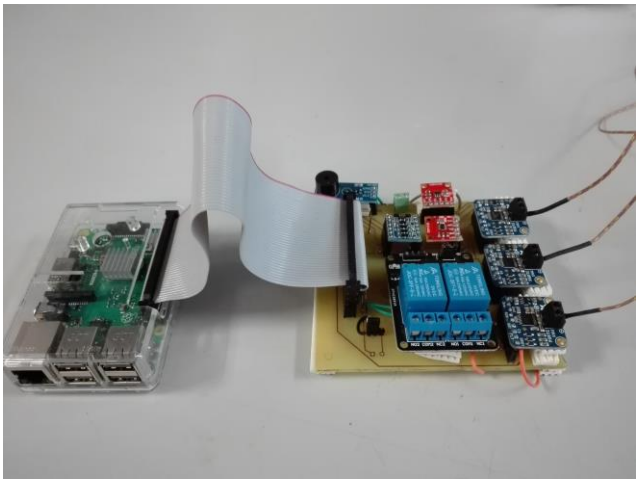


Figura 2.18: RPI3 y placa de laboratorio finalizada.

2.2.5 Placa de Fábrica

Para finalizar y una vez comprobado todos los pasos anteriores se envía el diseño a una empresa para que fabriquen la PCb con mayor precisión y calidad. También habrá que verificar que esta placa funcione perfectamente.

[6] Se puede ver en detalle el código y explicación en el capítulo 3: softawe

3 SOFTWARE

The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point.

Claude Shannon, 1948

La comunicación entre la Raspberry y los diferentes módulos no se lleva a cabo sola. Requiere de un conjunto de programas y rutinas que son las encargadas de hacer posible que la computadora (en este proyecto la RPI3) logre realizar las diferentes tareas que se especifican en este trabajo.

Debido a los motivos que se acaban de comentar, en este capítulo se describe en profundidad como se ha conseguido establecer una comunicación entre la computadora y los módulos; desde programas utilizados hasta librerías y códigos que se han implementado.

3.1 Configuración Raspberry Pi 3 B+

3.1.1 Sistema Operativo

El sistema operativo (SO) que se va a utilizar Raspbian, concretamente la versión 4.19.42-v7+. Raspbian es una distribución de GNU/Linux® basada en Debian lanzada para Raspberry Pi. Su elección se debe al ser un sistema abierto y gratuito de Linux® junto a la facilidad de conexión con los pies de la Raspberry GPIO.

Figura 3.1: Imagen de Raspbian

Para su instalación se ha descargado la versión nombrada anteriormente desde la web de Raspberry[7]

Realizada la descarga, el archivo imagen se copia en la tarjeta microSD mencionada en el punto 2.1.1.1. Pero antes cabe mencionar que la tarjeta microSD ha de ser formateada mediante el programa SD Card Formatter®. Con este programa se consigue restaurar los 32 GB completos en una única partición teniendo la otra inutilizada. Con formatearla en "Quick format" será suficiente.

[7] https://www.raspberrypi.org/downloads/raspbian/

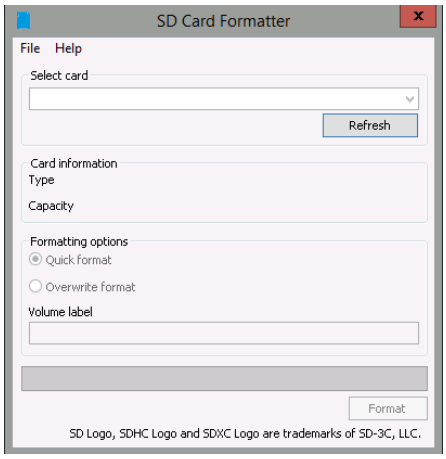


Figura 3.2: Ventana del programa SD Car Formatter

Como se ha utilizado Windows® para descargar raspbian se ha necesitado un programa adicional como Win32DiskImage® para copiar el archivo imagen de Linux en la tarjeta Se deja en el capítulo Referencias en enlace web [1] donde se explica como utilizar este programa de forma satisfactoria. Tras ello, se introduce la tarjeta micro SD en la Raspberry y se tiene preparada para su utilización.

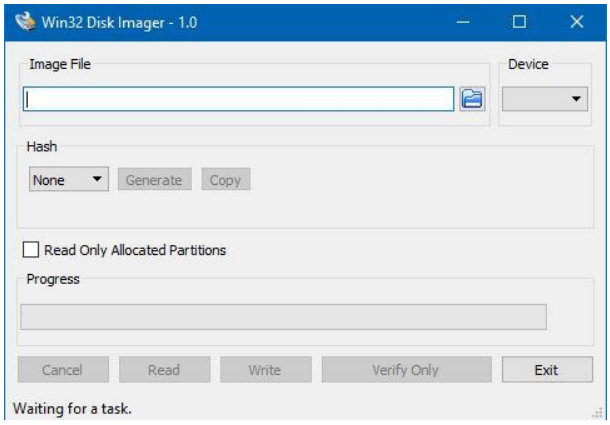


Figura 3.3: Imagen del programa Win32 Disk Imager

3.1.2 Conexión Raspberry Pi desde Windows®

Trabajar desde Windows® es más rápido que trabajar sobre el SO instalado en la RPI3. Además de la comodidad de trabajar, comunicar y transferir archivos de forma imalámbrica. Es por ello que todos los códigos elaborados y diseño de la interfaz gráfica (GUI) se han realizado en un PC con Windows® y posteriormente se han copiado los archivos en la RPI3. El problema reside en la forma de comunicar el PC con la RPI3 de forma inalámbrica y segura. La solución está en el protocolo SSH.

El protocolo SSH permite comunicaciones seguras mediante una arquitectura cliente/servidor entre dos sistemas (en este proyecto PC(cliente) y RPI3 (servidor)) y ofrece a los usuarios conectarse a un host de forma remota. La elección de este protocolo se debe a estas características entre otras:

- El cliente puede verificar que está conectado al mismo servidor al que se conectó enteriormente.
- El cliente transmite su información de autenticación al servidor usando una encriptación de 128 bits.
- Todos los datos enviados y reibidos se transfieren por medio de una encriptación de 128 bits, siendo extremadamente difícil de descifrar y leer.

Como primer paso se debe habilitar SSH en la RPI3. Para ello se crea un fichero sin extensión llamado SSH en la partición denominada "boor" en la microSD. Luego para la conexión de la RPI3 a la red de la universidad se dio acceso desde el exterior solicitando una IP pública vinvulada a una roseta del laboratorio con la instalación de un router para manejar los puertos de entrada/salida. Posteriormente se solicitó la apertura de puertos en el firewall perimetral de la universidad para conceder acceso desde el exterior para el uso del protocolo SSH RDP. La IP pública del laboratorio es *hertz.us.es*.

Existe un problema a la hora de tranferir datos mediante el protocolo de comunicación SSH desde una IP privada hacia la IP privada de la universidad. Por seguridad todos los puertos están bloquedados (incluido el puerto 22 que es el encargado de la comunicación SSH) y vigilados por el firewall. Para poder realizar la comunicación habría que abrir el puerto 22, cosa dificultosa. Aunque existe un método denominado *Port Forwading* (redireccionamiento de puertos), el cual permite acceder desde el exterior a un puerto de una IP privada dentro de una LAN redirigiendo los datos por un puerto abierto mediante el router.

En este proyecto para tener acceso al puerto 22 (encargado del protocolo SSH) se solicitaron puertos a la universidad y se configuró el router para redirigir los datos por el puerto 8857. De esta manera desde cualquier red externa puede tener acceso a la RPI3 conectada a la red del laboratorio de la universidad mediante el redirecionamiento del puerto 8857 al 22 de la propia RPI3

Figura 3.4: Configuración del redireccionamiento de puertos mediante el router.

Para realizar la conexión etre PC y RPI3 para transferir archivos se ha elegido el programa SSH Client®, descargado de su página web[8]. Su elección se debe a las siguientes características:

- Transferencia de archivos por SFTP.
- Capacidad conexión escritorio remoto.
- Reconexión automática en caso de caida de la conexión.

Una vez istalado el programa la conexión a la RPI3 resulta sencilla. Hay que distinguir dos casos:

- Conexión desde una red externa (p.e: conexión desde el hogar del usuario hacia el laboratorio): En la pestaña "Login" en el capo "Host" indicar la dirección IP pública del laboratorio *hertz.us.es* y en el campo Port, escribir el puerto 8557. Como "Username" y "Password" se ha de escribir *pi* y *raspberry* respectivamente.

[8] https://www.bitvise.com/ssh-client-download

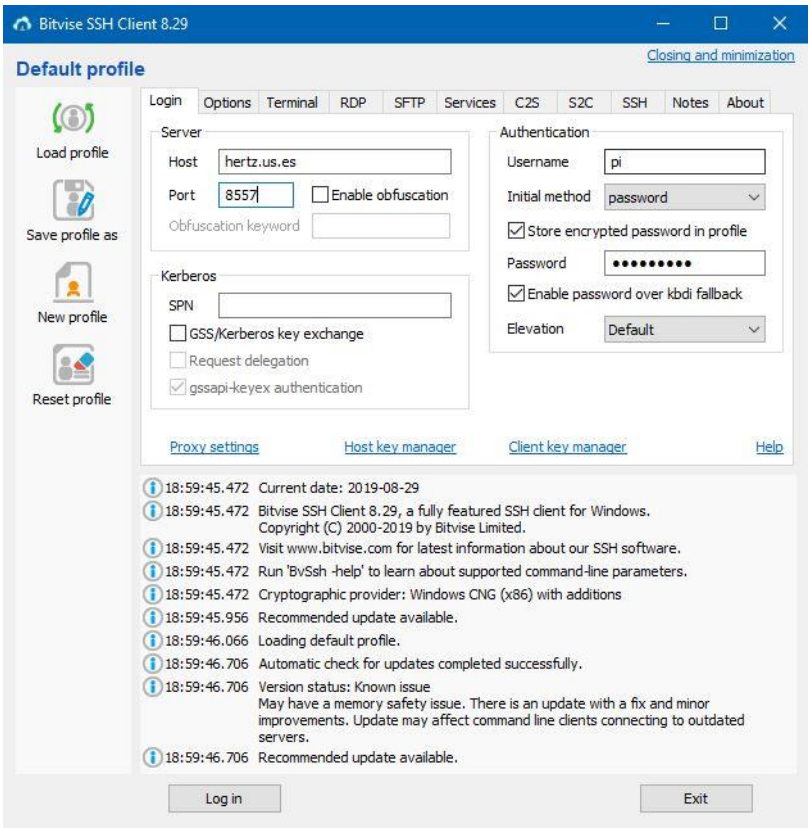


Figura 3.5: Conexión a la RPI3 desde una red externa

- Conexión desde la misma red (p.e: todos los dispositivos conectados a la red del laboratorio): Si se trabaja dentro de la misma red no hace falta habilitar puertos. En la pestaña "Login" en el capo "Host" indicar la dirección IP que el router proporciona a la RPI3 *192.168.1.151* y en el campo "Port", escribir el puerto 22. Como "Username" y "Password" se ha de escribir *pi* y *raspberry* respectivamente.

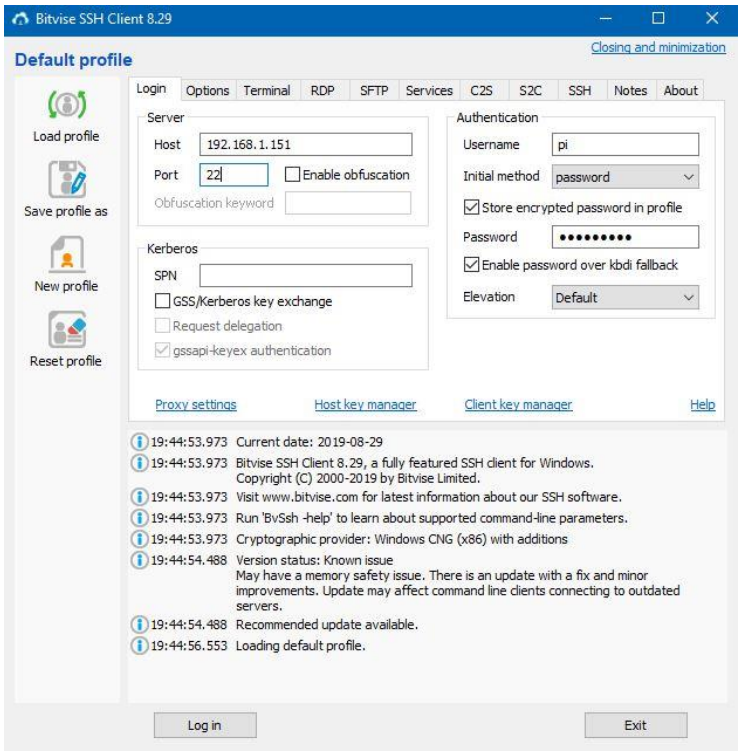


Figura 3.6: Conexión a la RPI3 desde la misma red

Si todo se ha realizado correctamente cuando se pulse sobre "Log in" saldrá una ventana de línea de comandos y otra de SFTP.

NOTA 1: Si se ha conectado por primera vez la placa en la línea de comandos se deben ejecutar los siguientes comandos, aceptando las preguntas que se hagan durante el proceso:

```
sudo apt-get update
sudo apt-get upgrade
```

NOTA 2: Para poder utilizar el escritorio remoto desde Windows® se debe instalar el siguiente paquete desde la línea de comandos.

```
sudo apt-get install xrdp
```

3.1.3 Lenguaje de programación: Python

Python es un lenguaje de programación multiplataforma (Windows, Linux, Mac) orientada a objeto. Contiene una sintaxis legible, fácil de interpretar y aprender por parte del programador. Es de código abierto, con licencia GNU.

Existen una amplia variedad de paquetes instalables con licencia libre (bibliotecas) y una gran comunidad activa en internet donde se puede encontrar códigos de ejemplos o preguntar en los foros para obtener ayuda.

Python no tiene una librería gráfica, pero mediante PyQt se puede crear interfaces gráficas multiplataforma. PyQt es un biding (adaptación de una biblioteca para ser utilizada en un lenguaje de programación distinto como es Python) de la librería gráfica Qt para el lenguaje de Python®.

Los motivos expuestos anteriormente hacen a Python el mejor lenguaje de programación para este proyecto. Su descarga es gratuita a través de su pagina web[9]. Se ha instalado sobre la RPI3 la versión Python 3.5.3 de julio 2017 siendo el entorno de desarrollo Python 3.5 IDLE.

Figura 3.7: Logo de Python.

3.1.4 Selección de pines GPIO

En el capítulo 2.1.1 una de las características de la RPI3 era que proporcionaba 40 pines GPIO de carácter general donde la gran mayoría son utilizados como entradas y salidas digitales y algunos son utilizados con una funcionalidad específica. Los pines GPIO de la RPI3 se estructuran según la siguiente distinción de colores:

- **Amarillo**: Alimentación a 3.3V.
- **Rojo**: Alimentación a 5V.
- **Naranja**: Entradas / salidas de proposito general. Pueden configurarse como entradas o salidas. Tener presente que el nivel alto es de 3.3V y no son tolerantes a tensiones de 5V.
- **Gris**: Reservados.

[9] https://www.python.org/downloads/

- **Negro**: Conexión a GND o masa.
- **Azul**: Comunicación mediante el protocolo I2C para comunicarse con periféricos que siguen este protocolo.
- **Verde**: Destinados a conexión para UART para puerto serie convencional.
- **Morado**: Comunicación mediante el protocolo SPI para comunicarse con periféricos que siguen este protocolo.

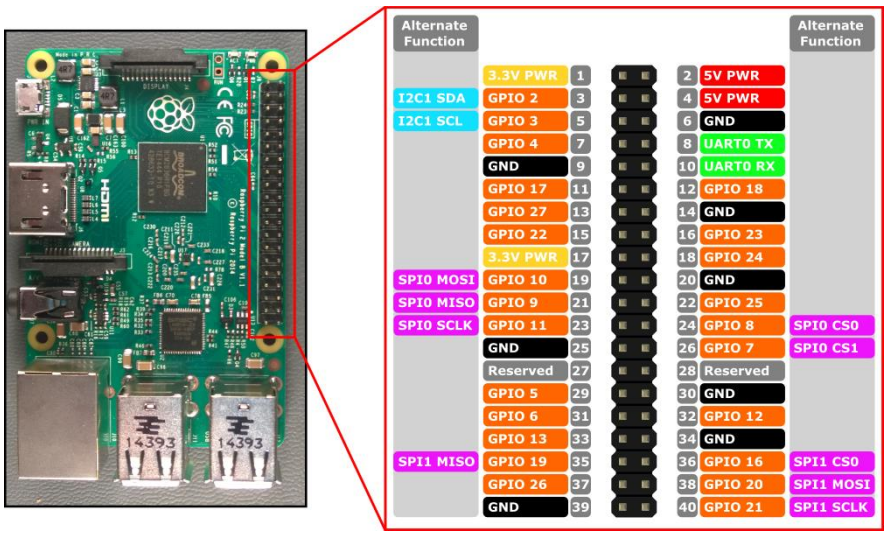


Figura 3.8: Distribucion pines GPIO Raspberry PI 3

En este proyecto se han elegido los GPIO recogidos en la tabla que se observa a continuación para conectar y comunicar los diferentes módulos con la RPI3:

Tabla 1: Configuración de los pines de la RPI3

Módulos	Pines elegidos	Pines donde la señal se adapta de 3.3V a 5V
MCP4725_a	SCL: GPIO 3	
	SDA: GPIO 2	
	VCC: 3.3V	✓
	GND(x2)[10]: Cualquier pin GND	
MCP4725_b	SCL: GPIO 3	
	SDA: GPIO 2	
	VCC: 3.3V	✓
	GND (x2): Cualquier pin GND	
Módulo Relé	Relé_1: GPIO 22	
	Relé_2: GPIO 27	
	VCC: 3.3V	✓
	GND: Cualquier pin GND	
Buzzer	GPIO 18	
	VCC: 3.3V	✓
	GND: Cualquier pin GND	

[10] X2 significa que el módulo MCP4725 contiene dos pines a tierra GND.

Termopar_1	SCLK: GPIO 11	
	SDI: GPIO 10	
	SDO: GPIO 9	
	CS: GPIO 8	
	VCC: 3.3V	
	GND: Cualquiera de los pines GND	
Termopar_2	SCLK: GPIO 11	
	SDI: GPIO 10	
	SDO: GPIO 9	
	CS: GPIO 8	
	VCC: 3.3V	
	GND: Cualquiera de los pines GND	
Termopar_3	SCLK: GPIO 11	
	SDI: GPIO 10	
	SDO: GPIO 9	
	CS: GPIO 8	
	VCC: 3.3V	
	GND: Cualquiera de los pines GND	

3.1.5 Puesta en marcha de los componentes

Antes de la realización de la interfaz gráfica se ha de comprobar el funcionamiento de cada uno de los módulos utilizados.

Una de las malas prácticas en la programación es no hacer códigos para testear la funcionalidad de cada módulo por separado. Por ello siguiendo esta recomendación, en este capítulo se explica cómo se ha elaborado cada uno de los códigos unitarios que hacen funcionar a cada módulo de forma correcta. Así, si la interfaz gráfica falla, se puede ejecutar estos códigos para dictaminar con más acierto y facilidad dónde puede haber un fallo en el caso de que surja

No se muestra el código al completo, sólo las líneas más importantes y las librerías utilizadas. En el caso de que se quiera consultar los códigos completos se recomienda mirar el apartadao ANEXO.

3.1.5.1 Código MCP4725

Como el MCP4725 es un convertidor Digitat-Analógico (DAC), el objetivo del código de comprobación consiste en ir aumentando de forma lineal los valores de tensión a la salida (de 0V a 5V) mediante un bucle en el que se aumenta linealmente los dígitos de entrada (de 0 a 4095) para posteriormente decrementar de la misma forma los valores de salida (de 5V a 0V) mediante otro bucle en el que los dígitos de entrada disminuyen (de 4095 a 0).

Los dígitos de entrada varían entre 0 y 4095; correspondiendo el valor 0 a 0V y el valor 4095 a 5V. Para conocer un valor intermedio de la salida de tensión basta con realizar el siguiente cálculo:

$$V_{OUT}[V] = \frac{Dígito\ entrada\ (0 \div 4095)}{4095} \cdot 5V$$

Si se quiere cumplir el objetivo mencionado anteriormente es necesario la instalación de la librería del MCP4725 (es suficiente realizarlo sólo una vez y ya se ha realizado en este proyecto). Para ello en la línea de comandos se ejecuta:

```
sudo pip3 install adafruit-circuitpython-mcp4725
```

Una vez instalada la librería se puede dar uso en Python importándola con el comando:

```
import adafruit_mcp4725
```

Sin duda todo lo realizado hasta ahora no cobra sentido si no existiese el bus I2C en la Raspberry PI 3 para comunicarse con el módulo MCP4725. Mediante el I2C la Raspberry puede compartir información con los módulos (en este caso los dos MCP4725) mediante las lineas de conexión SDA (Serial Data Line) y SCL (Serial Clock Line). Para poder utilizar este bus primero hay que activarlo:

Desde el terminal ejecutar: `sudo raspi-config`

Seleccionar la opción 5: “Interfacing Options”

Seleccionar la opción P5: “I2C”

Elegir “YES”

Acto seguido aparecerá una ventana en la que se solicita al usuario confirmación para finalizar la acción. Se deberá seleccionar “OK” y pulsar a continuación “Finish”

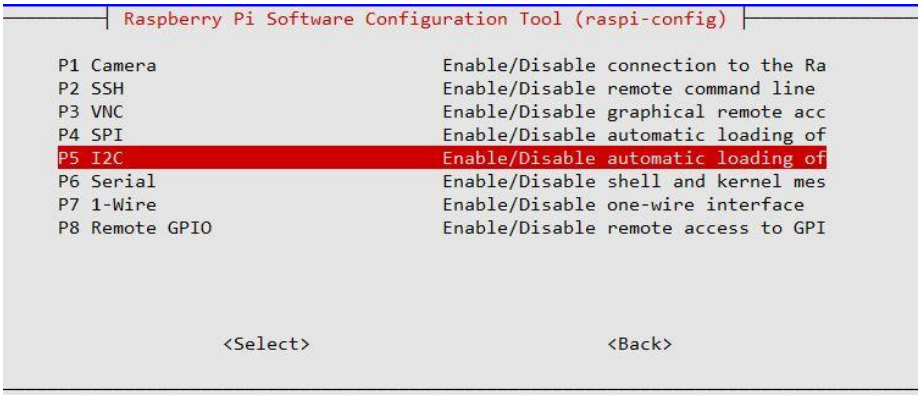


Figura 3.9: Configuración bus I2C

Activado el bus I2C, el siguiente paso es saber que dirección contiene cada MCP4725 para que la RPI3 sepa a qué MCP4725 de los dos tiene que mandar órdenes en un instante concreto.

Saber qué direcciones tiene cada MCP4725 es tan sencillo como ejecutar en el terminal:

```
sudo i2cdetect -y 1
```

En el mapa de direcciones que aparece se puede contemplar las direcciones que se les ha asignado a cada MCP4725 que serán utilizadas en el código de Python para diferenciar ambos módulos.

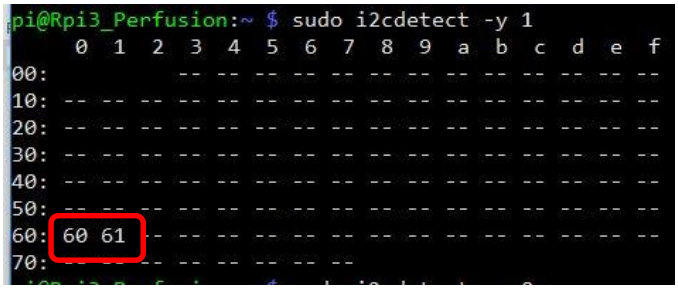


Figura 3.10: Mapa de direcciones del bus I2C

Realizada la configuración del bus I2C ya sí se puede realizar un código funcional para el MCP4725. Como se ha mencionado anteriormente se variará la tensión de salida que genera el módulose. Las lineas más relevantes del código se muestran a continuación junto a una breve explicación:

Librerias necesarias:

```
import board
import busio
import time
import adafruit_mcp4725
```

Inicialización del bus I2C:

```
i2c = busio.I2C(board.SCL, board.SDA)
```

Inicializacion de un MCP4725 concreto con su dirección

```
dac = adafruit_mcp4725.MCP4725(i2c, address=0x61)
```

3.1.5.2 Código Relé

El código del relé se basa en el encendido y apagado de cada uno de los dos relés dentro de un bucle contínuo. Las líneas a destacar son las siguientes.

`import Rpi.GPIO as GPIO` → Con esta linea se importa la libreria para controlar los GPIOs y se le asocia una forma más corta para referirse a ella. A partir de este momento si se quiere ejecutar una función deberemos escribir `GPIO._____` y la función que se requiera.

`GPIO.setmode(GPIO.BCM)` → se establece el sistema de numeración forma desordenada. Este es con la referencia del SoC System On Chip, que son los GPIOs integramente (en la imagen 3.8 los que se denominan GPIO).

`RelayPin_1 = 27` → se establece el GPIO a utilizer por el relé .1

`RelayPin_2 = 22` → se establece el GPIO a utilizer por el relé 2.

`GPIO.setup(RelayPin_1, GPIO.OUT)` → Configuración de GPIO como salida

`GPIO.setup(RelayPin_2, GPIO.OUT)` → Configuración de GPIO como salida

`GPIO.output(RelayPin_1, True)` → Apagado del relé 1

`GPIO.output(RelayPin_1, False)` → Encendido del relé 1

`GPIO.output(RelayPin_2, True)` → Apagado del relé 2

`GPIO.output(RelayPin_2, False)` → Encendido del relé 2

3.1.5.3 Código Buzzer

El código del buzzer no es más que la reproduccion de uno de los 5 tonos que el usuario elige por la entrada del teclado. Cabe destacar las siguientes líneas de código:

`import Rpi.GPIO as GPIO` → Con esta linea se importa la libreria para controlar los GPIOs y se le

asocia una forma más corta para referirse a ella. A partir de este momento si se quiere ejecutar una función deberemos escribir `GPIO. _____` y la función que se requiera.

`class Buzzer(object)` → Se crea una clase (Buzzer) con tipos de datos creados por el programador.

`GPIO.setmode(GPIO.BCM)` → se establece el sistema de numeración forma desordenada. Este es con la referencia del SoC System On Chip, que son los GPIOs integramente (en la imagen 3.8 los que se denominan GPIO).

`self.buzzer_pin = 18`→ se establece el GPIO a utilizer por el buzzer

`GPIO.setup(self.buzzer_pin, GPIO.OUT)` → Configuración de GPIO como salida

`def buzz(self,pitch, duration)` →Se crea la función buzz con la que se origina el tono en function de la melodía que haya introducido el usuario por teclado y la duracion de cada melodía. Está compuesto por:

`period = 1.0 / pitch` → Cálculo del periodo

`delay = period / 2` →Tiempo de la mitad de un periodo

`cycles = int(duration * pitch)` →Cálculo del número de ciclos. →

En un bucle desde 0 hasta el número de ciclos:

`GPIO.output(self.buzzer_pin, True)` →ENcendido del buzzer (emite un sonido)

`time.sleep(delay)` →Espera con el buzzer encendido

`GPIO.output(self.buzzer_pin, False)` →Apagado del buzzer (se deja de emitir sonido)

`time.sleep(delay)` → Espera con el buzzer encendido

Una vez elaborada la función que emite la melodía de forma genérica se crea la function en la que se elige, en función de lo que el usuario haya introducido por teclado (tune), la melodía a sonar:

`def play(self, tune)`

3.1.5.4 Código termopares MAX31856

El código de los termopares consiste en cuatro lecturas de cada termopar cada segundo, mostrando como lecturas dos valores concretos:

- temperatura detectada al final de la sonda del termpopar (temp).
- temperatura del chip del microcontrolador (internal).

El código de cada termopar es similar, diferenciándose entre ellos con las designaciones numéricas 1, 2 y 3. Para este módulo existe una librería que ha de ser instalada e importada denominada “Adafruit_MAX31856” (con hacerlo una vez es suficiente y ya se ha realizado en la RPI3). Para la comunicación entre la RPI3 y los módulos de termopares se debe habilitar el bus SPI. Este bus permite encadenar los termopares desde un solo set de pines, asignando a cada chip un pin distindto de Chip Select.

Las líneas más importantes del código de los termopares son las siguientes:

`from Adafruit_MAX31856 import MAX31856 as MAX31856`→Importa la librería del modulo MAX3856 para su uso.

`software_spi_1 = {"clk": 11, "cs": 7, "do": 9, "di": 10}` →Configura la comunicación SPI del termpopar 1 (Similar para el resto de termopares, cambiando los valores de “cs” por 8 y 12 respectivamente.)

`sensor_1 = MAX31856(software_spi=software_spi_1, tc_type=MAX31856.MAX31856_K_TYPE)` → Establece la configuracion del termopar 1 y le asigna el

tipo K.

`temp = sensor_1.read_temp_c()` → Lectura de la temperatura al final de la sonda.

`internal = sensor_1.read_internal_temp_c()` → Lectura de la temperatura del microcontrolador.

`time.sleep(1.0)` → Espera de un segundo entre lectura y lectura.

3.1.5.5 Código test general

Si se desea se puede ejecutar cada uno de los códigos por separado, pero se ha elaborado un código de testeo general en el que se ejecuta el código de cada uno de los módulos en secuencia cíclica para facilitar al usuario su uso en caso de fallo.

La secuencia seguida por el programa es:

- Encendio y apagado de los relés.
- Reproducción melodía del buzzer.
- Cuatro lecturas de cada uno de los tres termopares.
- Variación de la tensión del MCP4725

El programa completo se puede encontrar en el capítulo ANEXO.

4 INTERFAZ GRÁFICA

B mhjjnlkml.,ll,

Una interfaz gráfica de usuario (GUI) es un software a través del cual un usuario interactúa con el software de la computadora, como el SO, para mostrar información y controles de usuarios, mediante la utilización de menús, iconos, ventanas y otras representaciones de indicadores visuales. Una de sus ventajas es que facilita al usuario la comunicación con el SO de una computadora, siendo más dinámico y visual la ejecución de los códigos, frente a la interfaz de línea de comandos basada en texto. Por ello el usuario que la utilice no requiere de grandes conocimientos informáticos si sólo quiere usarla.

5.3 Estructura

4.1.1 Software

Existen varios programas para realizar una interfaz gráfica como Tkinter o WxPython, pero para este proyecto PyQt ha sido el elegido. PyQt es una adaptación de la biblioteca gráfica Qt para el lenguaje de Python. Con él se puede desarrollar aplicaciones con un entorno gráfico. Concretamente se ha utilizado la versión más actual: PyQt5.

Figura 4.1: Binding PyQt.

Para realizar visualmente la interfaz gráfica se pueden tomar dos vías:

- Programación en Python con PyQt5.
- Qt Designer.

En este proyecto se ha optado por Qt Designer, una herramienta de Qt para diseñar y construir interfaces gráficas a través de Widgets (botones, cuadro de textos, ventanas). Su elección se debe a la facilidad, dinamismo e interactividad para poder crear, dimensionar y modificar la interfaz al gusto del usuario. Además, una vez diseñada, se puede exportar a un archivo ".ui", el cual se puede pasar a un archivo de Python ".py".

Figura 4.2: Logo Qt Designer

El último programa que se ha requerido ha sido PyQtGraph. Este es una biblioteca de gráficos e interfaz de usuario para Python. Con PyQtGraph se consigue mostrar gran catidad de datos en tiempo real y mostrarlos sobre una gráfica. Es ideal para la monotorización de la temperatura medida por los tres termopares y se asocia bien con el programa de diseño Qt Designer.

Cada uno de los programas han sido instalados en la RPI3, por lo que no hace falta volverlos a instalar[11].

4.1.2 Diseño

La realización del diseño de la interfaz gráfica se ha realizado mediante el programa QT Designer. En este subcapítulo se describirá todos los aspectos de dicha interfaz gráfica como: dimensiones, ventanas, botones, displays y ventana gráfica.

Cabe señalar que no todos los widgets[12] que componen la interfaz gráfica tienen funcionalidad, es decir puede haber botones o cuadros combinados que hagan referencia a un dispositivo, GPIO o direcciones pero que no estén vinculados a la RPI3, careciendo de funcion en la interfaz gráfica. Estos se han implementado para dejar el proyecto abierto a posibles mejoras.

La interfaz se ha diseñado con una dimensión de 800x500 píxeles. En ella se puede visualizar diferentes pestañas en las que se muestran las diferentes funciones. Empezando desde la pestaña derecha hacia la izquierda se observan:

- **About:** esta pestaña contiene información acerca de la interfaz gráfica como quién la ha diseñado, por quién ha estado supervisada durante su elaboración, softwares utilizados, fecha de realización, departamento que surpevisa el proyecto, etc.

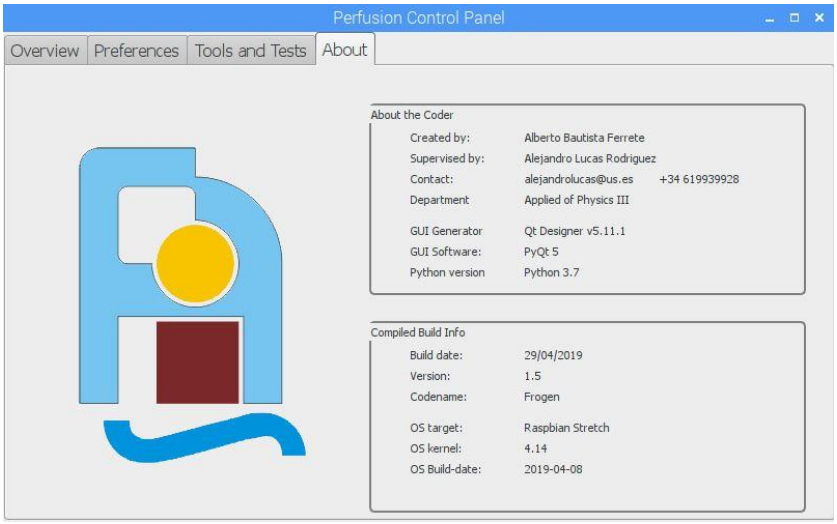


Figura 4.3: Pestataña About de la GUI.

- **Tool and Test:** en esta pestaña el usuario encuentra diferentes herramientas con las que puede interactuar con el sistema de las bombas realizando diferentes pruebas que se especifican a continuación:

[11] En el apartado de referencias se muestran las webs en las que se han descargado todos los programas.
[12] Widgets se entiende como el conjunto de botones, cuadro de textos, ventanas y cuadros combinados que forman la interfaz gráfica.

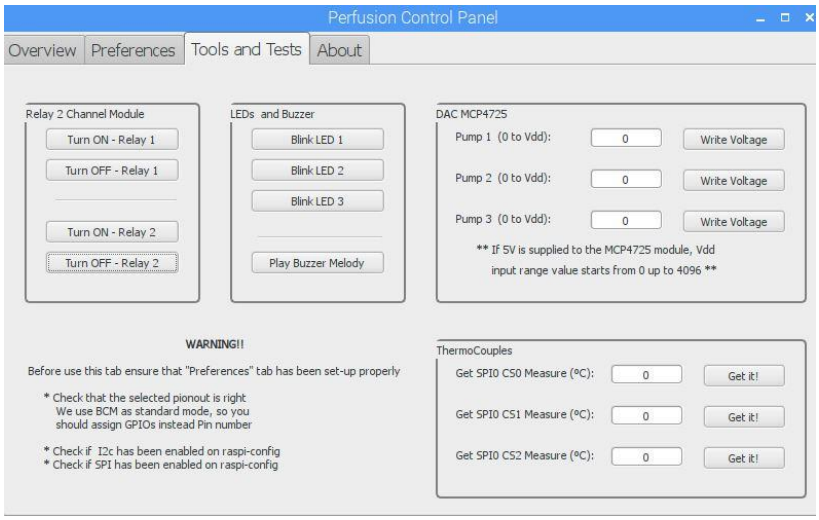


Figura 4.4: Pestaña Tool and Test de la GUI

- Relay 2 Channel Module: la interfaz dispone de 4 botones en los que el usuario puede encender (Turn ON) y apagagar (Turn OFF) los dos relés.

Figura 4.5: Botones módulo relé.

- LEDs and Buzzer: En esta sección el usuario encuentra cuatro bonotes de los cuales sólo el botón del buzzer está operativo. Al pulsar sobre él sonará una melodía a modo de aviso. El resto de botones están puesto sobre la interfaz a modo de aviso para algún evento, pero carecen de funcionalidad en este proyecto dejándolo como posible mejora.

Figura 4.6: Botones LEDs y Buzzer

- DAC MCP4725: esta sección es la encargada de interactuar con las bombas peristálticas. Contiene tres casillas, donde el usuario puede introducir por teclado valores digitales entre 0 y 4095, y tres botones (Write Voltage), donde al ser pulsados se envía una señal analógica entre 0 y 5 V (según el valor digital introducido en la casilla correspondiente y la fórmula del subapartardo 3.1.5.1) a las bombas peristálticas. Se ha diseñado para tres bombas, en el caso

de que en un futuro se quiera añadir una nueva bomba peristáltica, pero actualmente sólo las dos primeras casillas y botones, empezando desde arriba, están operativas.

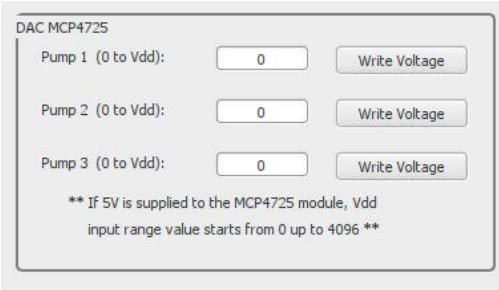


Figura 4.7: Casillas y botones de las bombas.

- ThermoCouple: en el caso de que se quiera conocer el valor de temperatura de cualquiera de los tres termopares, el usuario dispone de tres displays donde se muestran las lecturas de cada uno de los termopares en un instante concreto cada vez que se pulse el botón (Get It!).

Figura 4.8: Lectura de los termopares

- **Preferences:** esta pestaña se ha añadido como futura mejora. Se ha diseñado para que en un siguiente proyecto se pueda cambiar la configuración de cada uno de los módulos desde la interfaz gráfica y no mediante la programación. En ella se puede cambiar:
  - Cambiar la asignación de pines de cada módulo.
  - Cambiar la dirección del protocolo I2C de cada MCP4725.
  - Seleccionar el tipo de termpopar: K, J, N, R, S, T, E y B.

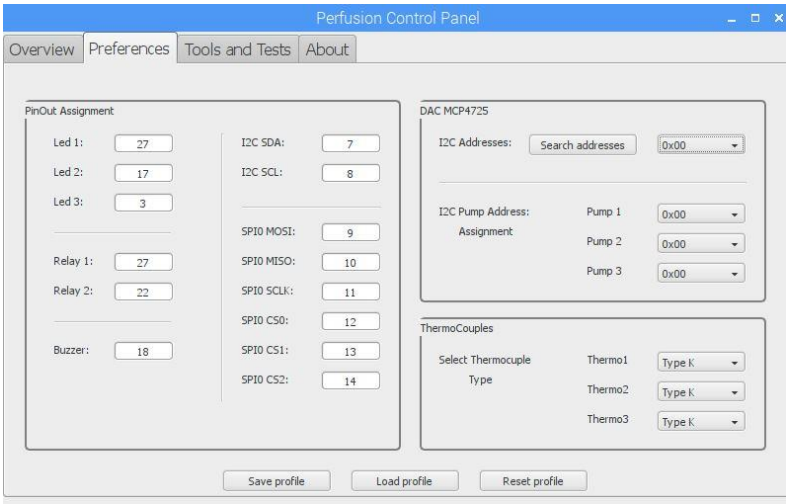


Figura 4.9: Pestaña Preferences de la GUI.

- **Overview:** en esta pestaña el usuario puede encontrar la monitorización de las temperaturas, medidas en grados Celsius, mediante una gráfica en tiepo real, medida en centésimas de segundo (cs), para visualizar los cambios de temperaturas que se puedan producir durante la experimentación.

Cabe mencionar que los valores de temperatura del eje vertical de la gráfica cambian automáticamente, amoldándose a los valores máximos y mínimos que tengan los termopares durante un intervalo de tiempo. Aparte se ha añadido bajo la gráfica una leyenda a color para facilitar el reconocimiento de cada termopar al usuario que utilice la interfaz gráfica.

Se ha añadido unos botones (STOP y START) en el caso de que se quiera parar la gráfica en un momento concreto. En este proyecto no tienen función, pero se dejan diseñados como posible mejora.

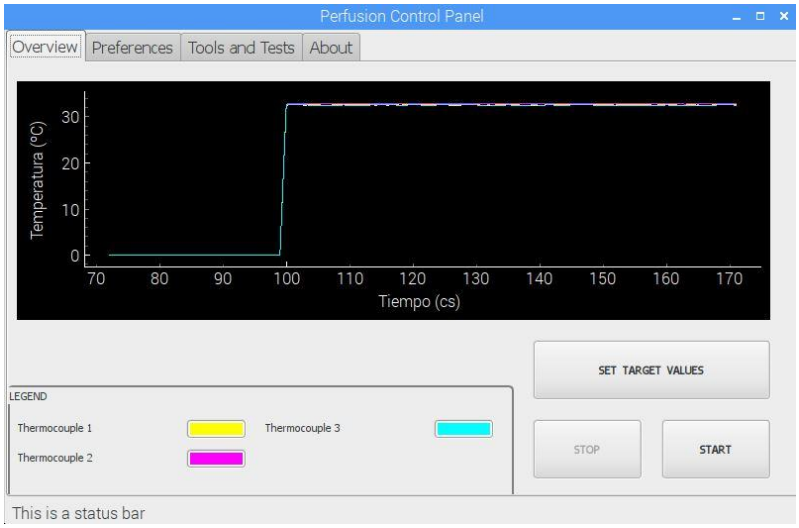


Figura 4.10: Pastaña Overview de la GUI

4.1.3 Funcionamiento

La interfaz gráfica se visualiza mediante la ejecución del código *perfusión_19.py* que se puede encontrar en la ruta *alberto/gui_code/pyqt5_gui*. Cuando se abre este fichero aparece un cuadro de diálogo en el que se pregunta qué acción se quiere realizar. Con pulsar sobre "execute" o sobre "execute in terminal" se ejecuta y se abrela interfaz gráfica.

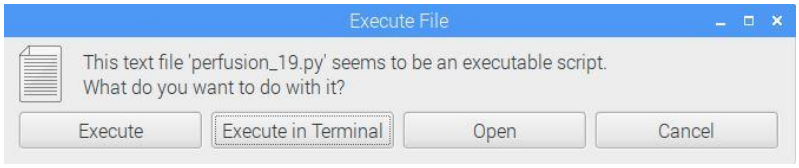


Figura 4.11: Opciones al abrir perfusión_19.py

El código *perfusión_19.py* contiene las acciones descritas en los códigos de cada módulo del subapartado 3.1.5 junto a nuevas líneas de código para algunas acciones específicas de este proyecto. En este subapartado se explicarán las líneas de código más relevantes, aunque si se desea ver el código completo, se puede encontrar en el apartado ANEXO:

- Se importan desde PyQt5 ciertas librerías para la codificación de la interfaz gráfica:

```
from PyQt5 import uic, QtWidgets, QtGui, QtCore
from PyQt5.QtGui import QPixmap
from PyQt5.QtGui import QFont
```

```
from PyQt5.QtWidgets import QApplication, QFileDialog
```

- Se le asigna a una variable el nombre del archivo en el que se creó la interfaz g´rafica con Qt Designer

```
qtCreatorFile = "interfaz_05.ui"
```

- Se crea la interfaz cargando el archivo .ui generado en Qt Designer

```
Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)
```

- Se crea una clase llamada "MyApp" que hereda de las bibliotecas de Qt e inicializa las clases principales. Esta parte siempre debe aparecer en cualquier código de Python en el que se quiera realizar una interfaz gráfica con Qt.

```
class MyApp(QtWidgets.QMainWindow, Ui_MainWindow):
    def __init__(self):
        QtWidgets.QMainWindow.__init__(self)
        Ui_MainWindow.__init__(self)
        self.closeEvent = self.closeEvent
        self.setupUi(self)
```

- Se añaden botones los cuales vinculan botones creados en la interfaz grafica (.ui) con funciones en el código *perfusión_19.py*. Nota: Se ponen algunos ejemplos no todas las vinculaciones de todos los botones.

```
self.btn_relay1_on.clicked.connect(self.buttonPress_relay1_on)
self.btn_pump_9.clicked.connect(self.getit_thermo1)
self.pushButton_buzzer.clicked.connect(self.buttonPress_buzzer)
self.btn_pump2.clicked.connect(self.buttonPress_pump2)
```

- Se definen las diferentes funciones que se ejecutan cuando se pulsa los botones correspondientes. Un ejemplo de cada módulo sería:
  - Relé: Encendido del relé 1.

```
def buttonPress_relay1_on(self):
    GPIO.setup(pin_relay1, GPIO.OUT)
    GPIO.output(pin_relay1, False)
```

  - Termopar: lectura del termopar 1 y visualización sobre el display.

```
def getit_thermo1(self):
    temp_1 = sensor_1.read_temp_c()
    internal_1 = sensor_1.read_internal_temp_c()
    self.lineEdit_34.setText(format(temp_1))
```

  - Buzzer: ejecución de la melodía del buzzer

```
def buttonPress_buzzer(self):
    buzzer = Buzzer()
    buzzer.play(3)
```

  - MCP4725: Se escribe un valor digital en la casilla (lineEdit_pump2) y al pulsar el botón (byn_pump2) dicho valor se convierte a voltaje analógico en el MCP4725.

```
def buttonPress_pump2(self):
    self.writeVoltage_b(int(self.lineEdit_pump2.text()))
```

```
    def writeVoltage_b(self, voltage):
        try:
          dac_b.raw_value = int(voltage)
          print("well")
          print(voltage)

        except:
          print("Error with I2C module")
          print(int(voltage))
```

- Se importa el código *Plotter.py* a la ventana (CustomWidget) creada enla pestaña Overview.En él se ha programado la lectura en tuempo real de cada termopar y se ha representado sobre una gráfica. Su explicación se detalla tras acabar con la descripción del código *perfusión_19.py*.

```
from Plotter import CustomWidget
```

- El código principal crea una nueva aplicación de interfaz gráfica en la que se hace una llamada a la clase creada y se muestra.

```
if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    #We call 'MyApp' class
    window = MyApp()
    #Show the generated form
    window.show()
    sys.exit(app.exec_())
```

Para visualizar la monitorización de las temperaturas de los termopares se ha elaborado un código denominado *Plotter.py,* el cual se ha importado sobre el programa *perfusión_19.py* por lo que no hace falta ejecutarlo de forma individual. A continuacións explicarán las líneas de código más relevantes, aunque si se desea ver el código completo, se puede encontrar en el apartado ANEXO:

- Se importan las librerías pyqtgraph para elaborar representaciones gráficas, y numpy para el procesamiento de gran cantidad de datos.

```
from numpy import *
from pyqtgraph.Qt import QtGui, QtCore
```

- Se crea la clase “CustomWidget” donde se visualizará la gráfica. Se configura el fondo de la gráfica de color negro y se inicializa el tiempo en 0

```
class  CustomWidget(pg.GraphicsWindow):
    pg.setConfigOption('foreground', 'w')
    ptr1 = 0
```

- Dentro de esta clase se configuran aspectos como título de la gráfica, títulos de los ejes, inicialización de 3 matrices donde se guardarán las lecturas de los termopares (self.data), legenda de la gráfica y velocidad de lectura.

```
    def __init__(self, parent=None, **kargs):
        pg.GraphicsWindow.__init__(self, **kargs)
```

```
        self.setParent(parent)
        self.setWindowTitle('pyqtgraph example: Scrolling Plots')
        p1 = self.addPlot(labels = {'left':'Temperatura (°C)',
'bottom':'Tiempo (s'})
        self.data1 = np.random.normal(size=100)
        self.data2 = np.random.normal(size=100)
        self.data3 = np.random.normal(size=100)

        self.curve1 = p1.plot(self.data1, pen='y', name="termpopar 1")
        self.curve2 = p1.plot(self.data2, pen='m', name="termpopar 2")
        self.curve3 = p1.plot(self.data3, pen='c', name="termpopar 3")

        timer = pg.QtCore.QTimer(self)
        timer.timeout.connect(self.update)
        timer.start(100) # number of seconds  for next update
```

- Tras la configuración se crea una función en la que los datos de la lectura se actualizan junto al tiempo (ptr) y los antiguos se van desplazando hacia la izquierda. Esta parte se repite para cada uno de los tres termopares

```
    def update(self):
        self.data1[:-1] = self.data1[1:]  # shift data in the array one
sample left
        temp_1 = sensor_1.read_temp_c()
        self.data1[-1] = float(temp_1)
        self.ptr1 += 1
        self.curve1.setData(self.data1)
        self.curve1.setPos(self.ptr1, 0)
```

- El código principal crea una gráfica en la que se hace una llamada a la clase creada y se muestra.

```
if __name__ == '__main__':
    w = CustomWidget()
    w.show()
    QtGui.QApplication.instance().exec_()
```

5 CONCLUSIONES Y LÍNEAS FUTURAS

5.1 Conclusiones

Tras haber realizado este proyecto se ha comprobado que una interfaz gráfica facilita y mejora el trabajo de experimentación con el proceso de crioperservación. El usuario realiza las pruebas pertinentes en un menor tiempo y de manera más cómoda ya que la interfaz hace la experimentación más dinámica entre el usuario y el software. Así, cualquier persona que trabaje en el laboratorio no requiere de conocimientos previos sobre informática y electrónica para poder configurar y utilizar las bombas.

Por otro lado, durante la elaboración se ha encontrado numerosos inconvenientes que han costado resolver y han ralentizado los plazos de ejecución.

En primer lugar, los módulos no estaban soldados por lo que se tuvo que aprender durante varios días cómo soldar dispositivos electrónicos. Seguido, la puesta en marcha de cada uno de estos módulos para comprobar su correcto funcionamiento y descartar que estuvieran defectuosos no fue tarea sencilla. Python es un lenguaje de programación relativamente sencillo, pero como nunca antes se ha había trabajado con él, costó tiempo acomodarse a utilizar su estructura, funciones y variables. Además, ciertos módulos tienen protocolos de comunicación específicos que se tuvieron que habilitar en la configuración de la Raspberry y librerías que tuvieron que ser importadas en los códigos para poderlos programar conforme a funciones específicas y que actuaran según las especificaciones que debía cumplir la interfaz grafica como por ejemplo la lectura de los termopares o el envio de la señal analógoca a las bombas a través del MCP4725.

Una vez comprobado que todos los módulos funcionaban correctamente, se pasó a diseñar una placa PCB. De nuevo, nunca antes se había trabajado con el programa EAGLE y se invirtió mucho en aprender a diseñar el esquema de cada módulo, sus huellas, teniendo encuenta sus medidas y la distancia entre pines de cada uno, y el ruteado para que ninguna pista se cruzara con otra y provocase un cortocircuito.

Todas estas precauciones había que tenerlas presentes ya que más tarde había que imprimir la PCB y contruirla con las medidas reales de cada módulo.

Llegados hasta este punto, ahora sí se podía empezar a crear la interfaz gráfica. Los mayores problemas en esta parte se centraron en vincular cada Widget (conjunto de botones, cuadro de textos, ventanas y cuadros combinados que forman la interfaz gráfica) a la función de Python que hacia actuar a cada módulo. Existe información y tutoriales sobre cómo crear una interfaz gráfica utilizando PyQt5 y Qt Creator, pero la gran mayoría son ejemplos muy sencillos que no se aproximan a la dificultad de este proyecto. Debido a esto para algunas acciones de la interfaz gráfica se consiguieron a base de pruebas, recopilando información de la web oficial de Qt[13].

La realización de la monitorización, es decir, la lectura de cada uno de los tres termopares y la posterior representación sobre una gráfica en tiempo real fue el mayor de los problemas. En un principio se pensó que mediante Qt Creator existía algún método para representar lo que los tres termopares recopilaban sobre una misma gráfica. Resultó no ser así. La solución pasaba por crear una ventana en Qt Creator denominada "CustomWidget" sobre la que se importaría un código aparte programado en Python que realizara la lectura y la representación en tiempo real (Plotter.py). Para la realización de dicho código se consideraron diferentes alternativas:

- Matplotlib
- Pyqtgraph

De las dos opciones se decantó por Pyqygraoh ya que tiene un mejor funcionamiento cuando se trata gran cantidad de datos que se quieren representar.

[13] https://www.qt.io/

Para finalizar hay mucho trabajo por delante para mejorar la información sobre la creación de interfaces gráficas con software gratuito mediante los medios utilizados en este proyecto, pero aún así se ha logrado crear una interfaz gráfica funcional que cumpla con los requisitos especificados. Junto a esta memoria se adjunta unos vídeos en los que se puede visualizar el correcto funcionamiento de la interfaz gráfica.

Líneas futuras

La finalidad de este trabajo fin de grado es actualizar un sistema obsoleto que hace uso de unas aplicaciones en deshuso y tras él el proyecto es escalable y abierto a posibles mejoras, es decir cualquier persona que lo desee puede actuar sobre el sistema implementado y modificar el comportamiento del proyecto con pocos conocimientos.

Una de las posibles mejoras sería el ajuste automático de la funcion de transferencia que caracteriza el sistema con la incorporación del caudalímetro que se puede observar en la imagen, para saber cuánto líquido circula por los tubos. Con él se podría implementar un sistema de control PID para corregir las desviaciones en tiempo real que se producen con la cantidad de líquido que atraviesa los tubos y autoajustar la función de transferencia del sistema.

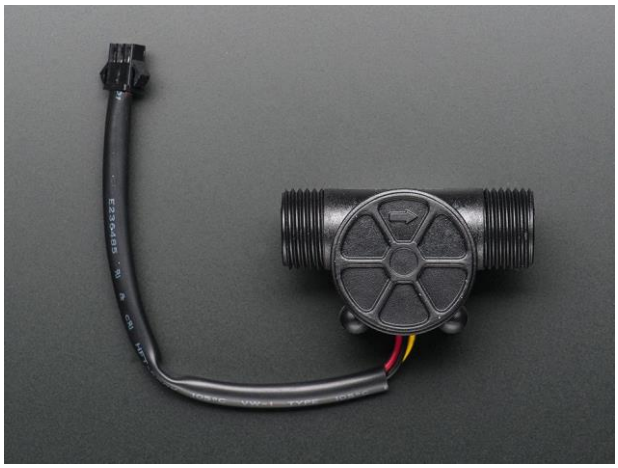


Figura 5.1: Medidor de flujo de líquido (caudalímetro)

Tambien existen unas zonas muertas en las que las bombas no funcionan ya que dependen del rozamiento que produce la carga con la que se esté trabajando. Con la implementación de un control PID se podría resolver ya que se sube el voltaje hasta que las bombas logran arrancar y empieza a medirse fluido.

Por último, la mejora más llamativa y que no se ha podido llevar a cabo por falta de tiemo sería la ampliación de la interfaz gráfica con una nueva pestaña llamada “Protocols” en la que el usuario podría elegir qúe experimento de concentraciones se quiere llevar a cabo: experimento en meseta o experimento en rampa.

ANEXOS

ANEXO A: CÓDIGOS

Código: test_general.py

```
#!/usr/bin/env python3
import RPi.GPIO as GPIO
# Global Imports
import logging          #para los termopares
import time             #para el tiempo
import Adafruit_GPIO    #para los termopares
import board            #para los MCP4725
import busio            #para los MCP4725
import adafruit_mcp4725  #para los MCP4725

# Local Imports
from Adafruit_MAX31856 import MAX31856 as MAX31856  #para los termopares

########Para los termopares###################### solo cambiar la
configuracion
###solo cambiar la configuracion "cs" en el caso de que se quieran utilizar
otros pines
logging.basicConfig(
    filename='simpletest.log',
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
_logger = logging.getLogger(__name__)

# Uncomment one of the blocks of code below to configure your Pi to use
software or hardware SPI.

## Raspberry Pi software SPI configuration.
software_spi_1 = {"clk": 11, "cs": 7, "do": 9, "di": 10}
sensor_1 = MAX31856(software_spi=software_spi_1,
tc_type=MAX31856.MAX31856_K_TYPE)

software_spi_2 = {"clk": 11, "cs": 8, "do": 9, "di": 10}
sensor_2 = MAX31856(software_spi=software_spi_2,
tc_type=MAX31856.MAX31856_K_TYPE)

software_spi_3 = {"clk": 11, "cs": 12, "do": 9, "di": 10}
sensor_3 = MAX31856(software_spi=software_spi_3,
tc_type=MAX31856.MAX31856_K_TYPE)

# Raspberry Pi hardware SPI configuration.
#SPI_PORT   = 0
#SPI_DEVICE = 0
#sensor = MAX31856(hardware_spi=Adafruit_GPIO.SPI.SpiDev(SPI_PORT,
SPI_DEVICE), tc_type=MAX31856.MAX31856_K_TYPE)

##############################################




############Para los MCP4725##############################
```

```
# Initialize I2C bus.
i2c = busio.I2C(board.SCL, board.SDA)

# Initialize MCP4725_a.
dac_a = adafruit_mcp4725.MCP4725(i2c, address=0x60)
# Optionally you can specify a different addres if you override the A0 pin.
#amp = adafruit_max9744.MAX9744(i2c, address=0x60)

# There are a three ways to set the DAC output, you can use any of these:
dac_a.value = 65500  # Use the value property with a 16-bit number just like
                     # the AnalogOut class.  Note the MCP4725 is only a 12-bit
                     # DAC so quantization errors will occur.  The range of
                     # values is 0 (minimum/ground) to 65535 (maximum/Vout).

dac_a.raw_value = 4095  # Use the raw_value property to directly read and
write
                        # the 12-bit DAC value.  The range of values is
                        # 0 (minimum/ground) to 4095 (maximum/Vout).

dac_a.normalized_value = 1.0  # Use the normalized_value property to set the
                              # output with a floating point value in the range
                              # 0 to 1.0 where 0 is minimum/ground and 1.0 is
                              # maximum/Vout.


# Initialize MCP4725_b.
dac_b = adafruit_mcp4725.MCP4725(i2c, address=0x61)
# Optionally you can specify a different addres if you override the A0 pin.
#amp = adafruit_max9744.MAX9744(i2c, address=0x60)

# There are a three ways to set the DAC output, you can use any of these:
dac_b.value = 65500  # Use the value property with a 16-bit number just like
                     # the AnalogOut class.  Note the MCP4725 is only a 12-bit
                     # DAC so quantization errors will occur.  The range of
                     # values is 0 (minimum/ground) to 65535 (maximum/Vout).

dac_b.raw_value = 4095  # Use the raw_value property to directly read and
write
                        # the 12-bit DAC value.  The range of values is
                        # 0 (minimum/ground) to 4095 (maximum/Vout).

dac_b.normalized_value = 1.0  # Use the normalized_value property to set the
                              # output with a floating point value in the range
                              # 0 to 1.0 where 0 is minimum/ground and 1.0 is
                              # maximum/Vout.


#########################################################




## Establecemos el sistema de numeracion que queramos, en mi caso BCM
GPIO.setmode(GPIO.BCM)

## Definimos pines a usar
RelayPin_1 = 27
RelayPin_2 = 22

led_pins = [5,6,13]

## Configurar GPIOS como salidas
```

```
GPIO.setup(RelayPin_1, GPIO.OUT)
GPIO.setup(RelayPin_2, GPIO.OUT)


for x in led_pins:
    GPIO.setup(x, GPIO.OUT)
    GPIO.output(x, GPIO.LOW)


######################################################################
####  DEFINICION DE CLASES COMO OBJETOS                          #########
######################################################################
class Buzzer(object):
 def __init__(self):
  GPIO.setmode(GPIO.BCM)
  self.buzzer_pin = 18 #set to GPIO pin 18
  GPIO.setup(self.buzzer_pin, GPIO.IN)
  GPIO.setup(self.buzzer_pin, GPIO.OUT)
  print("buzzer ready")

 def __del__(self):
  class_name = self.__class__.__name__
  print (class_name, "finished")

 def buzz(self,pitch, duration):   #create the function “buzz” and feed it
the pitch and duration)

  if(pitch==0):
   time.sleep(duration)
   return
  period = 1.0 / pitch     #in physics, the period (sec/cyc) is the inverse
of the frequency (cyc/sec)
  delay = period / 2     #calcuate the time for half of the wave
  cycles = int(duration * pitch)   #the number of waves to produce is the
duration times the frequency

  for i in range(cycles):    #start a loop from 0 to the variable “cycles”
calculated above
   GPIO.output(self.buzzer_pin, True)   #set pin 18 to high
   time.sleep(delay)    #wait with pin 18 high
   GPIO.output(self.buzzer_pin, False)    #set pin 18 to low
   time.sleep(delay)    #wait with pin 18 low

 def play(self, tune):
  GPIO.setmode(GPIO.BCM)
  GPIO.setup(self.buzzer_pin, GPIO.OUT)
  x=0

  print("Playing tune ",tune)
  if(tune==1):
    pitches=[262,294,330,349,392,440,494,523, 587, 659,698,784,880,988,1047]
    duration=0.1
    for p in pitches:
      self.buzz(p, duration)  #feed the pitch and duration to the function,
“buzz”
      time.sleep(duration *0.5)
    for p in reversed(pitches):
      self.buzz(p, duration)
      time.sleep(duration *0.5)

  elif(tune==2):
    pitches=[262,330,392,523,1047]
```

```
    duration=[0.2,0.2,0.2,0.2,0.2,0,5]
    for p in pitches:
      self.buzz(p, duration[x])  #feed the pitch and duration to the
function, "buzz"
      time.sleep(duration[x] *0.5)
      x+=1
  elif(tune==3):
    pitches=[392,294,0,392,294,0,392,0,392,392,392,0,1047,262]
    duration=[0.2,0.2,0.2,0.2,0.2,0.2,0.1,0.1,0.1,0.1,0.1,0.1,0.8,0.4]
    for p in pitches:
      self.buzz(p, duration[x])  #feed the pitch and duration to the func$
      time.sleep(duration[x] *0.5)
      x+=1

  elif(tune==4):
    pitches=[1047, 988,659]
    duration=[0.1,0.1,0.2]
    for p in pitches:
      self.buzz(p, duration[x])  #feed the pitch and duration to the func$
      time.sleep(duration[x] *0.5)
      x+=1

  elif(tune==5):
    pitches=[1047, 988,523]
    duration=[0.1,0.1,0.2]
    for p in pitches:
      self.buzz(p, duration[x])  #feed the pitch and duration to the func$
      time.sleep(duration[x] *0.5)
      x+=1

  GPIO.setup(self.buzzer_pin, GPIO.IN)


##########################################################
###############    INICIO DEL PROGRAMA        #######
##########################################################

if __name__ == "__main__":

 try:

   #Reproducirá infinatemente el test de los dispositivos
   while True:

        ########################################################
        #Testeando relés (USa logica inversa)
        #por lo que cuando el voltaje que se manda es 0
        #el modulo se activa y si se mandan 5v se desactiva
        ########################################################
        print ("Encendiendo Rele 1...")
        GPIO.output(RelayPin_1, False) ## Enciendo el RelayPin_1
        time.sleep(3) ## Esperamos 3 segundos
        print ("Apagando Rele 1...")
        GPIO.output(RelayPin_1, True) ## Apago el RelayPin_1

        time.sleep(2) ## Esperamos 2 segundos

        print ("Encendiendo Rele 2...")
        GPIO.output(RelayPin_2, False) ## Enciendo el RelayPin_2
        time.sleep(3) ## Esperamos 3 segundos
        print ("Apagando Rele 2...")
        GPIO.output(RelayPin_2, True) ## Apago el RelayPin_2
```

```
        #Esperamos 2 segundos y pasamos al siguiente test
        time.sleep(2) ## Esperamos 2 segundos


        ######################################################
        #Testeando LEDS
        ######################################################
        # Turn on this GPIO then off
        for x in led_pins:
            print ("Encendiendo led del pin %s" % (str(x)) )
            GPIO.output(x, GPIO.HIGH)
            time.sleep(1)
            print ("Apagando led del pin %s" % (str(x)) )
            GPIO.output(x, GPIO.LOW)
            time.sleep(1)

        #Esperamos 2 segundos y pasamso al siguiente test
        # No LED GPIOS now (Estamos fuera del bucle)
        time.sleep(2) ## Esperamos 2 segundos



        ######################################################
        # Test dle buzzer
        ######################################################
        #Inicializamos el objeto de la clase Buzzer()
        buzzer = Buzzer()
        #Llamasmo a la funcion PLay de la clase Buzzer diciendole
        #que queremso reproducir la melodia 3 de las 5 que nos deja
        print ("Reproduciendo melodia")
        buzzer.play(3)
        print ("Melodia finalizada")


        #Esperamos 2 segundos y pasamso al siguiente test
        # No LED GPIOS now (Estamos fuera del bucle)
        time.sleep(2) ## Esperamos 2 segundos



        ######################################################
        # Test de los termopares
        ######################################################
        # lectura de cada termopar 4 veces cada segundo. Se utilizan 4
buvcles for
        for i in range(4):
            temp = sensor_1.read_temp_c()
            internal = sensor_1.read_internal_temp_c()
            print('Thermocouple Temperature_1: {0:0.3F}*C'.format(temp))
            print('    Internal Temperature_1: {0:0.3F}*C'.format(internal))
            print ("\n") #New line spacing
            time.sleep(1.0) #Sleep the main thread 1 s

        for j in range(4):
            temp = sensor_2.read_temp_c()
            internal = sensor_2.read_internal_temp_c()
            print('Thermocouple Temperature_2: {0:0.3F}*C'.format(temp))
            print('    Internal Temperature_2: {0:0.3F}*C'.format(internal))
            print ("\n") #New line spacing
            time.sleep(1.0) #Sleep the main thread 1 s
```

```
        for k in range(4):
            temp = sensor_3.read_temp_c()
            internal = sensor_3.read_internal_temp_c()
            print('Thermocouple Temperature_3: {0:0.3F}*C'.format(temp))
            print('    Internal Temperature_3: {0:0.3F}*C'.format(internal))
            print ("\n") #New line spacing
            time.sleep(1.0) #Sleep the main thread 1 s

        ######################################################
        # Test de los MCP4725. Conectar las bombas a las bornas de la placa
        ######################################################


        #MPP4725_a
        # Go up the 12-bit raw range.
        print(' MCP4725_a Going up 0-3.3V...')
        for i in range(4095):
            dac_a.raw_value = i
        time.sleep(2.0) #Sleep the main thread 2 s
        # Go back down the 12-bit raw range.
        print(' MCP4725_a Going down 3.3-0V...')
        for i in range(4095, -1, -1):
            dac_a.raw_value = i
        time.sleep(3.0) #Sleep the main thread 3 s


        #MCP4725_b
        # Go up the 12-bit raw range.
        print(' MCP4725_b Going up 0-3.3V...')
        for i in range(4095):
            dac_b.raw_value = i
        time.sleep(2.0) #Sleep the main thread 2 s
        # Go back down the 12-bit raw range.
        print(' MCP4725_b Going down 3.3-0V...')
        for i in range(4095, -1, -1):
            dac_b.raw_value = i
        time.sleep(3.0) #Sleep the main thread 1 s



except KeyboardInterrupt:
      #Para parar el codigo basta con hacer CTRL+C
      print ("\nKeyboardInterrupt has been caught.")
      # Hago una limpieza de los GPIO
      print ("GPio cleanup")
      GPIO.cleanup()
```

Código: perfusion_19.py

```
#!/usr/bin/env python3


#We import the libraries
import logging
import Adafruit_GPIO
import sys
import time
import inspect, os
import board
import busio
import RPi.GPIO as GPIO
import adafruit_mcp4725
from PyQt5 import uic, QtWidgets, QtGui, QtCore
from PyQt5.QtGui import QPixmap
from PyQt5.QtGui import QFont
from PyQt5.QtWidgets import QApplication, QFileDialog


from Adafruit MAX31856 import MAX31856 as MAX31856

#Define Default Pinout
#It'll be loaded by default

pin led1 = 23
pin_led2 = 17
pin_led3 = 3

pin_relay1 = 27
pin_relay2 = 22 #alberto

pin_buzzer = 18

pin_i2c_sda = 7
pin_i2c_scl = 8

pin_spi0_mosi = 9
pin_spi0_miso = 10
pin_spi0_sclk = 11
pin_spi0_cs0  = 12
pin_spi0_cs1  = 13
pin_spi0_cs2  = 14

i2c_pump1_addr = '0x00'
i2c_pump2_addr = '0x00'
i2c_pump3_addr = '0x00'

thermo1_type = 'Type K'
thermo2_type = 'Type K'
thermo3_type = 'Type K'

#Define the lang strings
appTitle = 'Perfusion Control Panel'


#Define the system variables
appWidth = 800
appHeight = 500
```

```
#Define the GUI and init libraries
qtCreatorFile = "interfaz_05.ui"


#We create the GUI using .UI Qt designer file
Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)

#Define GPIO board mode
GPIO.setmode(GPIO.BCM)

# Initialize I2C bus.
i2c = busio.I2C(board.SCL, board.SDA)


# Initialize MCP4725.
dac_a = adafruit_mcp4725.MCP4725(i2c, address=0x60)
dac_b = adafruit_mcp4725.MCP4725(i2c, address=0x61)

# Optionally you can specify a different addres if you override the A0 pin.
#amp = adafruit_max9744.MAX9744(i2c, address=0x60)

# There are a three ways to set the DAC output, you can use any of these:
dac_a.value = 65500  # Use the value property with a 16-bit number just like
                     # the AnalogOut class.  Note the MCP4725 is only a 12-bit
                     # DAC so quantization errors will occur.  The range of
                     # values is 0 (minimum/ground) to 65535 (maximum/Vout).

dac_a.raw_value = 4095  # Use the raw_value property to directly read and
write
                        # the 12-bit DAC value.  The range of values is
                        # 0 (minimum/ground) to 4095 (maximum/Vout).

dac_a.normalized_value = 1.0  # Use the normalized_value property to set the
                              # output with a floating point value in the range
                              # 0 to 1.0 where 0 is minimum/ground and 1.0 is
                              # maximum/Vout.


# There are a three ways to set the DAC output, you can use any of these:
dac_b.value = 65500  # Use the value property with a 16-bit number just like
                     # the AnalogOut class.  Note the MCP4725 is only a 12-bit
                     # DAC so quantization errors will occur.  The range of
                     # values is 0 (minimum/ground) to 65535 (maximum/Vout).

dac_b.raw_value = 4095  # Use the raw_value property to directly read and
write
                        # the 12-bit DAC value.  The range of values is
                        # 0 (minimum/ground) to 4095 (maximum/Vout).

dac_b.normalized_value = 1.0  # Use the normalized_value property to set the
                              # output with a floating point value in the range
                              # 0 to 1.0 where 0 is minimum/ground and 1.0 is
                              # maximum/Vout.

# Use the raw_value property to directly read and write
# the 12-bit DAC value.  The range of values is
# 0 (minimum/ground) to 4095 (maximum/Vout).
##dac.raw_value = 4095


#alberto-------------------------------------------------
logging.basicConfig(
```

```
    filename='simpletest.log',
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
_logger = logging.getLogger(__name__)

# Uncomment one of the blocks of code below to configure your Pi to use
software or hardware SPI.

## Raspberry Pi software SPI configuration.
software_spi_1 = {"clk": 11, "cs": 7, "do": 9, "di": 10}
sensor_1 = MAX31856(software_spi=software_spi_1,
tc_type=MAX31856.MAX31856_K_TYPE)

software_spi_2 = {"clk": 11, "cs": 8, "do": 9, "di": 10}
sensor_2 = MAX31856(software_spi=software_spi_2,
tc_type=MAX31856.MAX31856_K_TYPE)

software_spi_3 = {"clk": 11, "cs": 12, "do": 9, "di": 10}
sensor_3 = MAX31856(software_spi=software_spi_3,
tc_type=MAX31856.MAX31856_K_TYPE)
#-----------------------------------------------------------



class Buzzer(object):
 def __init__(self):
  GPIO.setmode(GPIO.BCM)
  self.buzzer_pin = 18 #set to GPIO pin 18
  GPIO.setup(self.buzzer_pin, GPIO.IN)
  GPIO.setup(self.buzzer_pin, GPIO.OUT)
  print("buzzer ready")

 def __del__(self):
  class_name = self.__class__.__name__
  print (class_name, "finished")

 def buzz(self,pitch, duration):   #create the function “buzz” and feed it
the pitch and duration)

  if(pitch==0):
   time.sleep(duration)
   return
  period = 1.0 / pitch     #in physics, the period (sec/cyc) is the inverse
of the frequency (cyc/sec)
  delay = period / 2     #calcuate the time for half of the wave
  cycles = int(duration * pitch)   #the number of waves to produce is the
duration times the frequency

  for i in range(cycles):    #start a loop from 0 to the variable “cycles”
calculated above
   GPIO.output(self.buzzer_pin, True)   #set pin 18 to high
   time.sleep(delay)    #wait with pin 18 high
   GPIO.output(self.buzzer_pin, False)    #set pin 18 to low
   time.sleep(delay)    #wait with pin 18 low

 def play(self, tune):
  GPIO.setmode(GPIO.BCM)
  GPIO.setup(self.buzzer_pin, GPIO.OUT)
  x=0

  print("Playing tune ",tune)
  if(tune==3):
```

```
    pitches=[392,294,0,392,294,0,392,0,392,392,392,0,1047,262]
    duration=[0.2,0.2,0.2,0.2,0.2,0.2,0.1,0.1,0.1,0.1,0.1,0.1,0.8,0.4]
    for p in pitches:
      self.buzz(p, duration[x])  #feed the pitch and duration to the func$
      time.sleep(duration[x] *0.5)
      x+=1



  GPIO.setup(self.buzzer_pin, GPIO.IN)

################################################
#     We create a class to Manage the GUI      ##
################################################

class MyApp(QtWidgets.QMainWindow, Ui_MainWindow):

    #Default Function that will be executed when we
    #call the 'MyApp' class
    def __init__(self):
        QtWidgets.QMainWindow.__init__(self)
        Ui_MainWindow.__init__(self)
        self.closeEvent = self.closeEvent
        self.setupUi(self)

        #Configure the Main Windows that it's going to be shown
        self.setWindowTitle(appTitle)    #Set the main window title
        self.resize(appWidth, appHeight) #Set the main window size

        #Add a status bar to notify the user what we are doing
        self.statusBar().showMessage('This is a status bar')

        #Load default profile (Init GUI Values)
        self.loadDefaultProfile()


        #Add button events (connect .ui element with button fcn action)
        self.btn_start.clicked.connect(self.buttonPress_Start)
        self.btn_stop.clicked.connect(self.buttonPress_Stop)

        self.btn_relay1_on.clicked.connect(self.buttonPress_relay1_on)
        self.btn_relay1_off.clicked.connect(self.buttonPress_relay1_off)
        self.btn_relay2_on.clicked.connect(self.buttonPress_relay2_on)
        self.btn_relay2_off.clicked.connect(self.buttonPress_relay2_off)

        #alberto-----------------------------------------------------------
---
        self.btn_pump_9.clicked.connect(self.getit_thermo1)
        self.pushButton_28.clicked.connect(self.getit_thermo2)
        self.pushButton_27.clicked.connect(self.getit_thermo3)

        self.pushButton_buzzer.clicked.connect(self.buttonPress_buzzer)
        #-----------------------------------------------------------------
---

        self.btn_saveProfile.clicked.connect(self.buttonPress_saveProfile)
        self.btn_loadProfile.clicked.connect(self.buttonPress_loadProfile)
        self.btn_resetProfile.clicked.connect(self.buttonPress_resetProfile)

        self.btn_blink_led1.clicked.connect(self.buttonPress_blink_led1)
        self.btn_blink_led2.clicked.connect(self.buttonPress_blink_led2)
        self.btn_blink_led3.clicked.connect(self.buttonPress_blink_led3)
```

```
        self.btn_pump1.clicked.connect(self.buttonPress_pump1)
        self.btn_pump2.clicked.connect(self.buttonPress_pump2)
        self.btn_pump3.clicked.connect(self.buttonPress_pump3)

        # Load default .conf data if exists
        self.buttonPress_loadProfile()


    #DEFINES FUNCTION ACTIONS FOR ELEMENTS (BUTTONS; ETC)#

    ############################################################
    ##            'TOOLS AND TEST' TAB (BUTTONS ACTIONS      ##
    ############################################################
    def buttonPress_relay1_on(self):
        GPIO.setup(pin_relay1, GPIO.OUT)
        GPIO.output(pin_relay1, False)

    def buttonPress_relay1_off(self):
        GPIO.setup(pin_relay1, GPIO.OUT)
        GPIO.output(pin_relay1, True)


    def buttonPress_relay2_on(self):
        GPIO.setup(pin_relay2, GPIO.OUT)
        GPIO.output(pin_relay2, False)

    def buttonPress_relay2_off(self):
        GPIO.setup(pin_relay2, GPIO.OUT)
        GPIO.output(pin_relay2, True)

#alberto-----------------------------------------------------

    def getit_thermo1(self):
        temp_1 = sensor_1.read_temp_c()
        internal_1 = sensor_1.read_internal_temp_c()
        self.lineEdit_34.setText(format(temp_1))
        #self.lineEdit_34.setText('5')

    def getit_thermo2(self):
        temp_2 = sensor_2.read_temp_c()
        internal_2 = sensor_2.read_internal_temp_c()
        self.lineEdit_35.setText(format(temp_2))
##        #self.lineEdit_34.setText('5')
##
    def getit_thermo3(self):
        temp_3 = sensor_3.read_temp_c()
        internal_3 = sensor_3.read_internal_temp_c()
        self.lineEdit_36.setText(format(temp_3))
##        #self.lineEdit_34.setText('5')


    def buttonPress_buzzer(self):
        buzzer = Buzzer()
        buzzer.play(3)


#----------------------------------------------------------

    def buttonPress_blink_led1(self):
        GPIO.setup(pin_led1, GPIO.OUT)
        GPIO.output(pin_led1, True)
```

```
        time.sleep(1)
        GPIO.output(pin_led1, False)

    def buttonPress_blink_led2(self):
        GPIO.setup(pin_led2, GPIO.OUT)
        GPIO.output(pin_led2, False)
        time.sleep(1)
        GPIO.output(pin_led2, True)

    def buttonPress_blink_led3(self):
        GPIO.setup(pin_led3, GPIO.OUT)
        GPIO.output(pin_led3, True)
        time.sleep(1)
        GPIO.output(pin_led3, False)

    def buttonPress_pump1(self):
##        print('Hola, el valor elegido es: ' +
int(self.lineEdit_pump1.text()))
        print('Hola, el valor elegido es: ' + self.lineEdit_pump1.text())
        self.writeVoltage_a(self.lineEdit_pump1.text())

    def buttonPress_pump2(self):
        self.writeVoltage_b(int(self.lineEdit_pump2.text()))

    def buttonPress_pump3(self):
        self.writeVoltage(int(self.lineEdit_pump3.text()))

    ############################################################
    ##            'Preferences' TAB (BUTTONS ACTIONS          ##
    ############################################################


    def buttonPress_saveProfile(self):

        #We save the config file with the user parameters
        with open('config//' + 'profile.conf', 'w') as f:
              f.write('led1 = '+ self.lineEdit_led1.text()+ '\r')
              f.write('led2 = '+ self.lineEdit_led2.text()+ '\r')
              f.write('led3 = '+ self.lineEdit_led3.text()+ '\r')
              f.write('relay1 = '+ self.lineEdit_relay1.text()+ '\r')
              f.write('relay2 = '+ self.lineEdit_relay2.text()+ '\r')
              f.write('buzzer = '+ self.lineEdit_buzzer.text()+ '\r')
              f.write('i2c_sda = '+ self.lineEdit_i2c_sda.text()+ '\r')
              f.write('i2c_scl = '+ self.lineEdit_i2c_scl.text()+ '\r')
              f.write('spi0_mosi = '+ self.lineEdit_spi0_mosi.text()+ '\r')
              f.write('spi0_miso = '+ self.lineEdit_spi0_miso.text()+ '\r')
              f.write('spi0_sclk = '+ self.lineEdit_spi0_sclk.text()+ '\r')
              f.write('spi0_cs0 = '+ self.lineEdit_spi0_cs0.text()+ '\r')
              f.write('spi0_cs1 = '+ self.lineEdit_spi0_cs1.text()+ '\r')
              f.write('spi0_cs2 = '+ self.lineEdit_spi0_cs2.text()+ '\r')
              f.write('pump1 = '+ str(self.comboBox_i2c_pump1.currentText())+
'\r')
              f.write('pump2 = '+ str(self.comboBox_i2c_pump2.currentText())+
'\r')
              f.write('pump3 = '+ str(self.comboBox_i2c_pump3.currentText())+
'\r')
              f.write('thermo1 = '+ str(self.comboBox_thermo1.currentText())+
'\r')
              f.write('thermo2 = '+ str(self.comboBox_thermo2.currentText())+
'\r')
              f.write('thermo3 = '+ str(self.comboBox_thermo3.currentText())+
'\r')
```

```
        ## We call this function to override the globalVars with the
qelements
        self.guiToGlobalVar()


    def buttonPress_loadProfile(self):
        #Check if file exists, if exist load the default conf
        exists = os.path.isfile('config//' + 'profile.conf')

        if exists:
            # Store configuration file values
            # We read the .conf file to load the saved previous data
            file = open('config//' + 'profile.conf', mode = 'r', encoding =
'utf-8-sig')
            lines = file.readlines()
            file.close()

            for line in lines:
              line = line.split('=')
              line = [i.strip() for i in line]
              # Check line by line the header to assign the value to the
global var
              if(line[0] == 'led1'):
                self.lineEdit_led1.setText(line[1])
              if(line[0] == 'led2'):
                self.lineEdit_led2.setText(line[1])
              if(line[0] == 'led3'):
                self.lineEdit_led3.setText(line[1])
              if(line[0] == 'relay1'):
                self.lineEdit_relay1.setText(line[1])
              if(line[0] == 'relay2'):
                self.lineEdit_relay2.setText(line[1])
#alberto-------------------------------------------------------------------
--------------


#-------------------------------------------------------------------------
---------------
              if(line[0] == 'buzzer'):
                self.lineEdit_buzzer.setText(line[1])
              if(line[0] == 'i2c_sda'):
                self.lineEdit_i2c_sda.setText(line[1])
              if(line[0] == 'i2c_scl'):
                self.lineEdit_i2c_scl.setText(line[1])
              if(line[0] == 'spi0_mosi'):
                self.lineEdit_spi0_mosi.setText(line[1])
              if(line[0] == 'spi0_miso'):
                self.lineEdit_spi0_miso.setText(line[1])
              if(line[0] == 'spi0_sclk'):
                self.lineEdit_spi0_sclk.setText(line[1])
              if(line[0] == 'spi0_cs0'):
                self.lineEdit_spi0_cs0.setText(line[1])
              if(line[0] == 'spi0_cs1'):
                self.lineEdit_spi0_cs1.setText(line[1])
              if(line[0] == 'spi0_cs2'):
                self.lineEdit_spi0_cs2.setText(line[1])
              if(line[0] == 'pump1'):
                index = self.comboBox_i2c_pump1.findText(line[1],
QtCore.Qt.MatchFixedString)
                if index >= 0:
                     self.comboBox_i2c_pump1.setCurrentIndex(index)
```

```
            if(line[0] == 'pump2'):
              index = self.comboBox_i2c_pump2.findText(line[1],
QtCore.Qt.MatchFixedString)
              if index >= 0:
                   self.comboBox_i2c_pump2.setCurrentIndex(index)
            if(line[0] == 'pump3'):
              index = self.comboBox_i2c_pump3.findText(line[1],
QtCore.Qt.MatchFixedString)
              if index >= 0:
                   self.comboBox_i2c_pump3.setCurrentIndex(index)
            if(line[0] == 'thermo1'):
              index = self.comboBox_thermo1.findText(line[1],
QtCore.Qt.MatchFixedString)
              if index >= 0:
                   self.comboBox_thermo1.setCurrentIndex(index)
            if(line[0] == 'thermo2'):
              index = self.comboBox_thermo2.findText(line[1],
QtCore.Qt.MatchFixedString)
              if index >= 0:
                   self.comboBox_thermo2.setCurrentIndex(index)
            if(line[0] == 'thermo3'):
              index = self.comboBox_thermo3.findText(line[1],
QtCore.Qt.MatchFixedString)
              if index >= 0:
                   self.comboBox_thermo3.setCurrentIndex(index)

        ## We call this function to override the globalVars with the
qelements
        self.guiToGlobalVar()


    else:
        # Keep presets
        print('Config file does not exists')




def guiToGlobalVar(self):
    #We override the global variables
    pin_led1 = int(self.lineEdit_led1.text())
    pin_led2 = int(self.lineEdit_led2.text())
    pin_led3 = int(self.lineEdit_led3.text())
    pin_relay1 = int(self.lineEdit_relay1.text())
    pin_relay2 = int(self.lineEdit_relay2.text())
    pin_buzzer = int(self.lineEdit_buzzer.text())
    pin_i2c_sda = int(self.lineEdit_i2c_sda.text())
    pin_i2c_scl = int(self.lineEdit_i2c_scl.text())
    pin_spi0_mosi = int(self.lineEdit_spi0_mosi.text())
    pin_spi0_miso = int(self.lineEdit_spi0_miso.text())
    pin_spi0_sclk = int(self.lineEdit_spi0_sclk.text())
    pin_spi0_cs0 = int(self.lineEdit_spi0_cs0.text())
    pin_spi0_cs1 = int(self.lineEdit_spi0_cs1.text())
    pin_spi0_cs2 = int(self.lineEdit_spi0_cs2.text())
    i2c_pump1_addr = str(self.comboBox_i2c_pump1.currentText())
    i2c_pump2_addr = str(self.comboBox_i2c_pump2.currentText())
    i2c_pump3_addr = str(self.comboBox_i2c_pump3.currentText())
    thermo1_type = str(self.comboBox_thermo1.currentText())
    thermo2_type = str(self.comboBox_thermo1.currentText())
    thermo3_type = str(self.comboBox_thermo1.currentText())
```

```
    def buttonPress_resetProfile(self):
        self.loadDefaultProfile()


    def buttonPress_Pump_1(self):
        if self.btn_pump_1.isEnabled():
            self.btn_pump_1.setText('STOP PUMP 1')
            #self.btn_pump_1.setEnabled(False)
        else:
            self.btn_pump_1.setText('FORCE PUMP 1')
            #self.btn_pump_1.setEnabled(True)


    def buttonPress_Start(self):
        if self.btn_start.isEnabled():
            self.btn_start.setEnabled(False)
            self.btn_stop.setEnabled(True)
        else:
            self.btn_start.setEnabled(True)
            self.btn_true.setEnabled(False)


    def buttonPress_Stop(self):
        if self.btn_stop.isEnabled():
            self.btn_start.setEnabled(True)
            self.btn_stop.setEnabled(False)
        else:
            self.btn_start.setEnabled(False)
            self.btn_true.setEnabled(True)

    def loadDefaultProfile(self):
        list_thermo_types = [
            self.tr('Type K'),
            self.tr('Type J'),
            self.tr('Type N'),
            self.tr('Type R'),
            self.tr('Type S'),
            self.tr('Type T'),
            self.tr('Type E'),
            self.tr('Type B')
        ]

        list_i2c_addr = [
            self.tr('0x00'),
            self.tr('0x01'),
            self.tr('0x02')
        ]

        #First we get the current working path
        directory = os.path.dirname(os.path.realpath(__file__))

        #Load Main App "Icon"
        self.setWindowIcon(QtGui.QIcon(directory + '//res//' + 'icon.png'))

        #Load "About" tab logo
        pixmap = QPixmap(directory + '//res//' + 'logo_fIII.png')
        self.label_about_logo.setPixmap(pixmap)

        #Load ThermoCouples list
        self.comboBox_thermo1.clear()
        self.comboBox_thermo2.clear()
        self.comboBox_thermo3.clear()
```

```
        self.comboBox_thermo1.addItems(list_thermo_types)
        self.comboBox_thermo2.addItems(list_thermo_types)
        self.comboBox_thermo3.addItems(list_thermo_types)

        #Load PUMP Addresses
        self.comboBox_i2c_addr.clear()
        self.comboBox_i2c_pump1.clear()
        self.comboBox_i2c_pump2.clear()
        self.comboBox_i2c_pump3.clear()

        self.comboBox_i2c_addr.addItems(list_i2c_addr)
        self.comboBox_i2c_pump1.addItems(list_i2c_addr)
        self.comboBox_i2c_pump2.addItems(list_i2c_addr)
        self.comboBox_i2c_pump3.addItems(list_i2c_addr)

        #Load default pinout values
        #WE must convert pin integer type to strings
        self.lineEdit_led1.setText(str(pin_led1))
        self.lineEdit_led2.setText(str(pin_led2))
        self.lineEdit_led3.setText(str(pin_led3))

        self.lineEdit_relay1.setText(str(pin_relay1))
        self.lineEdit_relay2.setText(str(pin_relay2))
        self.lineEdit_buzzer.setText(str(pin_buzzer))

        self.lineEdit_i2c_sda.setText(str(pin_i2c_sda))
        self.lineEdit_i2c_scl.setText(str(pin_i2c_scl))

        self.lineEdit_spi0_mosi.setText(str(pin_spi0_mosi))
        self.lineEdit_spi0_miso.setText(str(pin_spi0_miso))
        self.lineEdit_spi0_sclk.setText(str(pin_spi0_sclk))
        self.lineEdit_spi0_cs0.setText(str(pin_spi0_cs0))
        self.lineEdit_spi0_cs1.setText(str(pin_spi0_cs1))
        self.lineEdit_spi0_cs2.setText(str(pin_spi0_cs2))


        # Load the font:
        QtGui.QFontDatabase.addApplicationFont(directory + '//res//' +
'Tahoma.ttf')
        stylesheet = open(directory + '//res//' + 'mystylesheet.qss').read()
        self.setStyleSheet(stylesheet)


    ## DACMCP4725_a Voltage writter
    def writeVoltage_a(self, voltage):
        try:
          dac_a.raw_value = int(voltage)
          print("well")
          print(voltage)

        except:
          print("Error with I2C module")
          print(int(voltage))

    ## DACMCP4725_b Voltage writter
    def writeVoltage_b(self, voltage):
        try:
          dac_b.raw_value = int(voltage)
          print("well")
          print(voltage)
```

```
        except:
          print("Error with I2C module")
          print(int(voltage))

    ## Trigger event when we close the form
    def closeEvent(self, event):
        # Free up GPIOs
        GPIO.cleanup()

#############################################################
# Default function that will be executed when        ####
# we run this program                                ####
#############################################################

from Plotter import CustomWidget

if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    #We call 'MyApp' class
    window = MyApp()
    #Show the generated form
    window.show()
    sys.exit(app.exec_())
```

Código: Plotter.py

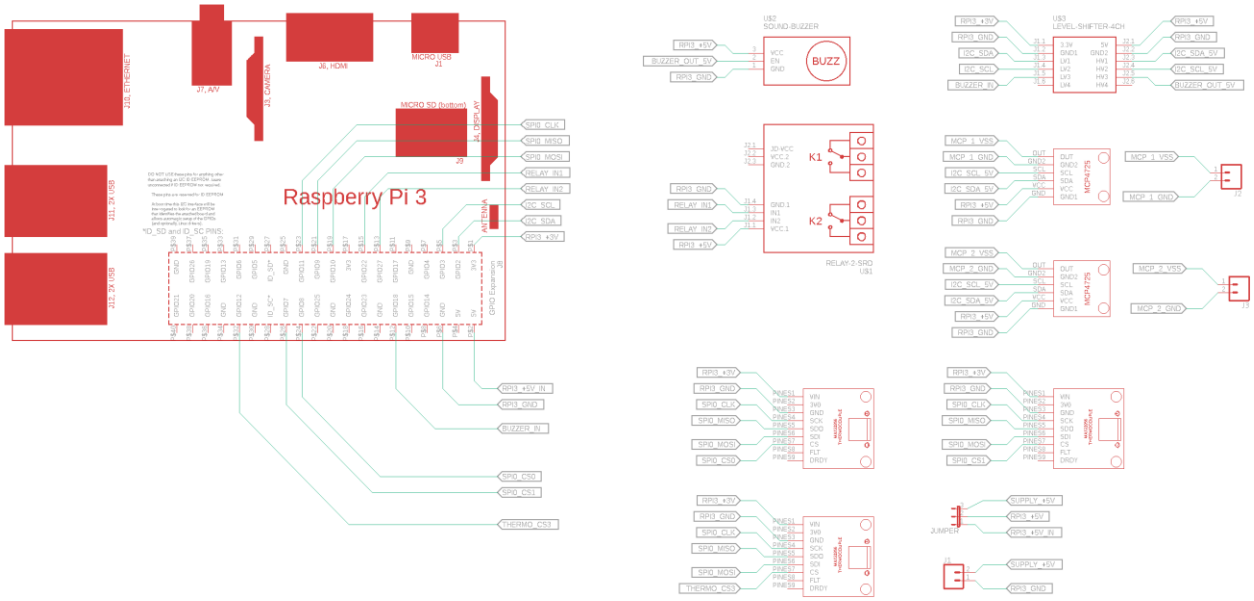
```
import pyqtgraph as pg
import numpy as np


# Global Imports

from numpy import *
from pyqtgraph.Qt import QtGui, QtCore

import logging
import time
import Adafruit_GPIO

# Local Imports
from Adafruit_MAX31856 import MAX31856 as MAX31856

logging.basicConfig(
    filename='simpletest.log',
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
_logger = logging.getLogger(__name__)

# Uncomment one of the blocks of code below to configure your Pi to use
software or hardware SPI.

## Raspberry Pi software SPI configuration.
software_spi_1 = {"clk": 11, "cs": 7, "do": 9, "di": 10}
sensor_1 = MAX31856(software_spi=software_spi_1,
tc_type=MAX31856.MAX31856_K_TYPE)

software_spi_2 = {"clk": 11, "cs": 8, "do": 9, "di": 10}
sensor_2 = MAX31856(software_spi=software_spi_2,
tc_type=MAX31856.MAX31856_K_TYPE)

software_spi_3 = {"clk": 11, "cs": 12, "do": 9, "di": 10}
sensor_3 = MAX31856(software_spi=software_spi_3,
tc_type=MAX31856.MAX31856_K_TYPE)


class  CustomWidget(pg.GraphicsWindow):
#    pg.setConfigOption('background', 'k')
    pg.setConfigOption('foreground', 'w')
    ptr1 = 0
    def __init__(self, parent=None, **kargs):
        pg.GraphicsWindow.__init__(self, **kargs)
        self.setParent(parent)
        self.setWindowTitle('pyqtgraph example: Scrolling Plots')
        p1 = self.addPlot(labels =  {'left':'Temperatura (ยบC)',
'bottom':'Tiempo (cs)'})
#        self.data1 = np.random.normal(size=250)
#        self.data2 = np.random.normal(size=250)
#        self.data3 = np.random.normal(size=250)
        self.data1 = np.zeros(100)
        self.data2 = np.zeros(100)
        self.data3 = np.zeros(100)
#        p1.addLegend()
        self.curve1 = p1.plot(self.data1, pen='y', name="termpopar 1")
        self.curve2 = p1.plot(self.data2, pen='m', name="termpopar 2")
        self.curve3 = p1.plot(self.data3, pen='c', name="termpopar 3")
#        p1.setRange(yRange=[20, 40])
```

```
        timer = pg.QtCore.QTimer(self)
        timer.timeout.connect(self.update)
        timer.start(100) # number of seconds  for next update

    def update(self):
        self.data1[:-1] = self.data1[1:]  # shift data in the array one
sample left
                            # (see also: np.roll)
        temp_1 = sensor_1.read_temp_c()
        self.data1[-1] = float(temp_1)
        self.ptr1 += 1
        self.curve1.setData(self.data1)
        self.curve1.setPos(self.ptr1, 0)
        self.data2[:-1] = self.data2[1:]  # shift data in the array one
sample left
                            # (see also: np.roll)
        temp_2 = sensor_2.read_temp_c()
        self.data2[-1] = float(temp_2)
        self.curve2.setData(self.data2)
        self.curve2.setPos(self.ptr1,0)

        self.data3[:-1] = self.data3[1:]  # shift data in the array one
sample left
                            # (see also: np.roll)
        temp_3 = sensor_3.read_temp_c()
        self.data3[-1] = float(temp_3)
        self.curve3.setData(self.data3)
        self.curve3.setPos(self.ptr1,0)


if __name__ == '__main__':
    w = CustomWidget()
    w.show()
    QtGui.QApplication.instance().exec_()
```

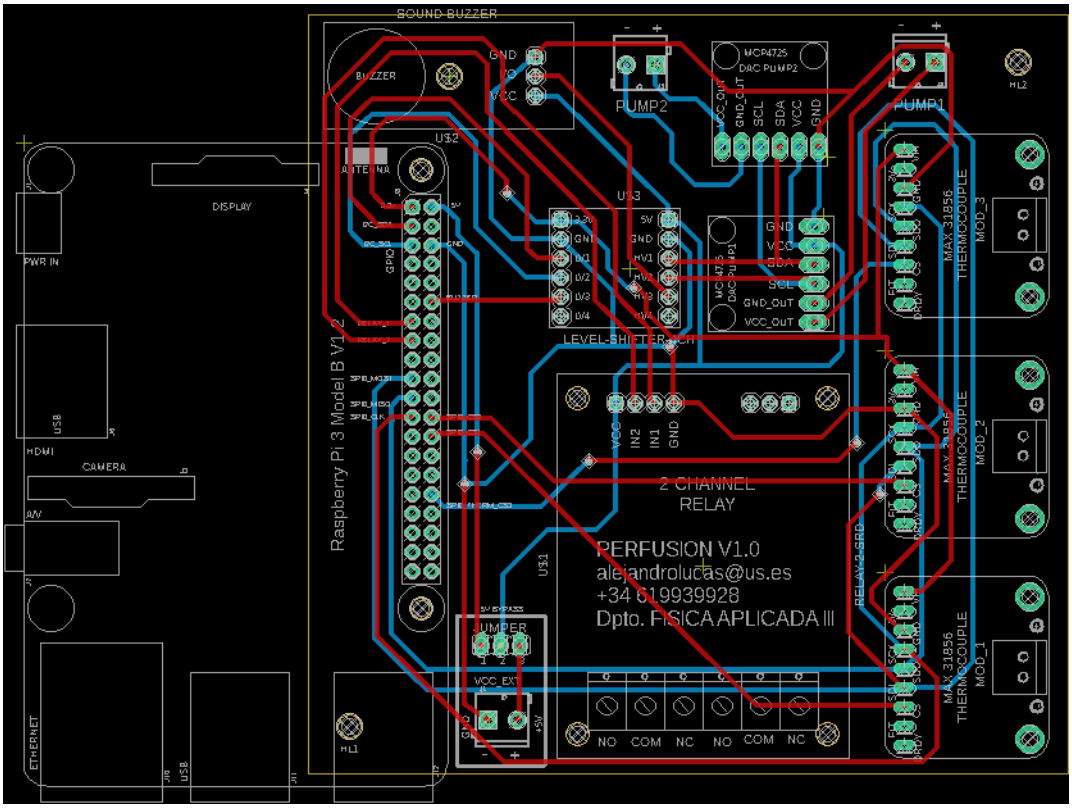
ANEXO B: PLANOS DE LA PCB

En este anexo se encuentran los planos referentes al diseño de la PCB (esquemático,). Todo el diseño fue realizado con el programa Eagle por lo que se adjunta junto a esta memoria los archivos correspondientes al diseño.

Esquemático

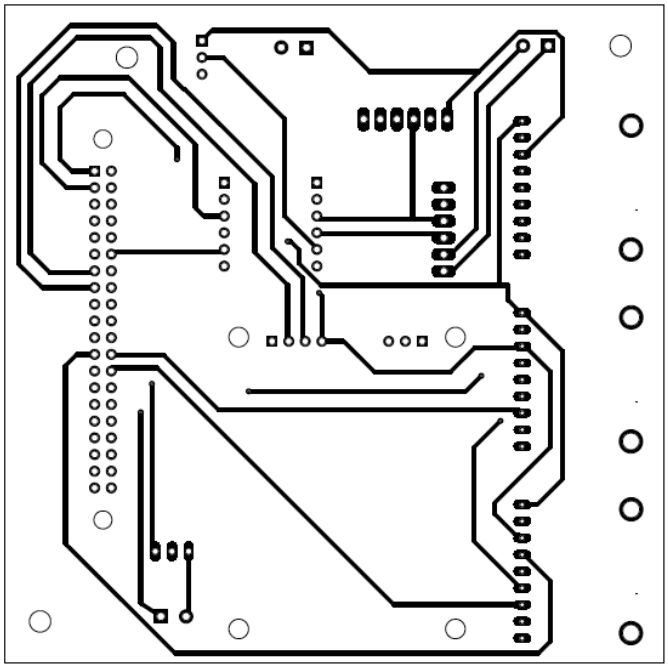


Ruteado

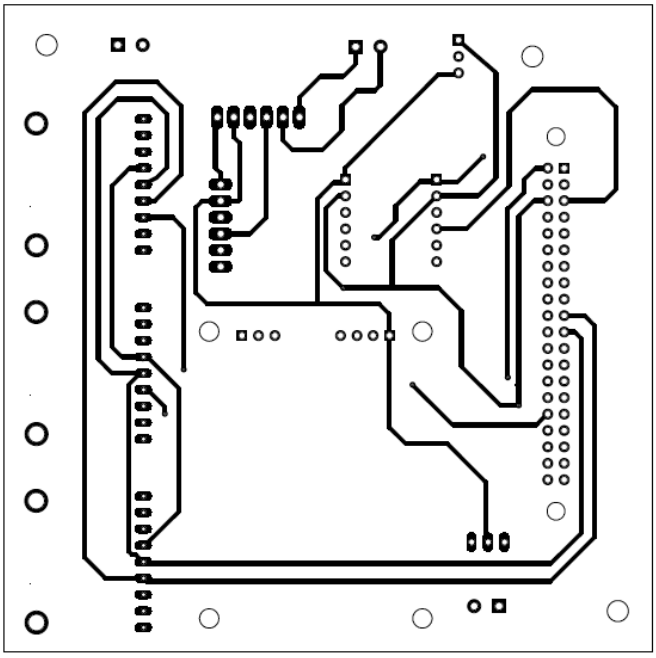


Circuitos impresos.

Cara top

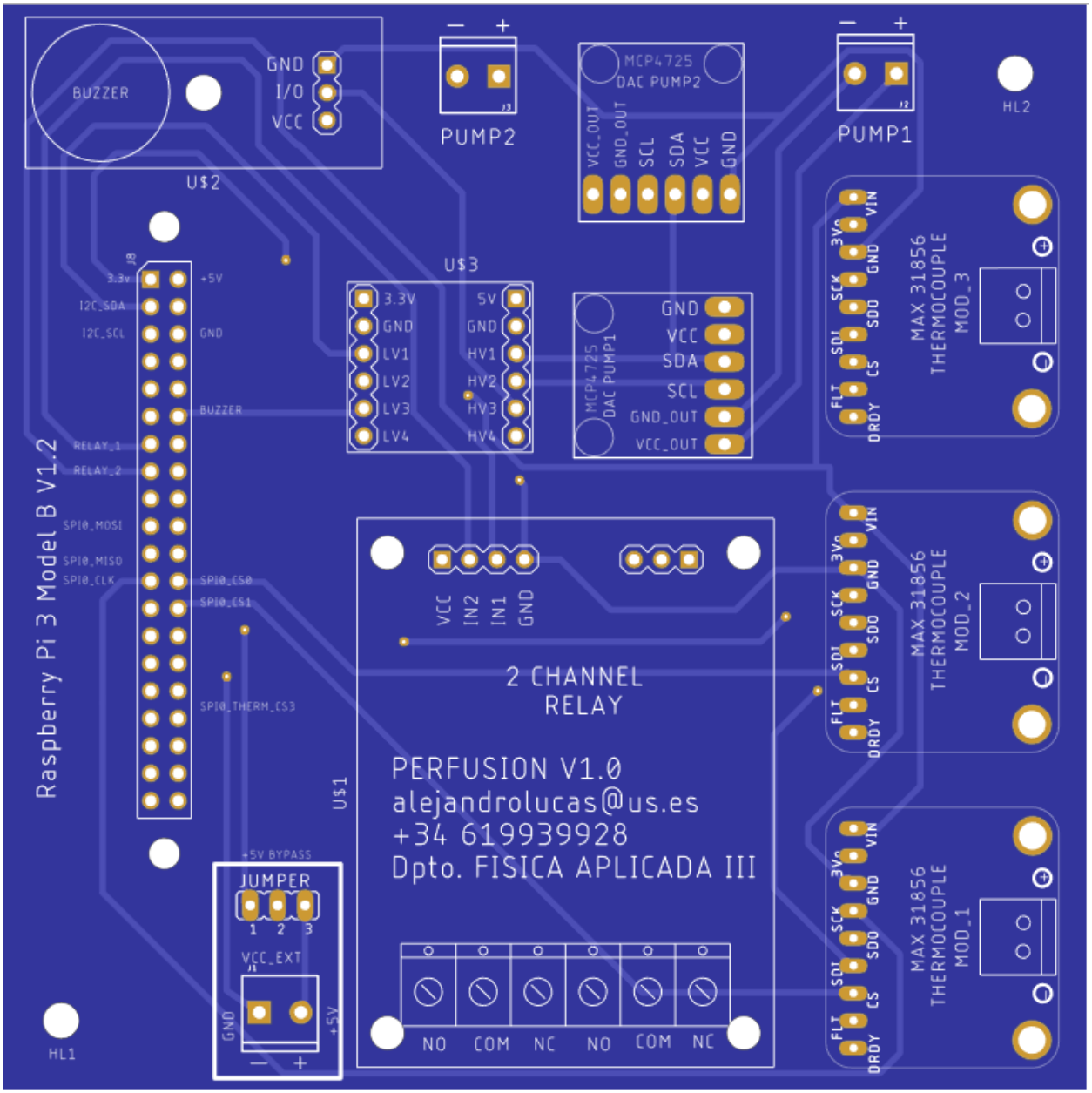


Cara bottom

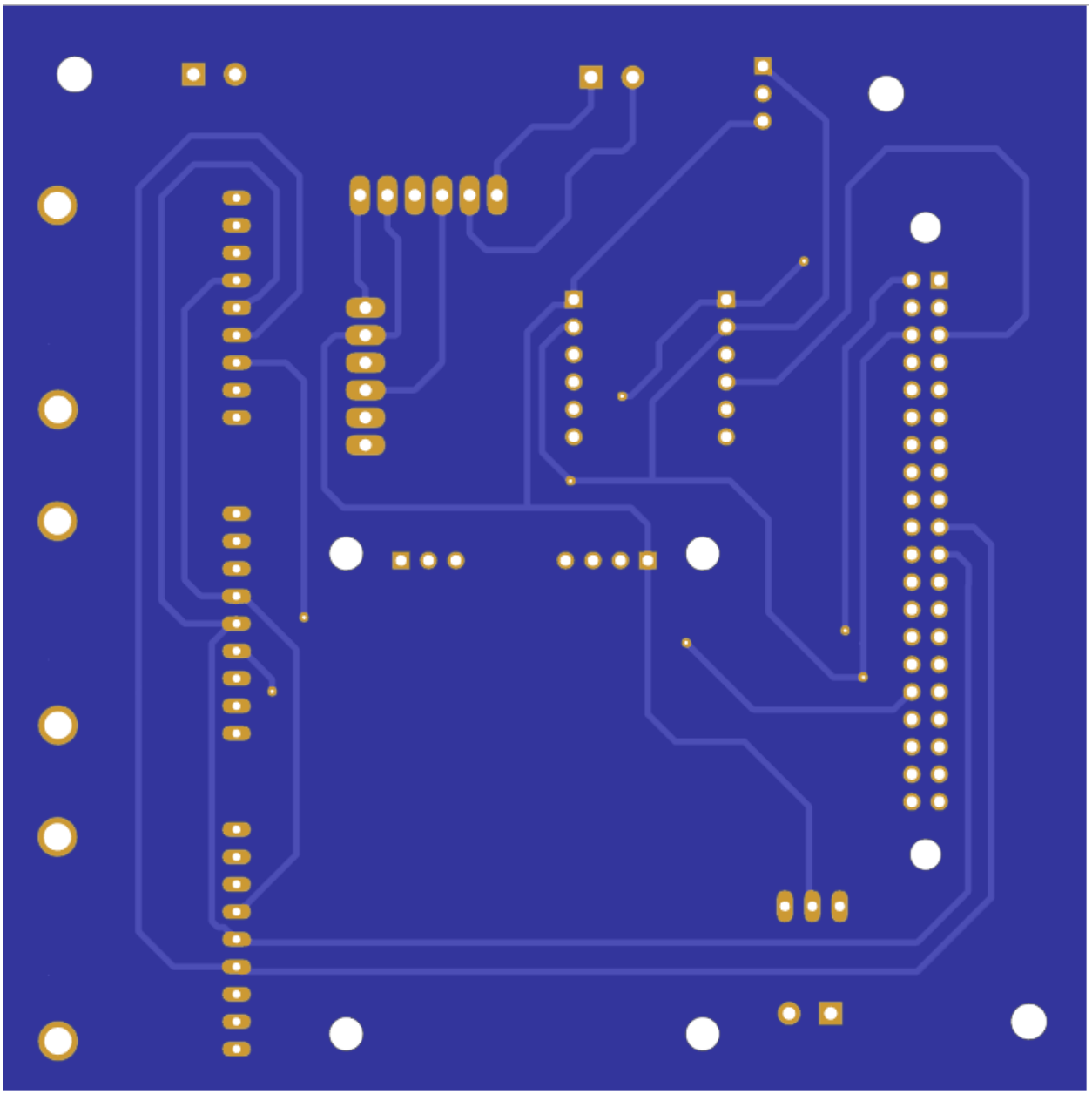


PCB

Cara top


BUZZER
GND
I/O
VCC
U$2
PUMP2
MCP4725
DAC PUMP2
VCC_OUT
GND_OUT
SCL
SDA
VCC
GND
PUMP1
HL2
U$3
3.3V
GND
LV1
LV2
LV3
LV4
5V
GND
HV1
HV2
HV3
HV4
MCP4725
DAC PUMP1
GND
VCC
SDA
SCL
GND_OUT
VCC_OUT
MAX 31856
THERMOCOUPLE
MOD_3
MAX 31856
THERMOCOUPLE
MOD_2
MAX 31856
THERMOCOUPLE
MOD_1
Raspberry Pi 3 Model B V1.2
J8
3.3v
+5V
I2C_SDA
I2C_SCL
GND
BUZZER
RELAY_1
RELAY_2
SPI0_MOSI
SPI0_MISO
SPI0_CLK
SPI0_CS0
SPI0_CS1
SPI0_THERM_CS3
U$1
VCC
IN2
IN1
GND
2 CHANNEL
RELAY
PERFUSION V1.0
alejandrolucas@us.es
+34 619939928
Dpto. FISICA APLICADA III
NO
COM
NC
NO
COM
NC
+5V BYPASS
JUMPER
1 2 3
VCC_EXT
J1
GND
+5V
HL1

Cara bottom



ANEXO C: DATASHEET

En este subapartado se puede encontrar todos los enlaces webs de los datasheet de cada uno de los dispositivos físicos utilizados en este proyecto en el caso de que se quieran consultar.

Raspberry Pi 3 B+

https://www.raspberrypi.org/documentation/hardware/computemodule/datasheets/rpi_DATA_CM3plus_1p0.pdf

MCP4725

http://ww1.microchip.com/downloads/en/DeviceDoc/22039c.pdf

Módulo rele de 2 canales

http://www.handsontec.com/dataspecs/2Ch-relay.pdf

Buzzer

Nota: Se deja el enlace de la compra donde viene información sobre el módulo ya que no se ha encontrado datasheet del mismo.

https://www.amazon.es/ARCELI-3-3-5V-M%C3%B3dulo-zumbador-Arduino/dp/B07MPYWVGD/ref=sr_1_15?s=electronics&ie=UTF8&qid=1553545289&sr=1-15&keywords=buzzer+arduino

Termopar

https://cdn-learn.adafruit.com/assets/assets/000/035/948/original/MAX31856.pdf

Módulo bidireccional

https://arduino-shop.eu/arduino-platform/1481-iic-i2c-5v-to-33-v-bidirectional-converter-logic-level.html

http://dlnmh9ip6v2uc.cloudfront.net/datasheets/BreakoutBoards/Logic_Level_Bidirectional.pdf

http://dlnmh9ip6v2uc.cloudfront.net/datasheets/BreakoutBoards/BSS138.pdf

REFERENCIAS

[1] Autor, «Este es el ejemplo de una cita,» *Tesis Doctoral,* vol. 2, nº 13, 2012.

[2] O. Autor, «Otra cita distinta,» *revista,* p. 12, 2001.

ÍNDICE DE CONCEPTOS

GLOSARIO

ISO: International Organization for Standardization 4

UNE: Una Norma Española 4