Control de Encendido y Apagado

Proyecto: Control de Encendido, Apagado y Monitorización para la Raspberry Pi 3

0. Introducción

¿Quién no ha tenido problemas con el encendido y apagado de una Raspberry Pi? Ese pequeño dispositivo que tanto amamos, pero que, seamos sinceros, no lo pone nada fácil cuando de apagarla de forma segura o encenderla sin cables extras se trata. La Raspberry Pi 3, aunque increíblemente versátil, carece de un botón de encendido/apagado sencillo como el de un ordenador.

Apagar la Raspberry Pi correctamente requiere tiempo y cuidado. Puedes conectarte al sistema para ejecutar un comando de apagado seguro, o te arriesgas a desconectar el cable de alimentación y esperar que no se corrompa la tarjeta SD. ¿Y para encenderla? Conectar y desconectar el cable funciona, pero está lejos de ser una solución práctica, especialmente si la Raspberry Pi está montada en un lugar de difícil acceso.

Además, en aplicaciones comunes como servidores multimedia, sistemas domóticos o servidores de archivos, la Raspberry Pi suele estar encendida 24/7 para garantizar la disponibilidad. Sin embargo, mantenerla activa todo el tiempo puede ser innecesario si solo se utiliza unas pocas horas al día. Esto no solo implica un consumo energético continuo (entre 2.5 y 4 kWh al mes, dependiendo de la carga), sino que también representa un desperdicio para aquellos proyectos que no requieren disponibilidad constante.

¿Por qué no tomar el control y encenderla solo cuando sea necesario? Y ya que estamos, ¿por qué no incorporar un sistema que nos permita también monitorizar su estado y tomar decisiones directamente desde un menú interactivo?

¿Qué queremos hacer y por qué?

Nuestro objetivo es crear un sistema integral que resuelva estas limitaciones, mejorando la comodidad, el control y la eficiencia energética al interactuar con la Raspberry Pi. Planteamos un diseño que permita:

1. **Control de encendido y apagado dual:**
    - **Botón físico:** Encender la Raspberry Pi de manera inmediata y apagarla de forma segura manteniendo presionado el botón durante al menos 2 segundos. Las pulsaciones breves mientras está encendida no tendrán efecto, evitando errores accidentales.
    - **Control remoto:** Implementar un sistema basado en un ESP8266 que permita encender y apagar la Raspberry Pi desde cualquier dispositivo con conexión Wi-Fi, tanto dentro de la red local como de forma remota desde cualquier lugar del mundo mediante HTTP.
2. **Monitorización con un menú interactivo:**

    - Incorporar un **LCD de 16x2 caracteres** que muestre información clave sobre la Raspberry Pi, como su temperatura, estado de conexión o mensajes de notificación.
    - Desarrollar un menú interactivo controlado mediante un **encoder rotatorio con pulsador**, que permita navegar entre opciones, consultar parámetros y ejecutar acciones, como apagar la Raspberry Pi desde el propio menú.
3. **Notificaciones sonoras:**
    - Integrar un **buzzer** para emitir alertas sonoras que indiquen estados importantes, como encendido, apagado, o alarmas configurables según las necesidades del sistema.

Nuestro enfoque

Solucionaremos estas dificultades con un diseño modular y eficiente:

1. **Control remoto con ESP8266:** Un pequeño servidor web configurado en el ESP8266 actuará como interfaz para enviar órdenes remotas de encendido/apagado a la Raspberry Pi en función del estado actual.
2. **LCD y menú interactivo:**
    - Implementaremos un sistema de navegación intuitivo con el encoder rotatorio, permitiendo desplazarse entre las distintas opciones del menú en el LCD.
    - El menú mostrará información relevante del sistema y permitirá ejecutar acciones directamente desde la pantalla.
3. **Buzzer programable:** Utilizaremos sonidos específicos para identificar eventos clave, como encendido, apagado o posibles alertas configurables.

¿Qué necesitamos?

- Una **Raspberry Pi 3**, que será el corazón del sistema.
- Un **ESP8266 (NodeMCU)**, que actuará como cerebro del control remoto.
- Un **LCD de 16x2** para la interfaz de usuario.
- Un **encoder rotativo** Keyes KY-040 para el desplazamiento por el menú.
- Un **buzzer**, para notificaciones sonoras.
- **Código** para gestionar tanto el ESP8266 como la lógica del menú y control en la Raspberry Pi.

Propósito y futuro

Con este sistema no solo lograremos un control total sobre el encendido y apagado de la Raspberry Pi, sino que también añadiremos funciones de monitorización que harán más práctico y eficiente su uso. Este proyecto tiene el potencial de evolucionar: podría integrar

sensores adicionales, automatizar tareas según condiciones específicas o ser la base para sistemas de domótica avanzados.

Estamos convencidos de que este enfoque hará que trabajar con la Raspberry Pi sea mucho más cómodo, eficiente y completo, permitiendo controlarla fácilmente, ya sea mediante un navegador o un menú interactivo… ¡todo con un toque o un clic desde cualquier lugar del mundo!

---

1. Encendido/Apagado Raspberry Pi 3 con botón físico.

Primero de todo configuraremos la raspberry para apagar y encender el sistema mediante un botón físico usando el **GPIO 3 (SDA1)**

1. Usamos el GPIO 3 (SDA1) para Encender la Raspberry Pi

- Conectamos un interruptor momentáneo entre el pin GPIO 3 y un pin GND:
- Si la Raspberry Pi está apagada pero con energía (alimentada por su fuente), al presionar el interruptor, el GPIO 3 activará el sistema y la Raspberry Pi se encenderá

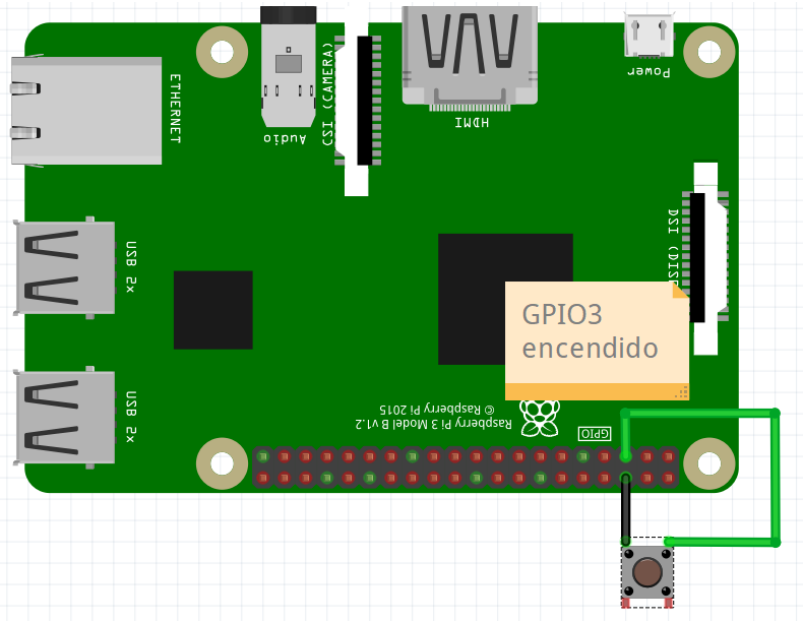
Ésta és **una función predeterminada del hardware**: El **GPIO 3** (también conocido como SCL en I2C) tiene una funcionalidad integrada por diseño para encender la Raspberry Pi cuando se conecta a **GND**. **No depende de `config.txt`**: Esta función no está gestionada por un overlay (`dtoverlay`) en el firmware. Está directamente implementada en el hardware y no se puede desactivar.

2. Apagado seguro básico con GPIO 3

Podemos establecer el mismo pin simplemente por configuración con config.txt como pin de apagado

- Abrimos el archivo para editar:
  `sudo nano /boot/firmware/config.txt`
- Agregamos esta linea:
  `dtoverlay=gpio-shutdown`
  - Esto permitirá que el GPIO 3 funcione como botón de apagado seguro cuando el sistema está encendido.
- Guardamos los cambios y reiniciamos:
  `sudo reboot`
- Si la Raspberry Pi está encendida, presionamos el interruptor para apagarla de forma segura (configurado con `gpio-shutdown`).

- Montaje:



Podríamos configurar otro pin de apagado también mediante config.txt.

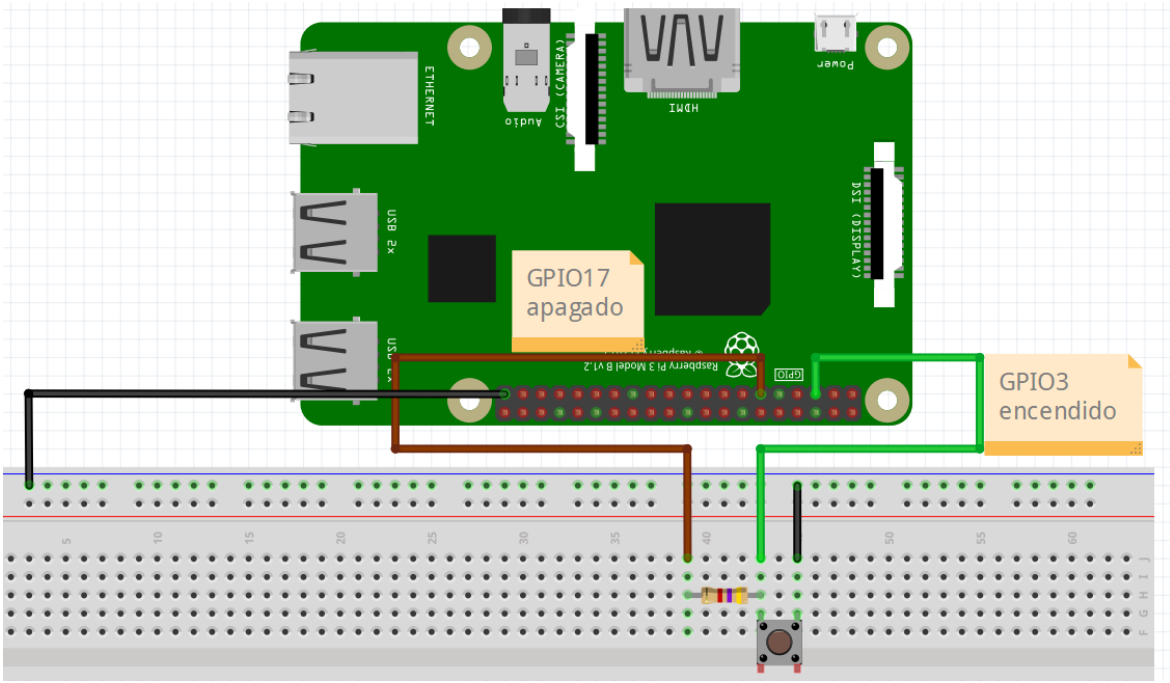
- Editamos `/boot/firmware/config.txt` para habilitar apagado seguro en el pin XX

  ```
  dtoverlay=gpio-shutdown,gpio_pin=XX
  ```

  - Reemplazamos `XX` con el número del pin GPIO que deseas usar.
  - Usaremos el GPIO 17 (pin físico 11): `dtoverlay=gpio-shutdown,gpio_pin=17`
  - Para requerir un tiempo de pulsación específico (en milisegundos): `dtoverlay=gpio-shutdown,gpio_pin=17,debounce=1500`

    Esto agrega un retardo de 1,5s para evitar activaciones accidentales.

- Guardamos los cambios y reiniciamos: `sudo reboot`
- `Montaje:`



En el README de overlays podemos ver los parámetros configurables de **gpio-shutdown**

```
admin@raspberrypi:/boot/firmware $ cat /boot/firmware/overlays/README | grep gpio-shutdown -A 10
Name:   gpio-shutdown
Info:   Initiates a shutdown when GPIO pin changes. The given GPIO pin
        is configured as an input key that generates KEY_POWER events.

        This event is handled by systemd-logind by initiating a
        shutdown. Systemd versions older than 225 need an udev rule
        enable listening to the input device:

                ACTION!="REMOVE", SUBSYSTEM=="input", KERNEL=="event*", \
                        SUBSYSTEMS=="platform", DRIVERS=="gpio-keys", \
                        ATTRS{keys}=="116", TAG+="power-switch"
--
Load:   dtoverlay=gpio-shutdown,<param>=<val>
Params: gpio_pin                GPIO pin to trigger on (default 3)
                                For Raspberry Pi 1 Model B rev 1 set this
                                explicitly to value 1, e.g.:

                                    dtoverlay=gpio-shutdown,gpio_pin=1

        active_low              When this is 1 (active low), a falling
                                edge generates a key down event and a
                                rising edge generates a key up event.
                                When this is 0 (active high), this is
                                reversed. The default is 1 (active low).

        gpio_pull               Desired pull-up/down state (off, down, up)
                                Default is "up".
```

Consideraciones:

1. GPIO 3 es un pin I2C:

El **GPIO3** es un pin fundamental si necesitamos usar dispositivos que requieran comunicación I2C, ya que funciona como **SDA1**. Este protocolo se utiliza para conectar

dispositivos como sensores, pantallas, etc., con solo dos pines, lo que lo convierte en una solución eficiente en cuanto al ahorro de pines.

Ell **GPIO3** ya tiene una funcionalidad predeterminada por hardware para el **encendido** de la Raspberry Pi y no podemos modificar este comportamiento, sin embargo, surge la necesidad de decidir cómo gestionar el **apagado** sin comprometer la funcionalidad I2C.

Inicialmente, consideramos usar otro pin, como el **GPIO17**, exclusivamente para el apagado. Esto dejaba el GPIO3 libre para el bus I2C mientras la Raspberry estaba encendida y utilizabamos dos pines (GPIO3 y GPIO17) para el encendido y apagado pero existe una forma de **mover el bus I2C a otros pines GPIO**.

Esto nos permite usar el GPIO3 tanto para el **encendido** como para el **apagado**, utilizando un pin en vez de dos para este cometido, y a la vez manteniendo la funcionalidad I2C mediante pines alternativos. Es la opción más eficiente y ahorrativa en cuanto a pines.

Pasos para cambiar el bus I2C a otros pines:

1. Abrimos el archivo config.txt:
   `sudo nano /boot/config.txt`
2. Agregamos un nuevo bus I²C en pines alternativos (por ejemplo, **GPIO23** y **GPIO24**):
   `dtoverlay=i2c-gpio,bus=3,i2c_gpio_sda=23,i2c_gpio_scl=24`
3. Guardamos los cambios y reiniciamos la Raspberry Pi:
   `sudo reboot`
4. Ahora, el nuevo bus I2C estará disponible en /dev/i2c-3. Esto libera el GPIO3 para gestionar tanto el encendido como el apagado.

Conclusión:

Con esta configuración, podemos aprovechar al máximo los pines de la Raspberry Pi, manteniendo la funcionalidad I2C y utilizando el GPIO3 tanto para encendido como apagado. Esta solución es funcional y optimiza los recursos de hardware disponibles.

2. Control del apagado por configuracion en config.txt:

Después de hacer varias pruebas de apagado y con diferentes parámetros no conseguimos un apagado estable con menos de dos pulsaciones y tampoco nos gusta cómo gestiona el release del botón ni con el pin **GPIO3** ni con el **GPIO17** (espera a que lo soltemos para contabilizar el tiempo de debounce en vez de apagar en cuanto alcance ese tiempo aunque permanezca pulsado)

Enlace -> video - evidencias funcionamento

Solución:

Decidimos gestionar el apagado mediante un script de python para tener un mayor control y flexibilidad sobre el comportamiento del botón en el **GPIO 3** y tenerlo como servicio ejecutándose desde el inicio del sistema.

---

2. Apagado de la Raspberry mediante script ejecutado como servicio de inicio.

- Lo planteamos con interrupciones de escucha de eventos por el GPIO3
- Implementamos que con una pulsación de menos de dos segundos no apague el sistema, pero que cuando la pulsación alcance ese tiempo hará el shutdown independientemente si el botón se ha soltado o no
- Antes de apagar el sistema limpiará todos los pines del GPIO por si hay otros procesos utilizándolos evitar que queden pines encendidos

Enlace -> script - shutdown_button.py

Establecemos el script como un servicio de systemd **de inicio de sistema**

1. Creamos un archivo de servicio

1. Lo creamos en `/etc/systemd/system/`
   `sudo nano /etc/systemd/system/shutdown-button.service`
2. Agregamos el siguiente contenido al archivo:

```
  GNU nano 7.2
[Unit]
Description=Botón de apagado en GPIO
After=multi-user.target

[Service]
Type=simple
ExecStart=/usr/bin/python3 /etc/init/shutdown_button.py
Restart=always
User=root

[Install]
WantedBy=multi-user.target
```

3. Nos aseguramos de ubicar `shutdown_button.py` en /etc/init y le damos permisos de ejecución:

```
admin@raspberrypi:~/Documents $ ls
llibreria_dispositius  prova_buzzer.py  prova_LCD.py  requirements.txt  shutd
admin@raspberrypi:~/Documents $ sudo cp shutdown_button.py /etc/init
admin@raspberrypi:~/Documents $ cd /etc/init
admin@raspberrypi:/etc/init $ ls -l
total 8
-rw-r--r-- 1 root root  202 Jan  1  2015 paxctld.conf
-rw-r--r-- 1 root root 1296 Nov 25 22:03 shutdown_button.py
admin@raspberrypi:/etc/init $ sudo chmod +x shutdown_button.py
admin@raspberrypi:/etc/init $ ls -l
total 8
-rw-r--r-- 1 root root  202 Jan  1  2015 paxctld.conf
-rwxr-xr-x 1 root root 1296 Nov 25 22:03 shutdown_button.py
```

2. Habilitamos y arrancamos el servicio

1. Recargamos los servicios para que `systemd` detecte el nuevo archivo:
   `sudo systemctl daemon-reload`
2. Activamos el servicio para que se ejecute automáticamente al inicio:
   `sudo systemctl enable shutdown-button.service`
3. Inicia el servicio manualmente para probarlo:
   `sudo systemctl start shutdown-button.service`
4. Verificamos el estado del servicio para asegurarte de que funciona correctamente:
   `sudo systemctl status shutdown-button.service`

```
admin@raspberrypi: /etc/init
admin@raspberrypi:/etc/init $ sudo systemctl daemon-reload
admin@raspberrypi:/etc/init $ sudo systemctl enable shutdown-button.service
admin@raspberrypi:/etc/init $ sudo systemctl start shutdown-button.service
admin@raspberrypi:/etc/init $ sudo systemctl status shutdown-button.service
● shutdown-button.service - Botón de apagado en GPIO
     Loaded: loaded (/etc/systemd/system/shutdown-button.service; enabled;
     Active: active (running) since Mon 2024-11-25 22:29:18 CET; 8s ago
   Main PID: 2352 (python3)
      Tasks: 4 (limit: 1557)
        CPU: 562ms
     CGroup: /system.slice/shutdown-button.service
             └─2352 /usr/bin/python3 /etc/init/shutdown_button.py

Nov 25 22:29:18 raspberrypi systemd[1]: Started shutdown-button.service - B
```

Observamos que el servicio está activo y sin errores. El script se ejecutará automáticamente cada vez que arranquemos la Raspberry Pi..

Enlace -> video - evidencias funcionamento

En este momento disponemos la siguiente funcionalidad de encendido/apagado mediante GPIO3:

- Si la raspberry está **encendida**:
  - Pulsación corta: no hace nada
  - Pulsación larga (>2s): apaga de forma segura

- Si la raspberry esta **apagada**:
    - Pulsación corta: enciende la raspberry

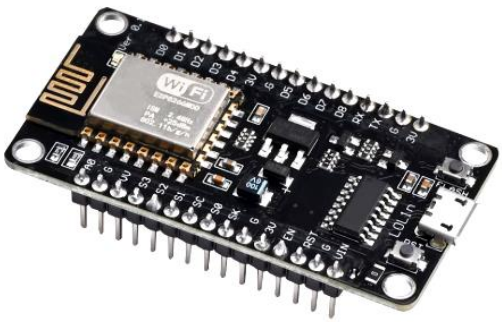
---

3. Encendido/apagado de la Raspberry mediante el ESP8266

0. Introducción

Los **ESP8266** y **ESP32** son microcontroladores potentes y económicos que te permiten crear proyectos IoT (Internet de las cosas) variados.

Utilizaremos un **ESP8266 NodeMCU** ampliamente utilizado en proyectos de IoT debido a su bajo costo y capacidades Wi-Fi integradas para controlar remotamente la conexión del GPIO 3 al GND. Estos dispositivos pueden conectarse a Internet y recibir comandos para activar o desactivar pines GPIO.



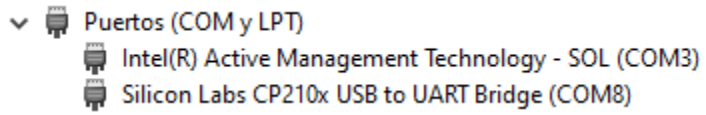
1. Pasos para Configurar el Dispositivo ESP8266 en MicroPython

Preparamos el entorno.

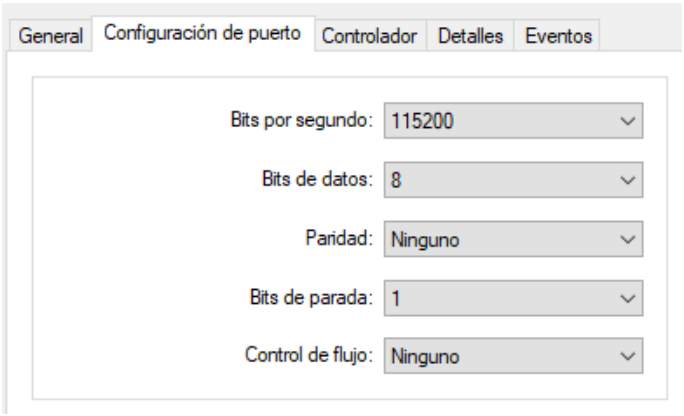
- Descargamos una herramienta para flashear el firmware, como esptool. Con Python, la también la podemos instalar con:
  `pip install esptool`
- Descargamos el firmware de MicroPython para el ESP8266 desde la página oficial
- Instalamos el driver para AZ-Delivery NodeMCU Amica V2 ESP8266MOD
  Según las especificaciones utiliza un chip CP2102 como interfaz USB-UART por lo que necesitamos instalar el controlador de Silicon Labs para que el ordenador pueda reconocer el ESP.
  Descargamos el driver desde la página oficial de Silicon Labs y seleccionamos el correspondiente.

Conectamos el ESP8266 al ordenador

- Conectamos el ESP8266 a tu ordenador mediante un cable micro-USB que permita la transmisión de datos.
- Verificamos que el ordenador detecte el puerto serie:
    - Usamos el Administrador de dispositivos de Windows para identificar el puerto COM



    - podemos ver que utiliza el COM8
    - Es muy importante establecer la velocidad de transmisión de bits en 115200



      El baud rate predeterminado de MicroPython en el ESP8266 es **115200**. Esta es la velocidad estándar para la mayoría de dispositivos de comunicación serie y está soportada por casi todas las herramientas y configuraciones de terminal.

Borramos la memoria flash

- Al borrar la memoria, nos aseguramos de que no haya configuraciones ni datos antiguos que puedan causar problemas o comportamientos inesperados. Si el ESP8266 está nuevo o nunca se ha usado no es estrictamente necesario, aun así es una buena práctica el hacerlo para impedir que queden fragmentos o configuraciones del firmware anterior (como contraseñas Wi-Fi, variables del programa o datos guardados)

  ```
  python -m esptool --port COM8 erase_flash
  ```
  Reemplazamos `<puerto>` con el COM3 identificado en el paso anterior y lo ejecutamos dentro de python.

```
C:\Users\digit>python -m esptool --port COM3 erase_flash
esptool.py v4.8.1
Serial port COM3
Connecting....
Detecting chip type... Unsupported detection protocol, switching and trying again...
Connecting....
Detecting chip type... ESP8266
Chip is ESP8266EX
Features: WiFi
Crystal is 26MHz
MAC: c8:c9:a3:1a:da:5b
Uploading stub...
Running stub...
Stub running...
Erasing flash (this may take a while)...
Chip erase completed successfully in 1.2s
Hard resetting via RTS pin...
```

Flasheamos el firmware de MicroPython

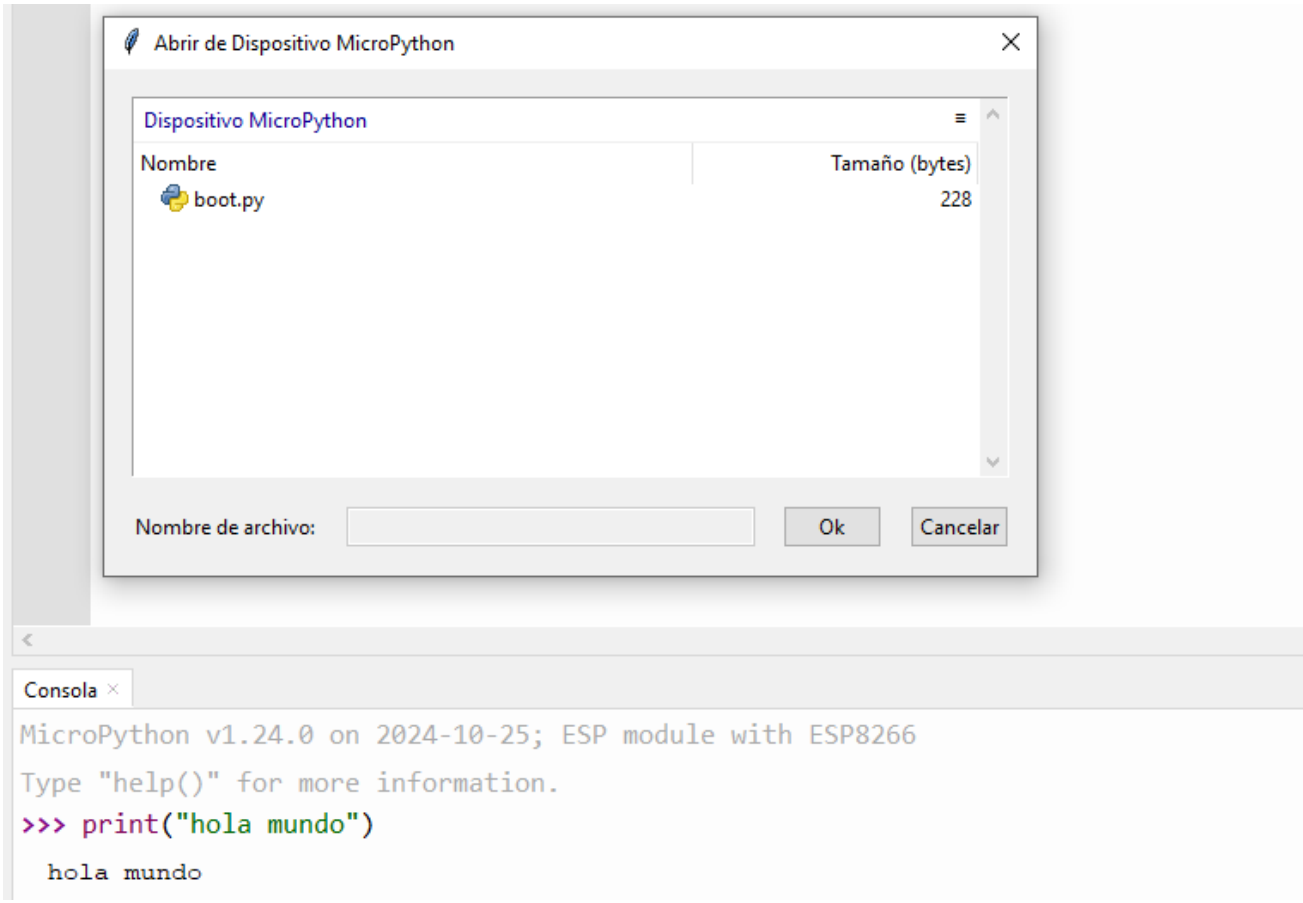
- Utilizamos esptool bajo python con el siguiente comando:

```
python -m esptool --port COM8 --baud 115200 write_flash --
flash_size=detect 0 <ruta_del_firmware>.bin
```

```
C:\Users\Alumne_mati1>python -m esptool --port COM8 --baud 115200 write_flash --flash_
"C:\Users\Alumne_mati1\Desktop\software\ESP\ESP8266_GENERIC-20241025-v1.24.0.bin"
esptool.py v4.8.1
Serial port COM8
Connecting....
Detecting chip type... Unsupported detection protocol, switching and trying again...
Connecting....
Detecting chip type... ESP8266
Chip is ESP8266EX
Features: WiFi
Crystal is 26MHz
MAC: c8:c9:a3:1a:da:5b
Uploading stub...
Running stub...
Stub running...
Configuring flash size...
Auto-detected Flash size: 4MB
Flash will be erased from 0x00000000 to 0x0009bfff...
Flash params set to 0x0040
Compressed 635916 bytes to 425558...
Wrote 635916 bytes (425558 compressed) at 0x00000000 in 37.9 seconds (effective 134.2
Hash of data verified.
```

Nos conectamos a ESP8266 mediante Thonny para gestionar por COM8 el intérprete de micropython y scripts.

- Descargamos e instalamos Thonny desde su página oficial: https://thonny.org/.
- Probamos de escribir un print en el intérprete y abrir los archivos donde vemos que tiene un boot.py



Ahora lo tenemos listo para trabajar con Micropython y crear scripts.

2. Implementación de un servidor HTTP de encendido/apagado en el ESP y control del estado de la Raspberry Pi

Propósito:

Queremos implementar un sistema que nos permita interactuar con la Raspberry Pi desde un ESP configurado como **servidor HTTP**. El ESP debe ejecutar un script que se inicie automáticamente al encenderse y que permita, al conectarnos vía IP con el protocolo HTTP desde la red local, evaluar si la Raspberry está encendida o apagada. En función del estado detectado, queremos que el ESP nos dé la opción de apagarla o encenderla.

Además, buscamos que el estado de la Raspberry Pi sea gestionado de manera confiable y eficiente, reflejando su estado real mediante el GPIO4.

Cómo funcionará el código

1. **Lectura del estado de la Raspberry Pi:**
    - El ESP utiliza el pin GPIO4 (D2) para leer el estado de la Raspberry Pi:

        - `HIGH` (`1`): La Raspberry Pi está encendida.
        - `LOW` (`0`): La Raspberry Pi está apagada.
2. **Página principal (`/`):**
    - Si la Raspberry Pi está encendida, muestra un mensaje indicando el estado y un botón para apagarla.
    - Si la Raspberry Pi está apagada, muestra un mensaje indicando el estado y un botón para encenderla.
3. **Acciones (`/encender` y `/apagar`):**
    - Enviar un pulso al GPIO5 (D1) para encender o apagar la Raspberry Pi:
        - Pulso de 0,5 segundos para encender.
        - Pulso de 2,5 segundos para apagar.
4. **Conexión Wi-Fi:**
    - El ESP se conectará a la red local por wifi para que el servidor HTTP esté disponible.

Ventajas de este enfoque

- Detecta el estado real de la Raspberry Pi usando el pin GPIO4.
- Proporciona una interfaz web simple para interactuar con la Raspberry Pi.
- Ofrece botones dinámicos basados en el estado actual de la Raspberry Pi.

3. Implementación por software para controlar el GPIO4 de la Raspberry Pi

Para lograr que el estado de la Raspberry Pi se refleje correctamente en el GPIO4, implementaremos una solución por software que maneja el estado de este pin en los eventos de encendido y apagado.

El servicio para manejar el estado del GPIO pretenderá:

- **Al iniciar la Raspberry Pi:**
    - Configuraremos el GPIO4 como salida y lo estableceremos en alto (`HIGH`) para indicar que la Raspberry Pi está encendida.
- **Al apagar la Raspberry Pi:**
    - Antes de completar el apagado, estableceremos el GPIO4 en bajo (`LOW`) para indicar que la Raspberry Pi está apagada.

Script de servicio de inicio de la RB3 para manejar el GPIO4

Creamos un script en Python llamado `gpio_status.py`, que se encargará de establecer el estado del GPIO4 en función del evento recibido.

[Enlace -> script - gpio_status.py]

Configurarmos un servicio de inicio y apagado en `systemd`

Para automatizar la ejecución del script, configuramos un servicio en `systemd` que maneje el estado del GPIO4 en los eventos de inicio y apagado de la Raspberry Pi.

1. **Archivo del servicio:** Creamos el archivo `/etc/systemd/system/gpio_status.service` con el siguiente contenido:

```
  GNU nano 7.2                                          gpio_status.service
[Unit]
Description=Gestiona el estado del GPIO4 para indicar el estado de la Raspberry Pi
After=multi-user.target

[Service]
Type=oneshot
ExecStart=/usr/bin/python3 /etc/init/gpio_status.py start
ExecStop=/usr/bin/python3 /etc/init/gpio_status.py stop
RemainAfterExit=true

[Install]
WantedBy=multi-user.target
```

   - **`ExecStart`:** Se ejecuta al iniciar la Raspberry Pi y establece GPIO4 en alto (`HIGH`).
   - **`ExecStop`:** Se ejecuta al apagar la Raspberry Pi y establece GPIO4 en bajo (`LOW`).
2. **Ubicamos `gpio_status.py`** en `/etc/init` y le damos permisos de ejecución
3. **Habilitar el servicio:** Ejecutamos los siguientes comandos para habilitar y activar el servicio:

```
admin@raspberrypi:/etc/systemd/system $ sudo systemctl daemon-reload
admin@raspberrypi:/etc/systemd/system $ sudo systemctl enable gpio_status.service
admin@raspberrypi:/etc/systemd/system $ sudo systemctl start gpio_status.service
admin@raspberrypi:/etc/systemd/system $ sudo systemctl status gpio_status.service
```

   Podemos comprobar el estado del servicio;

```
admin@raspberrypi:/etc/systemd/system $ sudo systemctl status gpio_status.service
● gpio_status.service - Gestiona el estado del GPIO4 para indicar el estado de la Raspberry Pi
     Loaded: loaded (/etc/systemd/system/gpio_status.service; enabled; preset: enabled)
     Active: active (exited) since Thu 2024-11-28 10:11:07 CET; 8min ago
    Process: 2196 ExecStart=/usr/bin/python3 /etc/init/gpio_status.py start (code=exited, status
   Main PID: 2196 (code=exited, status=0/SUCCESS)
        CPU: 236ms

Nov 28 10:11:07 raspberrypi systemd[1]: Starting gpio_status.service - Gestiona el estado del GP
Nov 28 10:11:07 raspberrypi python3[2196]: Raspberry Pi: GPIO4 establecido en HIGH (ENCENDIDA)
Nov 28 10:11:07 raspberrypi systemd[1]: Finished gpio_status.service - Gestiona el estado del GP
```

---

4. Evaluación del estado (encendido/apagado) de la RB3 desde el ESP

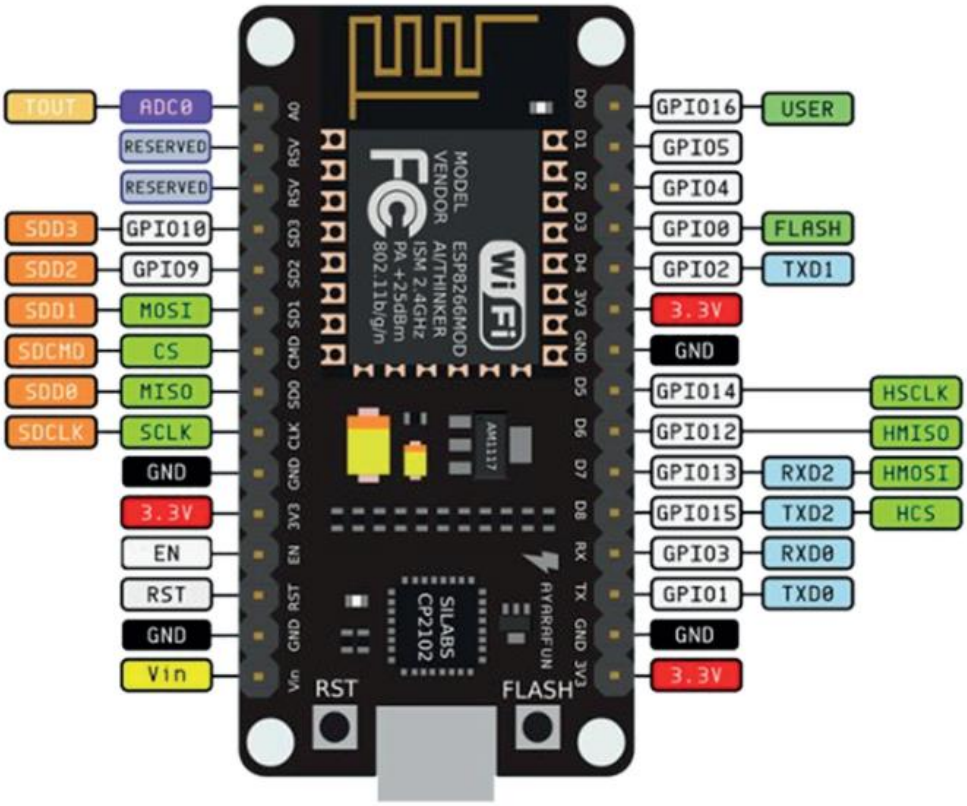
El ESP monitorizará el estado del GPIO4 de la Raspberry Pi leyendo el nivel de voltaje. Si el GPIO4 está en alto (`HIGH`), consideraremos que la Raspberry Pi está encendida. Si está en bajo (`LOW`), asumiremos que está apagada.

Proceso general de la interacción:

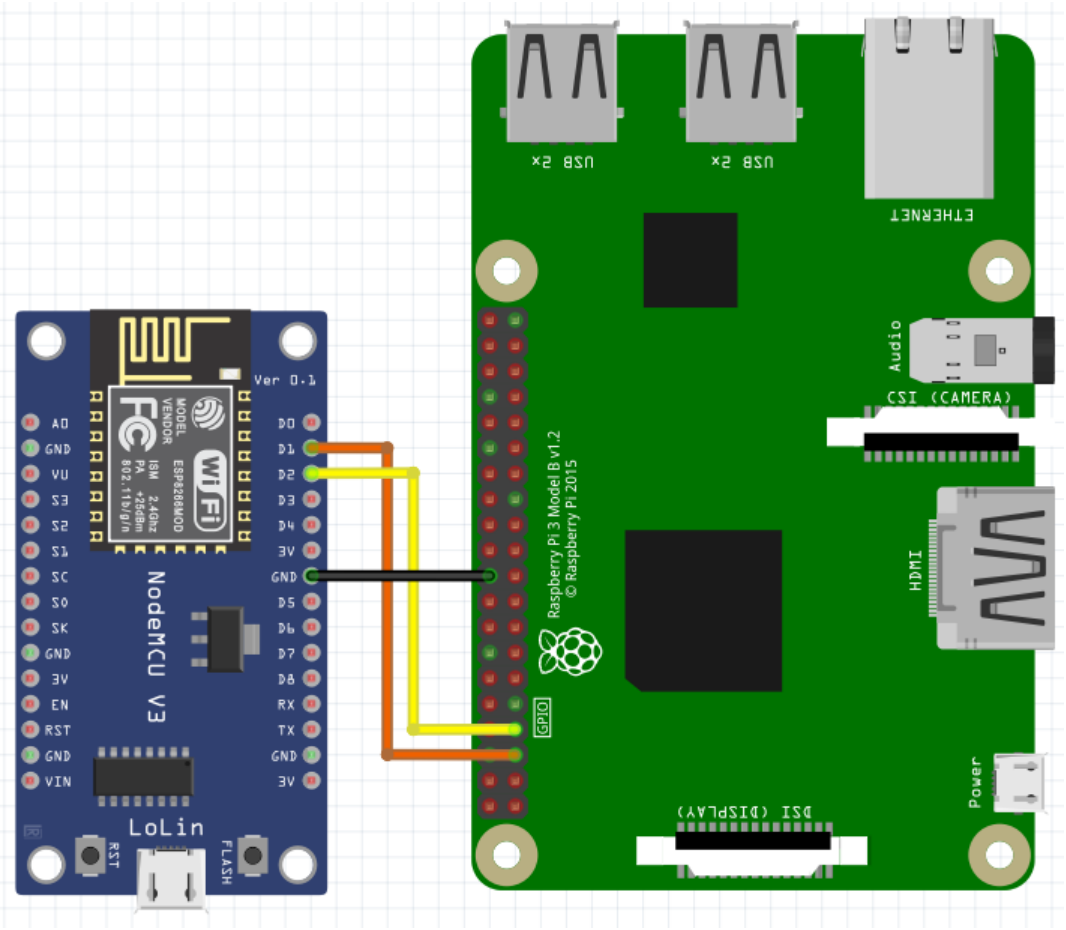
1. **Encender la Raspberry Pi desde el ESP:**
    - El ESP envía un pulso al GPIO3 de la Raspberry para conectarlo a GND durante un segundo. La Raspberry Pi se enciende, y el GPIO4 se establece en alto (`HIGH`) al completarse el arranque.
2. **Apagar la Raspberry Pi desde el ESP:**
    - El ESP envía un pulso al GPIO3 de la Raspberry para conectarlo a GND durante dos segundos. La Raspberry Pi procesa el apagado y establece el GPIO4 en bajo (`LOW`) antes de apagarse completamente.
3. **Evaluar el estado desde el ESP:**
    - Al conectarnos al servidor HTTP del ESP, este evalúa el estado del GPIO4 y muestra las opciones correspondientes:
        - Si la Raspberry Pi está encendida: muestra la opción de apagarla.
        - Si está apagada: muestra la opción de encenderla.

Con esta solución, aseguraremos una interacción confiable entre el ESP y la Raspberry Pi para encender y apagar el dispositivo de forma remota, reflejando su estado en tiempo real.

Conexionado:



- Conectamos el pin GPIO 5 del ESP8266 (llamado D1 en NodeMCU) al pin GPIO 3 de la Raspberry Pi con el propósito de enviar la señal de encendido/apagado
- Conectamos el pin GPIO 4 del ESP8266 (llamado D2 en NodeMCU) al pin GPIO 4 de la Raspberry para evaluar el estado (encendida o apagada) de la Raspberry
- Conectamos el GND del ESP al GND de la Raspberry Pi.



5. Creación del Script del ESP:

Primero crearemos dos librerias para usar con el script:

- **wifi.py** para gestionar la conexión wifi del dispòsitivo
- **sync_time_lib.py** para actualizar la hora real del dispositivo

- wifi.py

Con esta libreria creamos la clase `WifiConnection` para poder gestionar la conexión a redes Wi-Fi de manera eficiente si el dispositivo se mueve entre distintas ubicaciones y redes. Su principal objetivo es conectarse automáticamente a la mejor red disponible según una lista de redes predefinidas que proporcionamos al inicializar la clase.

1. **Inicialización y configuración**
    - Cuando instanciamos un objeto de la clase `WifiConnection`, el script activa la interfaz Wi-Fi del dispositivo y comienza automáticamente el proceso de conexión.

    - Le pasamos una lista de redes predefinidas (con su SSID y contraseña), el número máximo de intentos para cada red y el tiempo de espera entre intentos.
2. **Escaneo de redes disponibles**
    - La clase realiza un escaneo de las redes Wi-Fi visibles y las compara con la lista de redes predefinidas.
    - Da prioridad a las redes visibles de la lista para intentar la conexión primero con ellas, ya que tienen mayor probabilidad de éxito.
3. **Conexión a una red**
    - Si ya hay una conexión activa, la clase desconecta antes de intentar una nueva conexión.
    - Durante el proceso, realiza varios intentos según los parámetros configurados (número de intentos y retraso entre ellos).
    - Si la conexión es exitosa, muestra la dirección IP asignada por el router.
4. **Gestión de redes no visibles**
    - Si ninguna de las redes visibles coincide con las predefinidas, la clase intenta conectarse con las demás redes de la lista (en argumento con el diccionario de redes, pasaremos las no visibles primero para que sea por las que empiece en el caso de no coincidir con ninguna de las que ve).
5. **Desconexión manual**
    - También podemos desconectarnos manualmente del Wi-Fi actual si es necesario.

Características:

- **Automatización**: La clase gestiona automáticamente la conexión a redes Wi-Fi al iniciar, sin necesidad de nuestra intervención.
- **Optimización del uso en múltiples ubicaciones**: Ideal para dispositivos que se mueven entre diferentes redes dando preferencia a las redes visibles coincidentes para optimizar la velocidad de conexión
- **Reintentos configurables**: Nos permite ajustar el número de intentos y el tiempo de espera para cada red.
- **Fallback inteligente**: Si no encuentra coincidencias en redes visibles, recurre a intentar probar con todas las de la lista.
- **Simplicidad en la desconexión**: Nos ofrece un método para desconectar fácilmente del Wi-Fi actual.

Enlace -> libreria - wifi.py

- sync_time_lib.py

Esta librería tiene como objetivo sincronizar el tiempo del dispositivo con servidores NTP, ajustar la hora local al horario de España (incluyendo cambios por horario de verano/invierno) y devolver la hora en un formato fácilmente comprensible.

1. **Sincronización con NTP**
    - Usamos la función `sync_time` para sincronizarnos con servidores de tiempo mediante el módulo `ntptime`. Esto asegura que el dispositivo tenga la hora UTC más precisa disponible.
    - En caso de error durante la sincronización, devolvemos un mensaje indicando que no fue posible obtener los datos.
2. **Ajuste de la hora local**
    - Una vez sincronizados con UTC, aplicamos los ajustes correspondientes para el huso horario de España. Esto incluye la compensación por:
        - **Horario estándar** (UTC+1).
        - **Horario de verano** (UTC+2), que se aplica entre el último domingo de marzo y el último domingo de octubre.
    - Para determinar si estamos en horario de verano, verificamos si la fecha actual cae dentro del rango definido o si corresponde al último domingo de marzo/octubre, considerando las reglas oficiales.
3. **Formato personalizado**
    - Transformamos la hora local ajustada a un formato legible: `YYYY-MM-DD HH:MM:SS`.
    - Esto nos permite trabajar con la fecha y hora en un formato estándar y comprensible.
4. **Funciones auxiliares**
    - Implementamos varias funciones que nos ayudan a gestionar las fechas de manera precisa:
        - `last_day_of_month`: Calcula el último día de un mes específico.
        - `is_last_sunday_of_month`: Determina si una fecha es el último domingo de un mes, clave para los cambios de horario en España.
        - `local_formatted_time`: Ajusta la hora UTC sincronizada al huso horario local y la formatea.
5. **Opcional: Imprimir o devolver el resultado**
    - En la función `sync_time`, podemos elegir si deseamos imprimir directamente la hora sincronizada o devolverla como un valor para ser utilizada en otros procesos.

<u>Características</u>

- **Sincronización precisa:** Nos conectamos a servidores NTP para garantizar que la hora sea siempre exacta.
- **Soporte para horarios de verano/invierno:** Ajustamos automáticamente la hora local según las reglas de cambio de horario en España.
- **Formato estándar:** Ofrecemos la hora en un formato legible y estándar (`YYYY-MM-DD HH:MM:SS`).
- **Manejo de errores:** Si no podemos sincronizarnos, devolvemos un mensaje informativo en lugar de interrumpir el programa.
- **Uso flexible:** Podemos elegir entre imprimir directamente la hora sincronizada o reutilizarla en otros procesos.

- **Resiliencia:** Manejamos posibles errores de sincronización sin interrumpir el flujo del programa.

Enlace -> librería - sync_time_lib.py

- main.py (script de inicio)

Ell script principal en creado Micropython desde el ESP y que configuraremos como **servidor HTTP** para evaluar si la Raspberry Pi está encendida o apagada e interactuar con ella encendiéndola o apagándola en función de su estado dispondrá de las siguientes funcionalidades:

1. **Conexión Wi-Fi automática**
2. **Sincronización horaria**
3. **Control de la Raspberry Pi**
    - El script configura dos pines GPIO:
        - **PIN_RELAY**: Envía pulsos eléctricos para encender o apagar la Raspberry Pi.
        - **PIN_RASPBERRY_STATUS**: Verifica si la Raspberry Pi está encendida o apagada leyendo el estado de un pin GPIO configurado como entrada.
    - Dependiendo de la acción solicitada (encender o apagar), se envía un pulso eléctrico al relé para controlar la Raspberry Pi:
        - Un pulso breve (0.5 segundos) para encender.
        - Un pulso prolongado (2.5 segundos) para apagar de forma segura.
4. **Servidor HTTP**
    - El ESP8266 actúa como un servidor HTTP que responde a solicitudes de clientes web.
    - Tenemos una interfaz web sencilla con las siguientes funciones:
        - **Estado actual de la Raspberry Pi**: Mostramos si está encendida o apagada.
        - **Control de acciones**: Podemos encender o apagar la Raspberry Pi según su estado actual.
    - Cuando un usuario realiza una solicitud (por ejemplo, accede a `/encender` o `/apagar`), el servidor ejecuta la acción correspondiente y mostramos un mensaje de confirmación.
5. **Manejo de solicitudes**
    - Con la función `handle_request` analizamos las solicitudes HTTP recibidas y respondemos con páginas HTML predefinidas. Estas páginas incluyen:
        - **Página principal**: Muestra el estado de la Raspberry Pi y ofrece un botón para la acción disponible (encender o apagar).
        - **Página de estado**: Muestra un mensaje de confirmación después de una acción, como "Encendiendo..." o "Apagando...".
        - **Página de error**: Se muestra si se recibe una solicitud no válida.
6. **Limpieza y cierre**

    - El script incluye una función de limpieza (`cleanup`) que cierra el servidor, desconecta la red Wi-Fi y reinicia el ESP8266 en caso de interrupciones o errores.

Características:

- **Inicio automático**:
    - Al encender el ESP8266, este se conecta automáticamente a una red Wi-Fi de la lista configurada y sincroniza su hora.
    - Luego, inicia el servidor HTTP y queda listo para recibir solicitudes.
- **Control flexible de la Raspberry Pi**:
    - La Raspberry Pi puede encenderse o apagarse desde cualquier navegador web.
    - Se verifica el estado actual de la Raspberry Pi antes de permitir acciones, evitando comandos innecesarios.
- **Interfaz web intuitiva**:
    - Las páginas HTML generadas son simples, mostrando el estado actual, la hora sincronizada y botones para las acciones.
    - Incluimos retroalimentación visual y temporizadores para actualizar automáticamente la página después de acciones como encender o apagar.
- **Gestión horaria precisa**: Incluimos la sincronización con servidores NTP para asegurarnos que siempre se muestre la hora correcta en la que se generó la página.
- **Adaptabilidad**: El sistema puede ampliarse para incluir nuevas funcionalidades, como autenticación o integración con sistemas de domótica.

Enlace -> script - main.py

Una vez guardemos este script como `main.py` en el ESP, se ejecutará automáticamente cada vez que el dispositivo se reinicie. Esto se debe a que MicroPython ejecuta automáticamente cualquier archivo llamado `main.py` o `boot.py` almacenado en su sistema de archivos al inicio

Enlace -> video - evidencias funcionamento

---

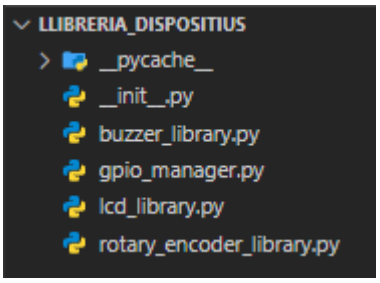
4. Pequeña interface visual y sonora de la Raspberry

- En este punto elaboraremos los scripts necesarios para implementar el menú del sistema con sus funcionalidades, en nuestro caso la de encendido/apagado será la que trabajaremos en profundidad, pero dejamos la puerta abierta a implementar otras.
- También agregaremos avisos sonoros en un buzzer para indicaciones, y en nuestro caso predefiniremos las de encendido y apagado pero dejamos la clase abierta para integrar más.
- Lo implementaremos mediante una librería de dispositivos modular que instalaremos como librería del intérprete python base del sistema
- Finalmente crearemos un script como servicio de de inicio para ejecutar todas las funcionalidades (desde interface gráfica, hasta escuchas de apagado por el GPIO3 o la inicialización del demonio de pigpio para poder usar el módulo)

1. Libreria modular (`llibreria_dispositius`)

En este proyecto, hemos implementado una biblioteca modular llamada `llibreria_dispositius` para gestionar todos los dispositivos que interactúan con nuestra Raspberry Pi. Esta estructura modular nos permite centralizar la lógica de control de dispositivos y facilitar su mantenimiento. Además, hemos decidido utilizar tanto `RPi.GPIO` como `pigpio`, seleccionando la biblioteca según las características específicas de cada dispositivo asegurando precisión en tareas críticas como la generación de PWM para el buzzer o el manejo de interrupciones para el encoder, mientras que mantendremos la simplicidad en dispositivos como el LCD con `RPi.GPIO`.

- Esta estructura modular nos facilitará la ampliación y el mantenimiento del proyecto a medida que evolucione.



- Instalaremos las librerías como módulos esto asegurará que las importaciones funcionen independientemente de la ubicación del script.
    - Crearemos un archivo **`setup.py`** en la carpeta **`/home/admin/Documents/llibreria_dispositius`**:

```
from setuptools import setup, find_packages

setup(
    name='llibreria_dispositius',
    version='1.0',
    packages=find_packages(),
)
```

- Instalaremos la librería globalmente**:**

  ```
  cd /home/admin/Documents/llibreria_dispositius
  sudo python3 setup.py install
  ```

- Las importaciones funcionarán así:

  ```
  from llibreria_dispositius.gpio_manager import
  gpio_setmode, gpio_cleanup
  from llibreria_dispositius.buzzer_library import Buzzer
  ```

Usaremos tanto `RPi.GPIO` como `pigpio`

1. **`RPi.GPIO`**
   - Es una biblioteca ligera que utilizaremos para operaciones básicas como configurar pines de entrada y salida y manejar señales simples en el **LCD**.
   - Su uso está gestionado en el submódulo `gpio_manager`.
2. **`pigpio`**
   - Es más avanzado y preciso, especialmente en operaciones que requieren PWM (modulación por ancho de pulso) o control de interrupciones.
   - Requiere que un demonio esté corriendo en la Raspberry Pi (`pigpiod`), lo que facilita una comunicación eficiente y precisa.
   - Hemos elegido `pigpio` para:
     - **Buzzer**: El PWM basado en hardware de `pigpio` permite generar tonos y melodías con alta precisión.
     - **Rotary Encoder**: Ofrece un manejo superior de interrupciones para registrar rotaciones y pulsaciones sin errores.

1. Submódulo `gpio_manager`

Este submódulo proporciona funciones de configuración y limpieza de los pines GPIO utilizando `RPi.GPIO`.

- **Funciones principales**:
  - `gpio_setmode(mode)`: Configura el modo GPIO (BCM o BOARD).
  - `gpio_cleanup(pins_l)`: Limpia la configuración de los pines especificados o de todos los pines si no se proporciona una lista.
- **Uso en otros submódulos**: Este módulo actúa como base para dispositivos como el LCD.

Enlace -> submódulo - gpio_manager.py

2. Submódulo `lcd_library`

El submódulo `lcd_library` está diseñado para gestionar un LCD de dos líneas en modo de 4 bits. Este módulo nos permite interactuar de forma sencilla con el LCD, ya sea para mostrar información básica o para crear animaciones como desplazamientos de texto.

Un aspecto fundamental de `lcd_library` es el uso de un diccionario llamado `lcd_commands`. En este diccionario hemos mapeado los nombres descriptivos de los comandos a sus valores en formato hexadecimal

Con este diccionario nos abstraemos de la complejidad del hardware, ya que podemos utilizar comandos legibles (como `"Clear Display"`) y asignarles fácilmente su valor hexadecimal tal y como vienen definidos en los datasheets del dispositivo.

El submódulo incluye varias funciones clave que utilizan los comandos del diccionario para interactuar con el LCD:

- **`send_command(command)`**:
  - Envía un comando al LCD.
  - Busca el valor hexadecimal correspondiente en el diccionario `lcd_commands` y lo transmite dividiéndolo en dos bloques de 4 bits.
- **`send_byte(data, is_data)`**:
  - Divide un byte en dos mitades (4 bits cada una) y las envía al LCD.
  - Usa el pin `RS` para indicar si estamos enviando un comando (`is_data=False`) o datos (como caracteres para mostrar, `is_data=True`).
- **`display(title, content)`**:
  - Nos permite escribir texto en las dos líneas del LCD: Primera línea para título y segunda para el contenido.
- **`lcd_init()`**:
  - Realiza la secuencia de inicialización del LCD en modo de 4 bits.
- **`lcd_clear()`**:
  - Limpia la pantalla utilizando el comando `"Clear Display"`.
- **Animaciones**:
  - Con la función `_animate_gradually(content, delay)` crearemos un efecto de desplazamiento para textos largos. Esto se logrará actualizando continuamente la segunda línea del LCD con ventanas de texto de 16 caracteres.

Enlace -> submódulo - lcd_library.py

3. Submódulo `buzzer_library`

Este submódulo nos permite generar tonos, melodías y notificaciones sonoras con precisión, aprovechando el PWM basado en hardware. Además, utilizaremos un diccionario

de notas musicales que optimizamos según el rango de frecuencias soportado por el buzzer, con lo que garantizamos un rendimiento acústico adecuado.

El buzzer funciona generando señales PWM con frecuencias específicas que corresponden a notas musicales. Estas frecuencias están determinadas por el estándar musical occidental, pero las adaptamos al rango operativo del buzzer. Esto nos permite reproducir tonos precisos, melodías predefinidas y notificaciones configurables.

1. **Uso de `pigpio`**:
    - Utilizamos `pigpio` para generar PWM con alta precisión.
    - El módulo requiere que el demonio `pigpiod` esté corriendo en la Raspberry Pi para funcionar.
2. **Diccionario de notas (`NOTES`)**:
    - Creamos un diccionario que mapea nombres de notas musicales (como `C1`, `D#4`) a sus frecuencias en hercios.
    - Inicialmente, generamos un conjunto completo de notas, pero lo reducimos al rango óptimo del buzzer mediante parámetros ajustables (`min_frequency` y `max_frequency`), definidos por defecto en `2100 Hz` y `4800 Hz`, respectivamente.
    - Con este enfoque garantizamos que el buzzer funcione en su rango ideal, evitando frecuencias inaudibles o ineficaces.

Mediante la función `generate_notes_in_freq_range` construiremos el diccionario `NOTES` con las notas musicales dentro de un rango específico. Este diccionario nos servirá de base para interpretar y reproducir melodías.

- **Cálculo de frecuencias**:
    - Partimos de una frecuencia base (`C1 = 32.70 Hz`) y calculamos las frecuencias de las notas utilizando la fórmula:

$$f = f_{base} \times 2^{\frac{\text{semitones desde C1}}{12}}$$

- **Filtrado por rango**:
    - Usamos los parámetros `min_frequency` y `max_frequency` para incluir solo las frecuencias que se encuentran dentro del rango óptimo especificado por el fabricante del buzzer.
    - Como resultado, obtenemos un subconjunto de notas musicales que el buzzer puede reproducir eficazmente.
- **Notas especiales**:
    - Hemos incluido una entrada especial en el diccionario: `"silenci"`, que corresponde a `0 Hz`. Esto nos permitirá pausar el sonido entre notas.

El submódulo incluye diversas funciones para interactuar con el buzzer:

1. **`play_tone(self, frequency, duration)`**:
    - Genera un tono con una frecuencia específica (`frequency`) durante un tiempo determinado (`duration`).
    - Usa `pigpio.hardware_PWM` para lograr precisión en la señal.
2. **`play_melody(self, melody)`**:
    - Reproducirá una secuencia de notas definida por una lista de tuplas `(nota, duración)`.
    - Convierte las notas en frecuencias utilizando el diccionario `NOTES`.
    - Si una nota no está definida en el diccionario, utilizaremos la frecuencia de `"silenci"` (`0 Hz`).
3. **`defined_melodies(self, name, duration=0.1)`**:
    - Proporciona melodías predefinidas que usaremos en el encendido y apagado como:
        - **"welcome"**: Escala ascendente dentro del rango del buzzer.
        - **"shutdown"**: Escala descendente.
    - Estas melodías pueden personalizarse o ampliarse según las necesidades del proyecto.
4. **`stop_pwm(self)`**:
    - Detiene el PWM y limpia el estado del buzzer, asegurando que no quede activo después de una operación.
5. **`stop(self)`**:
    - Libera todos los recursos asociados al buzzer, cerrando la conexión con el demonio `pigpiod` y configurando el pin como entrada.

Enlace -> submódulo - buzzer_library.py

4. Submódulo `rotary_encoder_library`

Este componente es fundamental para implementar una interfaz física en nuestro proyecto, permitiendo la navegación por menús y la selección de opciones mediante giros y pulsaciones. Para garantizar un control preciso y eficiente, utilizamos `pigpio`, que ofrece un manejo avanzado de **interrupciones**.

El submódulo `rotary_encoder_library` incluye diversas funciones para interactuar con el encoder rotatorio:

1. **`_handle_rotation(self, gpio, level, tick)`**:
    - Callback que maneja los cambios de estado en `CLK`.
    - Compara los estados actuales de `CLK` y `DT` para determinar el sentido del giro:
        - **Giro horario**: Incrementa un contador interno.
        - **Giro antihorario**: Decrementa el contador.
    - Llama al callback de rotación que definiremos en el script que use el módulo..

2. `_handle_button(self, gpio, level, tick)`:
    - Callback que maneja las pulsaciones del botón (`SW`).
    - Llama al callback de botón definido en el script..
3. `get_value(self)`:
    - Devuelve el valor acumulado del encoder, que representa el número total de giros realizados.
4. `disable_callbacks(self)`:
    - Desactiva las interrupciones registradas, eliminando los callbacks asociados.
5. `stop(self)`:
    - Libera los recursos de `pigpio`, deteniendo su conexión y desactivando el manejo de pines.

Enlace -> submódulo - rotary_encoder_library.py

2. Script de Inicio Configurado como Servicio de `systemd` para el lanzamiento de la interfaz

Este script será el motor principal de nuestra interface de usuario. Con él configuramos el entorno necesario para nuestro proyecto, inicializando los dispositivos, mostrando un mensaje de bienvenida y gestionando un menú interactivo. Su ejecución está diseñada para integrarse con `systemd`, garantizando que se inicie automáticamente junto con la Raspberry Pi y proporcione las funcionalidades esenciales desde el arranque.

El objetivo principal es inicializar y coordinar los dispositivos controlados por nuestra biblioteca modular `llibreria_dispositius` y proporcionar una interfaz física mediante un **LCD** y un encoder rotatorio.

- Muestra un mensaje de bienvenida al inicio.
- Proporciona un menú interactivo para gestionar el sistema.
- Permite apagar la Raspberry Pi de forma segura mediante un botón físico o una opción del menú.
- Sincroniza el hilo principal con el mensaje de bienvenida para garantizar un flujo ordenado.

Hemos configurado el script para ejecutarse como un servicio de `systemd`. Para ello reutilizaremos el servicio creado shutdown-button.service que iniciaba la escucha del GPIO3 para un apagado externo de la raspberry y lo reubicamos para que lance **rb_manager.py**

```
admin@raspberrypi:/etc/systemd/system $ cat shutdown-button.service
[Unit]
Description=Botón de apagado en GPIO
After=multi-user.target

[Service]
Type=simple
ExecStart=/usr/bin/python3 /etc/init/rb_manager.py
Restart=always
User=root
WorkingDirectory=/home/admin/Documents/llibreria_dispositius

[Install]
WantedBy=multi-user.target
```
:

Estructura del script

1. **Configuración inicial**:
   - Se configura el modo GPIO como `BCM` mediante el submódulo `gpio_manager`.
   - Se inicializan todos los dispositivos necesarios:
     - **LCD**: Configuración de pines y preparación para mostrar mensajes.
     - **Buzzer**: Configuración para reproducir tonos y melodías.
     - **Encoder rotatorio**: Manejo de giros y pulsaciones mediante callbacks.
     - **Botón físico**: Configuración para detectar pulsaciones prolongadas.
2. **Ejecución del mensaje de bienvenida**:
   - **Uso de hilos**:
     - El mensaje de bienvenida se ejecuta en un hilo secundario utilizando la función `threading.Thread`.
     - Esto permite que el flujo principal continúe inicializando otros dispositivos en paralelo.
   - **Sincronización con el hilo principal**:
     - Antes de mostrar el menú interactivo, el flujo principal espera a que el hilo del mensaje de bienvenida termine utilizando `welcome_thread.join()`.
3. **Gestión del menú interactivo**:
   - El menú interactivo se define mediante un diccionario (`MENU_STRUCTURE`), que permite navegar por las opciones y seleccionar submenús.
   - El encoder rotatorio gestiona:
     - **Giros**: Navegar entre las opciones del menú.
     - **Pulsaciones**: Seleccionar una opción o confirmar acciones.
   - Una de las opciones permite apagar la Raspberry Pi de forma segura.
4. **Reproducción de melodías**:
   - El buzzer proporciona retroalimentación sonora para las siguientes acciones:
     - **Inicio del sistema**: Melodía de bienvenida.
     - **Apagado del sistema**: Melodía de despedida.
5. **Bucle principal**:

- Mantiene el script en ejecución para responder a eventos como giros, pulsaciones o comandos de apagado.

Funciones principales

1. **Inicialización de dispositivos**:
   - `start_pigpiod()`: Asegura que el demonio `pigpiod` esté activo.
   - `start_lcd_menu()`: Gestiona el mensaje de bienvenida.
   - `play_welcome_melody()` y `play_shutdown_melody()`: Gestionan las notificaciones sonoras.
2. **Gestión del menú**:
   - `rotation_callback(direction)`: Detecta giros en el encoder y actualiza el menú mostrado en el LCD.
   - `button_callback()`: Detecta pulsaciones en el encoder para seleccionar opciones.
3. **Apagado seguro**:
   - `shutdown_pressed(channel)`: Detecta pulsaciones prolongadas del botón físico.
   - `shutdown_system()`: Gestiona el apagado, incluyendo la limpieza de recursos y la reproducción de la melodía de despedida.

Ventajas

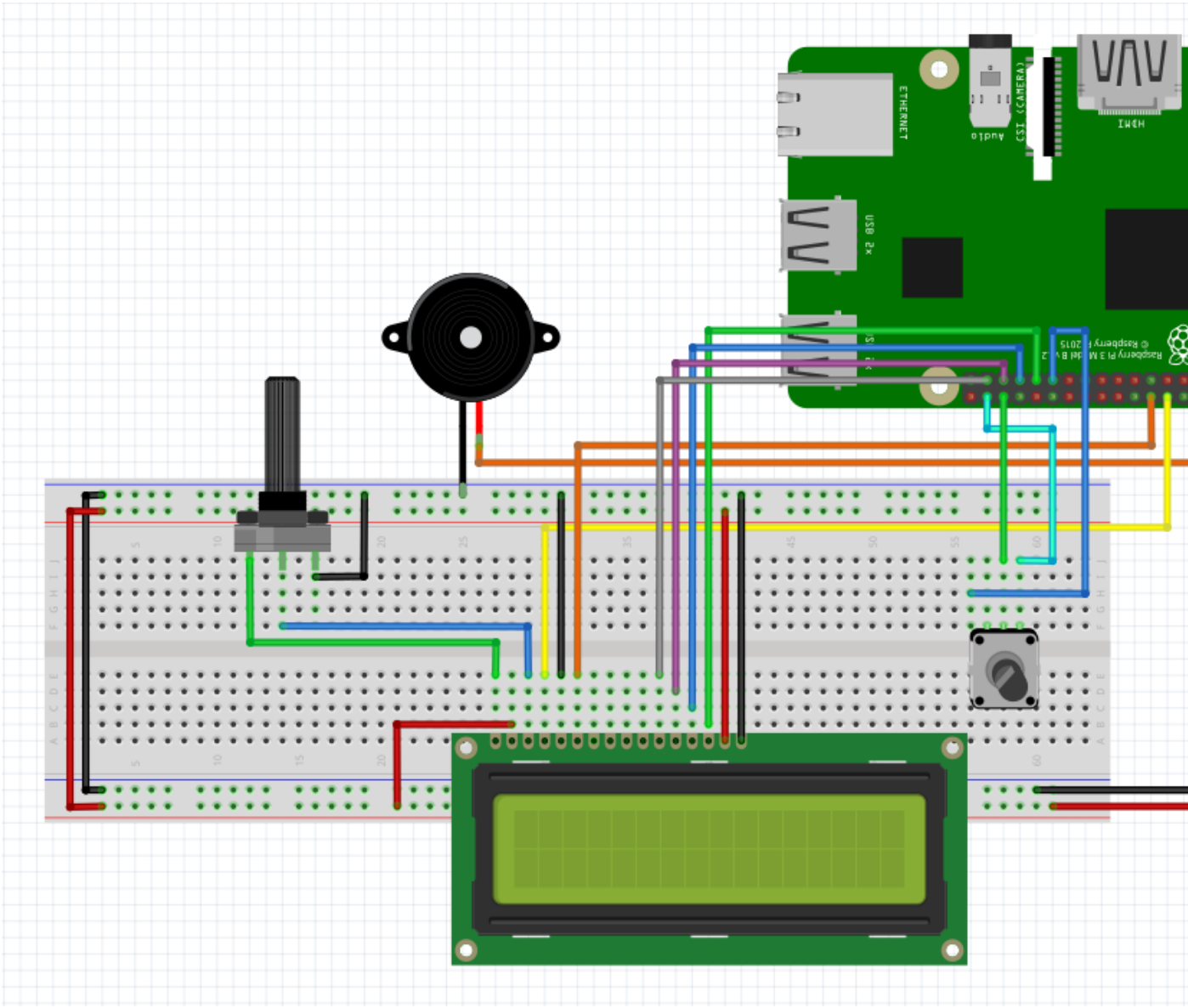
1. **Ejecución concurrente**:
   - Al usar un hilo para el mensaje de bienvenida, optimizamos el tiempo de inicio, permitiendo que el flujo principal continúe mientras el mensaje se muestra.
2. **Interfaz intuitiva**:
   - Proporciona una interacción sencilla y eficiente mediante el LCD y el encoder rotatorio.
3. **Robustez**:
   - Asegura un apagado seguro, evitando daños en el sistema de archivos.
4. **Feedback auditivo**:
   - Mejora la experiencia del usuario mediante melodías informativas en momentos clave.
5. **Escalabilidad**:
   - La estructura modular del menú y las funciones permite ampliar fácilmente las funcionalidades.

Enlace -> script - rb_manager.py

---

5. Montaje y funcionamiento

Montaje:



[Enlace -> video - evidencias funcionamento]