En la Raspberry, los pines GPIO se manejan a través de /sys/class/gpio/. En un contenedor LXC de Debian, sabiendo que Linux es «todo archivo», podemos intentar simularlos creando la estructura de archivos que se crea automáticamente en la raspberry, pero deberemos hacerlo en una carpeta diferente a /sys/class/gpio dado que en un contendor LXC el acceso a /sys/class/gpio/ está bloqueado para proteger el host. Incluso aunque fuese un contenedor **privilegiado**, al no tener acceso al hardware real, los GPIOs no estarían disponibles. El acceso a /sys/class/gpio/ depende de módulos del kernel como gpiod o gpiochip, los cuales no están presentes en un contenedor LXC a menos que el host los exponga explícitamente. Dado que el objetivo es simular los GPIOs en software, crearemos una estructura en /tmp/gpio/ que imita exactamente /sys/class/gpio/, pero sin necesidad de acceso al kernel.

Eso sí, antes de proceder con la creación de las carpetas y archivos que simularán el GPIO, debemos entender dos cosas:

1 – Aunque la raspberry tiene 40 pines, algunos son de voltaje, otros de masa, etc. Este es el esquema de pines de la raspberry, a la izquierda como Board/BCM y a la derecha como PLC:

```
       3V3 power < [01] [02] > 5V power           | 3V3 power < [01] [02] > 5V power
    GPIO 2 (SDA) < [03] [04] > 5V power           |    %IX0.0 < [03] [04] > 5V power
    GPIO 3 (SCL) < [05] [06] > Tierra             |    %IX0.1 < [05] [06] > Tierra
 GPIO 4 (GPCLK0) < [07] [08] > GPIO 14 (TXD)      |    %IX0.2 < [07] [08] > %QX0.0
          Tierra < [09] [10] > GPIO 15 (RXD)      |    Tierra < [09] [10] > %QX0.1
        GPIO 17 < [11] [12] > GPIO 18 (PCM_CLK)   |    %IX0.3 < [11] [12] > %QW0 (PWM)
        GPIO 27 < [13] [14] > Tierra              |    %IX0.4 < [13] [14] > Tierra
        GPIO 22 < [15] [16] > GPIO 23             |    %IX0.5 < [15] [16] > %QX0.2
       3V3 power < [17] [18] > GPIO 24            | 3V3 power < [17] [18] > %QX0.3
  GPIO 10 (MOSI) < [19] [20] > Tierra             |    %IX0.6 < [19] [20] > Tierra
   GPIO 9 (MISO) < [21] [22] > GPIO 25            |    %IX0.7 < [21] [22] > %QX0.4
  GPIO 11 (SCLK) < [23] [24] > GPIO 8 (CE0)       |    %IX1.0 < [23] [24] > %QX0.5
          Tierra < [25] [26] > GPIO 7 (CE1)       |    Tierra < [25] [26] > %QX0.6
  GPIO 0 (ID_SD) < [27] [28] > GPIO 1 (ID_SC)     |         - < [27] [28] > -
          GPIO 5 < [29] [30] > Tierra             |    %IX1.1 < [29] [30] > Tierra
          GPIO 6 < [31] [32] > GPIO 12 (PWM0)     |    %IX1.2 < [31] [32] > %QX0.7
  GPIO 13 (PWM1) < [33] [34] > Tierra             |    %IX1.3 < [33] [34] > Tierra
GPIO 19 (PCM_FS) < [35] [36] > GPIO 16            |    %IX1.4 < [35] [36] > %QX1.0
          GPIO 26 < [37] [38] > GPIO 20 (PCM_DIN) |    %IX1.5 < [37] [38] > %QX1.1
          Tierra < [39] [40] > GPIO 21 (PCM_DOUT) |    Tierra < [39] [40] > %QX1.2
```

De ese gráfico podemos concluir que realmente, funcionales, sólo tiene 26 pines GPIO.

2 – Hay que diferenciar entre la numeración física, la numeración BCM y la numeración de PLC, según se requiera referirse a los pines de una u otra forma:

Numeración física (BOARD)

- Basada en la posición de los pines en el conector GPIO de 40 pines.
- Va de 1 a 40 en los modelos actuales.
- No es la usada en /sys/class/gpio/.

Numeración BCM (Broadcom Chip Model)

- Se basa en cómo el chip Broadcom asigna los pines GPIO.
- Es la numeración usada en /sys/class/gpio/ en Raspberry Pi.
- Va desde GPIO2 hasta GPIO27 en los modelos comunes.

Numeración PLC

- Basada en las posiciones de memoria de un PLC.
- Tiene 25 pines útiles (entre entradas y salidas).

Esta tabla explica mejor la relación entre los pines, según se mire:

Numeración Física (BOARD)	Numeración BCM	Numeración PLC (IEC 61131-3)	Tipo (Entrada/Salida)
3	GPIO2	%IX0.0	Entrada
5	GPIO3	%IX0.1	Entrada
7	GPIO4	%IX0.2	Entrada
11	GPIO17	%IX0.3	Entrada
13	GPIO27	%IX0.4	Entrada
15	GPIO22	%IX0.5	Entrada
19	GPIO10	%IX0.6	Entrada
21	GPIO9	%IX0.7	Entrada
23	GPIO11	%IX1.0	Entrada
29	GPIO5	%IX1.1	Entrada
31	GPIO6	%IX1.2	Entrada
33	GPIO13	%IX1.3	Entrada
35	GPIO19	%IX1.4	Entrada
37	GPIO26	%IX1.5	Entrada
8	GPIO14	%QX0.0	Salida
10	GPIO15	%QX0.1	Salida
16	GPIO23	%QX0.2	Salida
18	GPIO24	%QX0.3	Salida
22	GPIO25	%QX0.4	Salida
24	GPIO8	%QX0.5	Salida
26	GPIO7	%QX0.6	Salida
32	GPIO12	%QX0.7	Salida
36	GPIO16	%QX1.0	Salida
38	GPIO20	%QX1.1	Salida

40	GPIO21	%QX1.2	Salida

Entonces, para simular en entorno de carpetas y archivos, basándonos en esta última tabla, lo mejor es que diferenciemos entre pines de salida y de entrada (independientemente de que los pines de la RBP se puedan usa para ambas cosas) y usemos una estructura que nos sea funcional también para simular un PLC. Esto es, usar los pines GPIO 2, 3, 4, 17, 27, 22, 10, 9, 11, 5, 6, 13, 19 y 26 para entradas y los pines GPIO 14, 15, 23, 24, 25, 8, 7, 12, 16, 20 y 21 para salida. Haremos todo esto creando un script:

```
nano /root/SimularPinesGPIORaspberry.sh; chmod +x /root/SimularPinesGPIORaspberry.sh
```

…con el siguiente texto:

```
#!/bin/bash

# Definir pines GPIO compatibles con entradas PLC (%IX)
  aPinesDeEntrada=(2 3 4 17 27 22 10 9 11 5 6 13 19 26)

# Definir pines GPIO compatibles con Salidas PLC (%QX)
  aPinesDeSalida=(14 15 23 24 25 8 7 12 16 20 21)

# Crear la carpeta general
  mkdir -p /tmp/rbp-gpio-simul

# Crear las carpetas para las entradas
  for vPinDeEntrada in "${aPinesDeEntrada[@]}"; do
    mkdir -p    "/tmp/rbp-gpio-simul/gpio$vPinDeEntrada"
    echo "in" > "/tmp/rbp-gpio-simul/gpio$vPinDeEntrada/direction"
    echo "0"  > "/tmp/rbp-gpio-simul/gpio$vPinDeEntrada/value"
  done

# Crear las carpetas para las salidas
  for vPinDeSalida in "${aPinesDeSalida[@]}"; do
    mkdir -p     "/tmp/rbp-gpio-simul/gpio$vPinDeSalida"
    echo "out" > "/tmp/rbp-gpio-simul/gpio$vPinDeSalida/direction"
    echo "0"   > "/tmp/rbp-gpio-simul/gpio$vPinDeSalida/value"
  done
```

…y lo ejecutaremos

```
/root/SimularPinesGPIORaspberry.sh
```

…para crear la estructura de archivos y carpetas con la que luego interactuaremos mediante otros scripts.

Entonces, ahora que tenemos la estructura de carpetas creada, podemos, por ejemplo, levantar un servidor OPCua mediante python que, de acuerdo con la configuración de pines que acabamos de crear, monitorice cada uno de los pines simulados, de forma que, por ejemplo con UaExpert, podamos leer lo que ocurre en cada uno de ellos, dado que todos serán agregado como un nodo a FreeOPCua.

Para visualizar el contenido del script hacemos click aquí. Para descargarlo directamente en nuestro Debian:

```
curl -L
https://raw.githubusercontent.com/nipegun/ot-scripts/refs/heads/main/OPCua/Servidor-SimulandoGPIOdeRBP-TodoBooleano.py
-o /tmp/Servidor-SimulandoGPIOdeRBP-TodoBooleano.py
```

Para ejecutarlo, primero instalamos los paquetes necesarios:

```
sudo apt -y update
sudo apt -y install python3-pip
sudo apt -y install python3-cryptography
pip3 install opcua --break-system-packages
```

…y luego, simplemente:

```
python3 /tmp/Servidor-SimulandoGPIOdeRBP-TodoBooleano.py
```

El script toma todas las entradas y salidas como booleanas, pero es un buen ejemplo para ver como, metiendo ceros y unos en los archivos value, dentro de cada carpeta /tmp/rbp-gpio-simul/gpioXX los valores que observemos en UaExpert irán cambiando de true a false y viceversa.

De distinta forma, este otro script que he creado, permite definir manualmente los nodos, de manera que podamos elegir que tipo de datos tiene cada nodo específico que definamos, mientras que los que no definamos manualmente, se crearán como booleanos, tal y como lo hacía el script anterior. Para descargarlo directamente en el Debian:

```
curl -L https://raw.githubusercontent.com/nipegun/ot-scripts/refs/heads/main/OPCua/Servidor-SimulandoGPIOdeRBP-ManualYRestoBooleano.py -o /tmp/Servidor-SimulandoGPIOdeRBP-ManualYRestoBooleano.py
```

Para ejecutarlo:

```
python3 /tmp/Servidor-SimulandoGPIOdeRBP-ManualYRestoBooleano.py
```

En este nuevo script, se específica que tipo de datos manejará cada una de las entradas. OPC UA, en su especificación, define un conjunto de tipos de datos “built-in” (nativos) que puede manejar. En la práctica, cualquier nodo OPC UA puede usar cualquiera de estos tipos (y muchos otros compuestos), independientemente de si está asociado a un GPIO, a un sensor, a una base de datos, etc. Pero a grandes rasgos, los datos built-in, según la especificación de OPC UA, son:

- **Boolean**: Valor lógico true/false.
- **SByte**: Entero de 8 bits con signo (rango: -128 a 127).
- **Byte**: Entero de 8 bits sin signo (0 a 255).
- **Int16**: Entero de 16 bits con signo (-32768 a 32767).
- **UInt16**: Entero de 16 bits sin signo (0 a 65535).
- **Int32**: Entero de 32 bits con signo.
- **UInt32**: Entero de 32 bits sin signo.
- **Int64**: Entero de 64 bits con signo.
- **UInt64**: Entero de 64 bits sin signo.
- **Float**: Coma flotante de 32 bits (precisión simple).
- **Double**: Coma flotante de 64 bits (precisión doble).
- **String**: Cadena de texto (Unicode).
- **DateTime**: Fecha/hora en formato OPC UA (similar a un datetime).
- **Guid**: Identificador global único (UUID).
- **ByteString**: Secuencia de bytes arbitraria.
- **XmlElement**: Nodo XML (poco común para un GPIO, pero soportado).
- **NodeId**: Identificador de un nodo en OPC UA (usado internamente en el modelo de datos).
- **ExpandedNodeId**: Variante extendida de NodeId (incluye namespace adicional, etc.).
- **StatusCode**: Código de estado (indica errores o estados de respuesta de OPC UA).
- **QualifiedName**: Nombre calificado (usado en la estructura de nodos de OPC UA).
- **LocalizedText**: Texto con información de localización (traducciones, etc.).
- **ExtensionObject**: Permite encapsular tipos de datos complejos/definidos por el usuario.
- **DataValue**: Estructura que combina el valor, timestamps, calidad, etc.

- **Variant**: Tipo genérico de OPC UA que puede llevar cualquiera de los anteriores.
- **DiagnosticInfo**: Información de diagnóstico (usado internamente para reportar errores).

¿Qué tipos se usan normalmente con GPIO?

En la mayoría de casos, cuando se exponen GPIO en un servidor OPC UA (como entradas/salidas de una Raspberry Pi), se usan tipos primitivos como:

- **Boolean**: para entradas/salidas digitales (botones, relés, LEDs…)
- **Int16 / Int32 / Float / Double**: para valores analógicos (sensores de temperatura, humedad, tensión, corriente…)
- **String**: si quisieras, por ejemplo, pasar texto o mensajes por un GPIO (algo menos común).

Pero, estrictamente, OPC UA no impide usar cualquiera de los tipos listados. Por ejemplo, podríamos definir un GPIO como Int64 si esperamos manejar rangos muy grandes o ByteString si quisieramos guardar datos binarios arbitrarios, etc.

Sabiendo todo esto, y teniendo corriendo el segundo script en background, podríamos crear otro script en python que simule ser un sensor DHT22 de temperatura, que irá metiendo los valores en el pin de entrada correspondiente. Lo normal es meterlo en el pin GPIO4 (%IX0.2). Entonces, teniendo en cuenta que el pin de datos del sensor emite esos datos con formato de 40 bits (16 bits para la humedad, en formato entero o con parte decimal según la implementación, 16 bits para la temperatura y 8 bits para un checksum), y teniendo en cuenta que, casualmente, tenemos el pin GPIO4 ya como una entrada, el script nos podría quedar así. Para descargarlo:

```
curl -L https://raw.githubusercontent.com/nipegun/ot-scripts/refs/heads/main/OPCua/Sensor-DHT22-SimularEnSimuladorGPIO.py -o /tmp/Sensor-DHT22-SimularEnSimuladorGPIO.py
```

Para ejecutarlo:

```
python3 /tmp/Sensor-DHT22-SimularEnSimuladorGPIO.py
```

El script se ejecutará, independientemente de que tengamos en background la ejecución del servidor FreeOPCua. Eso sí, si queremos monitorizar los datos con UaExpert, deberemos tener corriendo el script de servidor que creamos antes y deberemos conectarnos con UaExpert a la IP del servidor.