Control PI de un motor DC y monitorización con real-time Linux

Máster en Sistemas Electrónicos Avanzados

Resumen. Este proyecto consiste en la implementación de un sistema de control y de monitorización para la velocidad de giro de un motor DC. Las tareas real-time, como la adquisición de los pulsos del encoder y el cálculo de la señal de control, se implementan en una Raspberry Pi 4 B con un parche real-time Linux (PREEMPT-RT). Las tareas de monitorización (graficar datos adquiridos y modificar consigna), de más alto nivel, se implementan en un PC conectado a la Raspberry. La comunicación entre los dos dispositivos se realiza mediante ZMQ, mientras que la comunicación entre tareas dentro de la Raspberry se realiza con sockets de UNIX.

Índice

1 Introducción

El objetivo planteado en este proyecto es la implementación de un controlador de velocidad de un motor DC en una Raspberry Pi. También se incluye como objetivo la implementación de un sistema de monitorización del motor y su controlador en un PC conectado con la Raspberry. Las tareas de monitorización consisten en realizar una gráfica dinámica de la señal de control y la velocidad de giro, así como dar la posibilidad de realizar un cambio de la consigna al usuario.

La naturaleza multitarea del proyecto hace necesario involucrar ciertos protocolos de comunicación. Como protocolo de comunicación interno entre las distintas tareas implementadas en la Raspberry, se utilizan los sockets de UNIX (cliente/servidor). Para la comunicación entre la Raspberry y el PC se utiliza ZMQ (publisher/subscriber), de más alto nivel.

Las tareas de más bajo nivel (que son además las más críticas desde el punto de vista temporal), relacionadas con la adquisición de datos del encoder del motor y el control PI del mismo, que acceden a los GPIOs de la Raspberry, están escritas en código C. Las tareas de más alto nivel (las del PC y las que comunican la Raspberry con el PC) están escritas en Python.

El control del motor resulta realizable debido a que se ha introducido un parche real-time (PREEMPT-RT) en el sistema operativo Linux de la Raspberry. Esto último brinda la posibilidad de lanzar hilos (*threads*) de tiempo real (real-time, RT). Las tareas RT son aquellas cuyo correcto funcionamiento depende no sólo de que sus cálculos lógicos, matemáticos o de otra naturaleza sean correctos a nivel lógico, sino también de que dicho resultado se produzca en un instante temporal adecuado. Es decir, un sistema RT no sólo debe hacer bien la tarea, sino que debe hacerla cuándo debe [1].

En el ámbito de este proyecto, el hecho de utilizar un sistema real-time es imprescindible para garantizar que la actualización de la señal de control se realiza de forma sincronizada con un periodo de muestreo fijo, que en este caso es de 100 ms. Esta sincronización se realiza utilizando un reloj interno de la Raspberry.

Este documento tiene la siguiente estructura:

- La sección 1 es una introducción al proyecto.
- En la sección 2 se describe el hardware (motor, encoder y puente en H) y el controlador PI.
- En la sección 3 se describen los distintos programas y tareas por las que está compuesto el proyecto de software y cómo se comunican entre ellas.
- En la sección 4 se explica cómo se ha verificado el diseño y se proporciona un ejemplo de su correcto funcionamiento.
- La sección 5 son las conclusiones del proyecto.

2 Descripción del hardware y del controlador PI

2.1 Descripción del motor de continua

El motor utilizado en este trabajo es un motor de continua del fabricante Shayang Ye Industrial Co., LTD., con part number IG220019X00015R [2], que se muestra en la Figura 1. Este motor es solidario a una reductora de factor 1:53 (por cada vuelta del motor, la reductora da 53 vueltas), que lleva un encoder en cuadratura magnético (de efecto Hall, con 3 ranuras por vuelta). Este encoder se utiliza para medir la velocidad a la que está girando el motor [3].

Figura 1. Fotografía del motor IG220019X00015R del fabricante Shayang Ye Industrial Co [4].

El objetivo es poder controlar la velocidad angular de giro del motor entre 20 rpm y 150 rpm (tanto en un sentido como en el contrario).

2.2 Puente en H

Para aplicar la tensión eléctrica a los terminales del motor, que determina la corriente que circula por su devanado, se utiliza la tarjeta PmodHB5TM 2A H-Bridge Module de Digilent. En la Figura 2 se muestra una fotografía de la tarjeta [5].

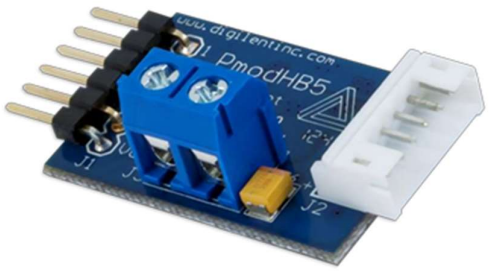

Figura 2. Fotografía de la tarjeta de puente en H de Digilent utilizada [5].

En la Figura 3 se muestra un esquema simplificado de un puente en H y su funcionamiento básico [6]:

- En la parte izquierda se puede ver que con los interruptores S1 y S4 cerrados se aplica una diferencia de tensión positiva entre los terminales del motor.

- En la parte derecha se ve que con los interruptores S3 y S2 cerrados se aplica una diferencia de tensión negativa entre los terminales del motor.

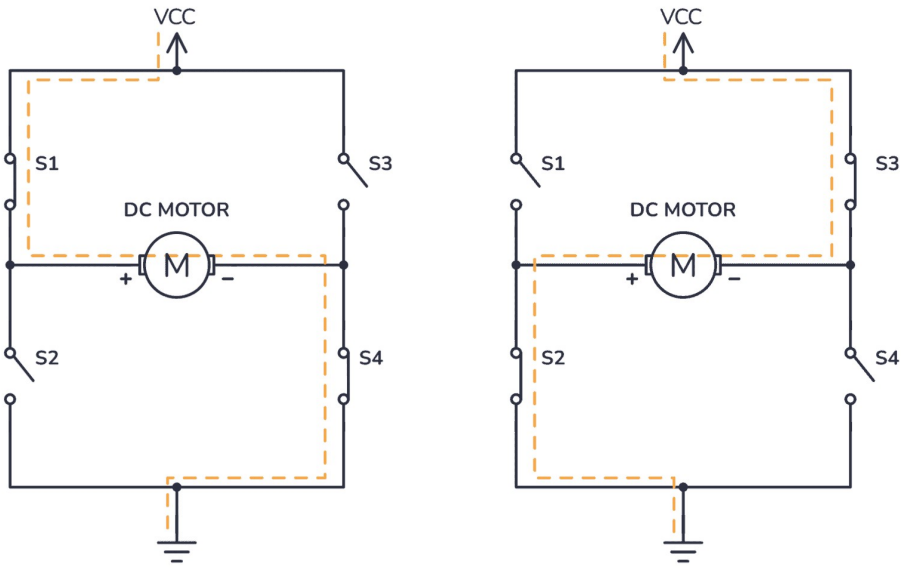

Figura 3. Estructura simplificada y funcionamiento básico de un puente en H, aplicando una diferencia de tensión positiva entre los terminales del motor (izquierda) y aplicando una diferencia de tensión negativa (derecha) [6].

En este proyecto, se utiliza un valor de $V_{cc} = 6\ \mathrm{V}$ para alimentar el puente en H (suministrado por una fuente de alimentación enchufable comercial, similar a un cargador de teléfono móvil). Para modular la tensión aplicada al puente (entre $-6\ \mathrm{V}$ y $+6\ \mathrm{V}$), se utiliza modulación PWM (*Pulse Width Modulation*, modulación por ancho de pulso).

En la Figura 4 se muestra el diagrama de bloques de la tarjeta de puente en H de Digilent. La interfaz de la tarjeta consta de tres conectores [5]:

- J1: este conector proporciona la interfaz digital para conectarse con el dispositivo digital que controla el motor, en este caso, una Raspberry Pi.
    1. En el puerto DIR se aplica una señal que controla la polaridad de la tensión aplicada en el motor, permitiendo variar entre los dos estados de la Figura 3.
    2. En el puerto EN se aplica la PWM que se utiliza para conmutar el puente en H.
    3. El puerto SA corresponde a la señal A proporcionada por el encoder.
    4. El puerto SB corresponde a la señal B proporcionada por el encoder, en cuadratura con A.
    5. Conexión con la tierra del dispositivo digital.
    6. Conexión con la alimentación de 3.3 V o 5 V del dispositivo digital.
- J2: proporciona la interfaz de conexión con el propio motor y su encoder.
- J3: constituye la conexión con la fuente de alimentación de 6 V para los transistores del puente en H.

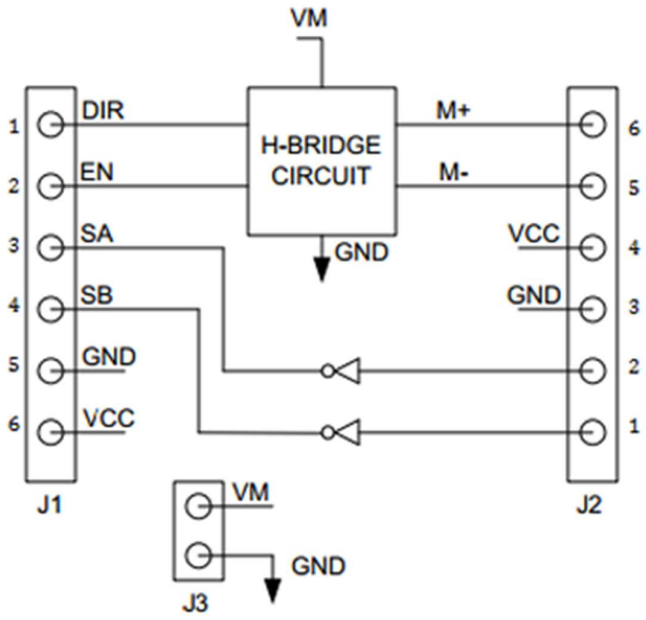

Figura 4. Diagrama de bloques de la tarjeta PmodHB5TM 2A H-Bridge Module de Digilent [5].

2.3 Raspberry Pi

Como dispositivo para implementar el controlador, se utiliza una Rasbperry Pi (modelo 4 B, con 4 GB de RAM), que a su vez se conecta mediante USB (ethernet) con un PC. De cara a realizar la conexión con la tarjeta del puente en H, es importante tener en cuenta la distribución de los pines GPIO, que se puede ver en la Figura 5 [7].

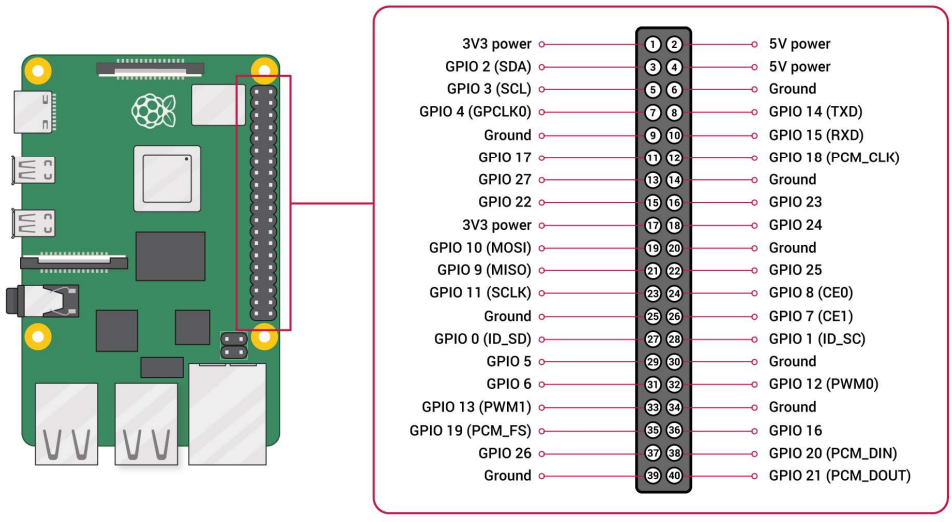

Figura 5. Distribución de los pines GPIO de la Raspberry Pi [7].

2.4 Conexión entre la Raspberry Pi y el puente en H

En la Tabla 1 se muestran las conexiones a realizar entre los puertos GPIO de la Raspberry Pi y los puertos de la tarjeta puente en H de Digilent. Desde el punto de vista de la Raspberry, el GPIO17 es de salida (polaridad de la tensión aplicada), el GPIO18 se configura para generar una PWM a nivel hardware y GPIO22 y GPIO23 son puertos de entrada (señales SA y SB del encoder).

Tabla 1. Conexiones entre los puertos GPIO de la Raspberry Pi y los puertos de la tarjeta puente en H de Digilent.

Raspberry Pi GPIO #	Digilent H Bridge Port
11: GPIO17	1: DIR
12: GPIO18 (PCM_CLK)	2: EN
15: GPIO22	3: SA
16: GPIO23	4: SB
6: Ground	5: GND
1: 3V3 power	6: VCC

2.5 Descripción del controlador PI

Para realizar el proyecto, se utiliza un controlador PI ya ajustado. El ajuste lo ha proporcionado el profesor Koldo Basterretxea, y corresponde a un laboratorio que se realiza en la asignatura Sistemas Electrónicos Digitales del Grado en Ingeniería Electrónica Industrial y Automática. El controlador continuo se ha ajustado utilizando el PID Tuner de Matlab/Simulink, y su función de transferencia es [3]:

$$C(s) = K_\mathrm{p} + K_\mathrm{i}\frac{1}{s} \quad ; \quad K_\mathrm{p} = 0.001 \quad ; \quad K_\mathrm{i} = 0.0891 \tag{1}$$

Para discretizar el controlador, se utiliza la transformación de Tustin o bilineal [3]:

$$C(z) = K_\mathrm{p} + K_\mathrm{i}\frac{T_\mathrm{s}}{2}\frac{z+1}{z-1} = \frac{b_1 z^{-1} + b_0}{a_1 z^{-1} + 1} \tag{2}$$

Identificando coeficientes, se obtienen los siguientes valores para a_1, b_1 y b_0 para un periodo de muestreo $T_\mathrm{s} = 100\ \mathrm{ms}$:

$$a_1 = -1.0, b_0 = 0.005455, b_1 = 0.003455$$

Para implementar el controlador en código C, se utiliza la forma directa 3D [3]:

$$u(k) = b_0 e(k) + b_1 e(k-1) - a_1 u(k-1) \tag{3}$$

Donde $e(k)$ hace referencia a la señal error en el instante kT_s y $u(k)$ a la señal de control en el instante kT_s.

3 Descripción del proyecto

El diseño está constituido por 7 tareas, distribuidas en 5 programas: *control_motor.c*, *bridge.py, commands_subscriber.py*, *commands_publisher.py* y *plot.py*. El programa central, que realiza el control del motor y la adquisición de los datos del encoder, es *control_motor.c*. Este programa es el que contiene las únicas dos tareas RT del sistema. Las otras cinco tareas no son RT.

En la Figura 6 se muestra un diagrama de las distintas tareas del sistema de monitorización y control del motor DC, indicando la interconexión entre ellas y con el mundo exterior, sus

funciones principales y en qué fichero fuente está su código. Las líneas de puntos azules verticales indican la separación entre las distintas interfaces. De izquierda a derecha, se observa el motor y su hardware asociado (tarjeta puente en H y encoder), la Raspberry Pi, el PC y, finalmente, la interfaz con el usuario.

A continuación, se describe brevemente la función de cada programa y las tareas por las que está compuesto:

- ***control_motor.c:*** este programa está constituido por 2 hilos RT.
    - **Task 1 (*thread1*):** este hilo se ejecuta una vez por periodo de muestreo (100 ms) y se encarga de calcular la señal de control $u(k)$, según el valor de la consigna (*sp*) y de las cuentas del decoder (task 2). En base a *uk*, actualiza el ciclo de trabajo de la PWM (generada en hardware) y la señal *dir*, que determina la polaridad de la tensión aplicada al motor. Este hilo también tiene dos servidores UNIX socket:
        - Socket server 1: se encarga de enviar un *timestamp*, la señal de control y la velocidad en RPMs a la tarea 3 de *bridge.py*.
        - Socket server 2: cada vez que el usuario actualiza la consigna desde el PC, este servidor recibe un mensaje con el nuevo valor de la consigna y se encarga de actualizar el valor de la variable *sp*.

      Como *control_motor.c* es el programa más crítico, pues es el que realiza el control RT de la planta, se establece como servidor para permitir su ejecución al margen de que el resto de los programas (en los que están los clientes) estén o no activos.
    - **Task 2 (*thread2*):** este hilo se encarga de decodificar los pulsos SA y SB generados por el encoder en cuadratura, utilizando para ello una máquina de estados. Esta máquina de estados actualiza un contador que es utilizado por *thread1* para determinar las RPMs a las que está girando el motor. La actualización de la máquina de estados y del contador tiene lugar cada vez que hay un flanco en SA o en SB.
- ***bridge.py:*** este programa está formado por dos hilos, que no son de tiempo real.
    - **Task 3 (*thread3*):** este hilo actúa como cliente UNIX socket, y recibe los datos enviados por socket server 1 desde la tarea 1. Tras decodificar los datos recibidos, los introduce en una FIFO que se utiliza para comunicarse con task 4.
    - **Task 4 (thread4):** este hilo actúa como publisher ZMQ. Mientras la FIFO no esté vacía, extrae un dato de la FIFO y lo envía al programa *plot.py* en el PC a través de un socket ZMQ con un patrón publisher/subscriber.
- ***plot.py:*** este programa tiene una única tarea, task 5. Esta tarea actúa como subscriber ZMQ y recibe los datos enviados por el publisher de la tarea 4. A continuación, grafica de manera dinámica los datos recibidos (RPMs y señal de control) utilizando la librería pyqtgraph.
- ***commands_publisher.py:*** este programa se encuentra en el ordenador y, al igual que el anterior, tiene una única tarea, task 6. Esta consiste en solicitar al usuario un nuevo valor para la consigna (entre -150.0 y 150.0 rpm). Posteriormente, esta tarea actúa como publisher ZMQ para enviar el nuevo valor de la consigna al programa *commands_subscriber.py*, en la Raspberry Pi, que actúa como subscriber.
- ***commands_subscriber.py:*** este programa consta de una única tarea, task 7. Actúa como subscriber ZMQ para recibir el valor de la consigna introducido por el usuario en *commands_subscriber.py*, que a su vez actúa como publisher ZMQ. Tras decodificar el valor recibido, recodifica dicha información y, como cliente UNIX socket, la transmite al segundo servidor del programa *control_motor.c*, que se encarga de actualizar el valor de la variable *sp*.

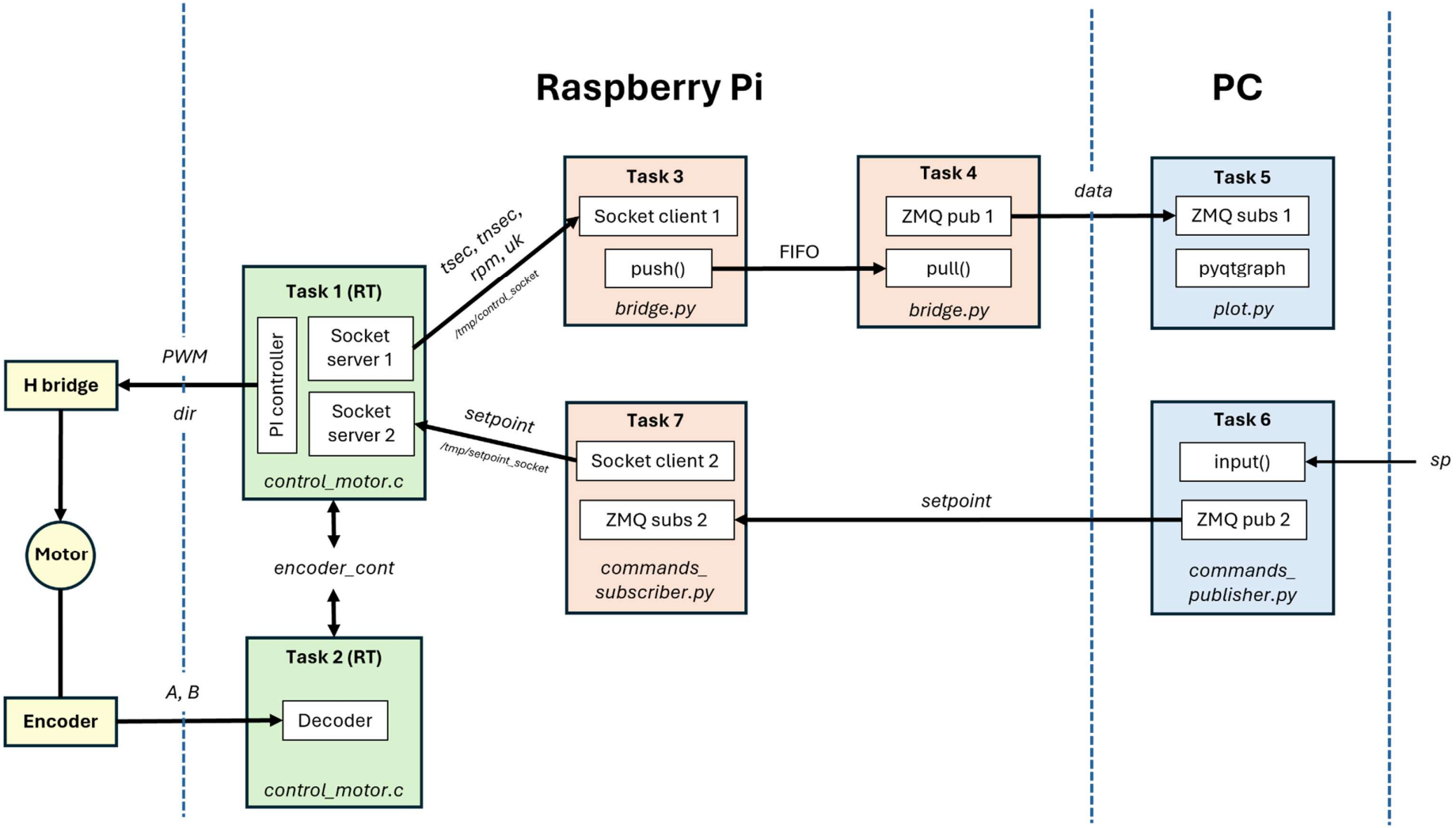

Figura 6. Esquema de las distintas tareas del sistema de monitorización y control del motor DC, indicando la interconexión entre ellas y con el mundo exterior, sus funciones principales y en qué fichero fuente está su código.

Es importante tener en cuenta por qué en el puente de comunicación entre el programa *control_motor.c* y el PC (o a la inversa) se han incluido dos etapas, una con sockets de UNIX y otra con ZMQ. El motivo de utilizar un protocolo de bajo nivel como son los sockets de UNIX para la comunicación de *control_motor.c* con otras tareas de la Raspberry Pi es mantener la carga de las tareas RT baja, para no comprometer el funcionamiento del aspecto más crítico del diseño: adquirir adecuadamente los datos del encoder y actualizar cada 100 ms la señal de control. La comunicación de más alto nivel, entre la Raspberry Pi y el PC, se realiza utilizando ZMQ, con protocolo publisher/subscriber, que es más sofisticado, pero también implica una mayor carga computacional.

El código completo está disponible en el repositorio github https://github.com/ooscaar93/Project-estructuras-software-para-SOPC. A continuación, se describen con mayor detalle algunos aspectos relevantes de las distintas tareas.

3.1 Programa *control_motor.c*

Este programa consta de 2 hilos RT, *thread1* y *thread2*. Para lanzar los hilos y configurarlos para que sean de tiempo real, se utiliza la API POSIX. Los hilos se lanzan en el programa principal, y se realizan configuraciones en su política de *scheduling*, prioridad y tamaño de pila para que sean de tiempo real. El código para lanzar los hilos se ha obtenido de [8].

Al crear cada hilo, se le asigna una función que es la que se va a ejecutar una vez lanzado el hilo:

- A *thread1* se le asigna la función *thread1_control*.
- A *thread2* se le asigna la función *thread2_encoder*.

En el programa principal, también se inicializan los GPIOs, llamando a la función *gpio_init()*. La librería utilizada para el acceso a los GPIOs es pigpio, una librería específica para la Raspberry. La documentación de sus funciones puede encontrarse en la referencia [9]. En este proyecto:

- Se configura GPIO18 como PWM hardware (correspondiente a EN en el puente en H).
- GPIO17 se configura como salida (correspondiente a DIR en el puente en H).
- GPIO22 y GPIO23 se configuran como entradas (SA y SB en el puente en H).

También en el programa principal, se crean los dos servidores UNIX socket que se utilizan para comunicarse con otros programas (*server_socket* y *server_socket2*). El proceso para crear el socket se describe con detalle en [10], pero a grandes rasgos consiste en llamar a las siguientes funciones:

- *socket()*: crear un socket de UNIX.
- *bind()*: asignar un nombre al socket, y asociarlo con un fichero local.
- *listen()*: quedarse a la escucha de conexiones de clientes al servidor. En este caso, ambos servidores se configuran con una cola de conexión máxima de un cliente.

Es importante mencionar que los dos sockets se crean como no bloqueantes para no obstaculizar la ejecución de las tareas de tiempo real. Esto significa que, si cuando el servidor está aceptando la conexión de un cliente no hay ningún cliente que haya solicitado conectarse, se pasa a la siguiente instrucción, en lugar de quedarse esperando a una conexión [11].

A continuación, se describen las dos tareas (hilos RT) del programa *control_motor.c*.

3.1.1 Tarea 1: controlador + sockets (RT)

Esta tarea corresponde al hilo *thread1* del programa *control_motor.c*. La función asociada a este hilo es *thread1_control*, y consiste en:

1. Declaración e inicialización de variables.
2. Obtener instante actual del reloj y esperar un segundo, para sincronizar la ejecución del bucle infinito de esta tarea.
3. Bucle infinito, que consta de:
   i. Espera hasta que el reloj llegue al instante correspondiente al siguiente periodo de muestreo (100 ms), utilizando la función *clock_nanosleep()*.
   ii. Cálculo de las RPMs a partir de la variable *encoder_cont* (contador del decoder), y puesta a cero de *encoder_cont*. Para ello, se multiplica la cuenta *encoder_cont* por el siguiente factor:

$$rpm = encoder\_cont \cdot \frac{60}{53 \cdot 3 \cdot 0.1 \cdot 4} \quad (4)$$

   Donde el factor 60 corresponde al número de segundos en un minuto, el factor 53 a la relación 1:53 de la reductora del motor, el factor 3 al número de ranuras en cada disco del encoder, el factor 0.1 al periodo de muestreo en segundos y el factor 4 al número de incrementos/decrementos de *encoder_cont* producidos en una vuelta de la máquina de estados del decoder (sección 3.1.2).
   iii. Cálculo de la señal de error y de la señal de control.

$$e(k) = sp(k) - rpm(k) \quad (5)$$

$$u(k) = b_0 e(k) + b_1 e(k-1) - a_1 u(k-1) \quad (6)$$

   iv. Saturación de *uk* entre -6 V y 6 V, aplicación de zona muerta si |*uk*| < 0.6 V y aplicación de un periodo con ciclo de trajo nulo si cambia la polaridad de la señal de control (para evitar cortocircuitos en las ramas del puente en H).
   v. Actualización de la señal de polaridad de la tensión aplicada (señal DIR del puente en H) y del ciclo de trabajo de la PWM (señal EN del puente en H). La frecuencia de la PWM es de 2 kHz, por las limitaciones debidas al tiempo de conmutación de los transistores del puente en H.
   vi. Si la conexión con el cliente UNIX socket en *bridge.py* está abierta, enviar *timestamp*, *rpm* y *uk* al cliente. Aquí la secuencia del servidor es [10]:
      a) *accept()*: esperar a que un cliente establezca la conexión. Como el socket es no-bloqueante, si el cliente no ha solicitado conexión se continúa con la siguiente instrucción.
      b) *write()*: en caso de que se haya establecido la conexión, se envían los datos al cliente.
      c) *close()*: se cierra la conexión con el cliente.
   vii. Si la conexión con el cliente UNIX socket en *commands_subscriber.py* está abierta, el servidor lee el dato enviado por el cliente y actualiza la consigna *sp*. Los pasos son idénticos al otro servidor, pero utilizando *read()* en lugar de *write()*.
   viii. Cálculo del instante temporal en el que se tiene que volver a ejecutar el código, 100 ms después que el anterior *clock_nanosleep()*.

3.1.2 Tarea 2: decoder (RT)

Esta tarea corresponde al hilo *thread2* del programa *control_motor.c*. El código del hilo consiste en la configuración de una función llamada *edgeDetected()* para que se ejecute cada vez que se

produzcan flancos en las señales SA y SB del encoder. Para ello, se configuran los correspondientes GPIOs con la función *gpioSetAlertFunc()* de la librería *pigpio* [9].

La función *edgeDetected()* actualiza el valor de dos variables estáticas, *a* y *b*, que contienen el estado de las señales asociadas a los discos A y B del encoder en cuadratura. Después, la función implementa una máquina de estados que cuenta con 4 posibles estados: 1 (correspondiente a (*a, b*) = (0, 0)), 2 (1, 0), 3 (1, 1) y 4 (0, 1). Esta máquina de estados se encarga de realizar el incremento/decremento del contador *encoder_cont* según las transiciones producidas en SA y SB. Para ello, se tiene en cuenta que:

- En una de las direcciones de giro la secuencia que se produce para (*a, b*) es (0, 0) → (1, 0) → (1, 1) → (0, 1) → (0, 0) → etc. Tal y como está conectada la tarjeta del puente en H con el motor, en cada transición de estado hay que decrementar el contador.
- En la otra dirección la secuencia es (0, 0) → (0, 1) → (1, 1) → (1, 0) → (0, 0) → etc. En este caso, en cada transición de estado se incrementa el contador.

El diagrama de transición de estados de la máquina que actualiza el contador se muestra en la Figura 7. Esta máquina de estados se ha diseñado teniendo en cuenta la referencia [12].

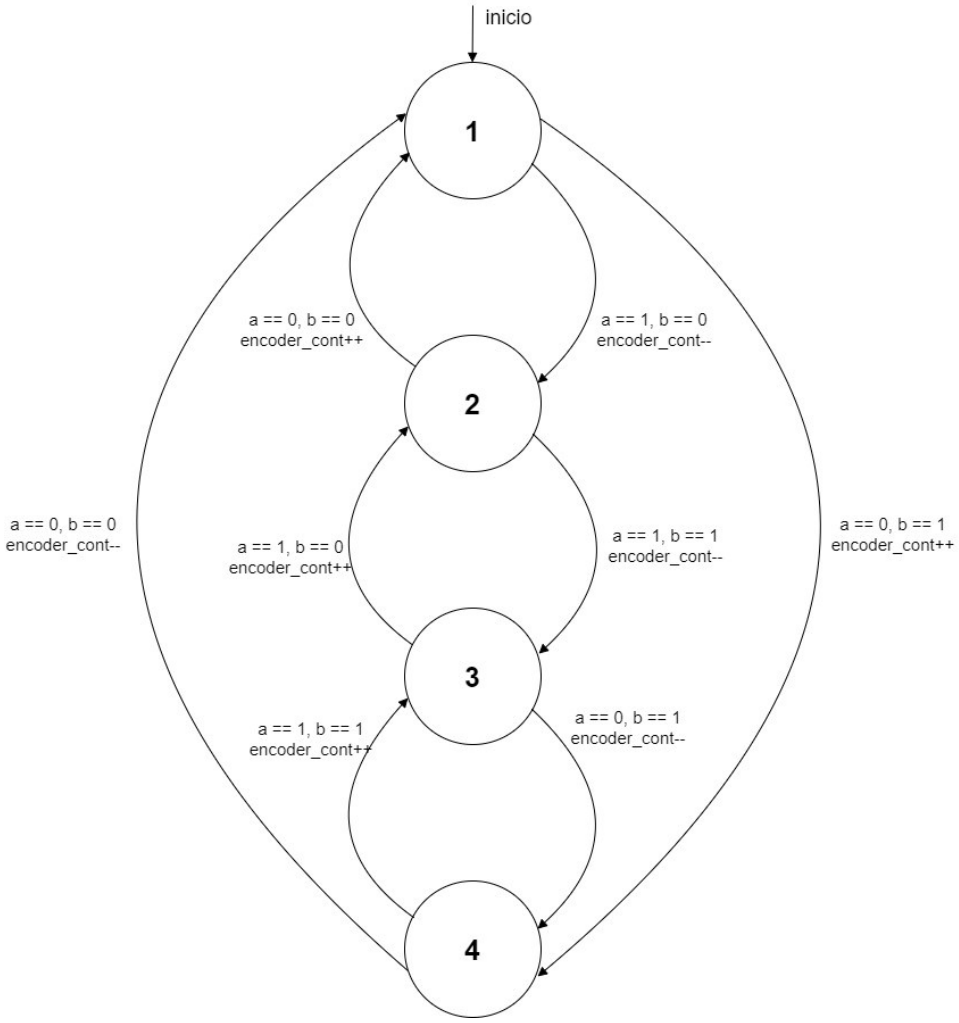

Figura 7. Diagrama de transición de estados de la máquina de estados del decoder.

3.2 Programa *bridge.py*

Este programa se encarga de hacer de puente entre *control_motor.c*, en la Raspberry Pi, y *plot.py*, en el PC. En el programa principal, se lanzan dos hilos *thread3* y *thread4*, correspondientes a las tareas 3 y 4, respectivamente. Estos hilos no son de tiempo real.

Para la comunicación entre ambos hilos, se crea una cola FIFO de 1000 elementos, utilizando la clase *Queue* del módulo *queue* de Python [13].

3.2.1 Tarea 3: client socket + FIFO

Esta tarea se encarga de recibir la señal de control y la salida de la planta enviadas por el servidor 1 de *control_motor.c*. Para ello, se realiza la siguiente secuencia [10]:

- *socket()*: crear socket UNIX.
- *connect():* conectarse al servidor.
- *recv():* recibir un mensaje del servidor.

Tras recibir la respuesta, se decodifica la información y se introduce en la FIFO que se comunica con la tarea 4, que a su vez envía la información al PC. Finalmente, se cierra la conexión con el servidor utilizando la función *close()*.

3.2.2 Tarea 4: FIFO + ZMQ publisher

En esta tarea, mientras la FIFO no esté vacía, se van sacando los datos almacenados en la FIFO y se envían a través de ZMQ al PC. Para ello, se utiliza un protocolo publisher/subscriber, en el que esta tarea actúa como publisher y el programa *plot.py* del PC como subscriber.

El código utilizado para implementar el publisher ZMQ se ha tomado de la referencia [14]. A la hora de hacer el *bind()*, se utiliza la dirección "tcp://*5555".

3.3 Programa *plot.py*

Este programa se ejecuta en el PC. Recibe los datos publicados por la tarea 4 de *bridge.py*, y va realizando una gráfica dinámica de la señal de control *uk* y de la velocidad de giro del motor *rpm*, utilizando para ello la librería de Python pyqtgraph. Solamente tiene una tarea, la tarea 5.

3.3.1 Tarea 5: ZMQ subscriber + plot

Esta tarea consiste en ir ploteando los datos que se reciben del Publisher ZMQ en una gráfica actualizada dinámicamente realizada con la librería pyqtgraph.

Para hacer una aplicación que realice un gráfico con pyqtgraph, se importa la librería PyQt5 y se crea un widget. Esto se consigue mediante la definición de una función de inicialización (*__init__*) y una función de actualización (*update_plot_data*) para la clase *MainWindow(QtWidgets.QMainWindow*).

El código se ha tomado de la referencia [15]. La función *__init__* es el constructor de la clase, y se encarga de crear la ventana de la aplicación con las dos gráficas (*rpm* y *uk*), de esperar a recibir el primer dato de *bridge.py* para saber cuál es el *timestamp* inicial (que se toma como referencia para el instante inicial 0 segundos) y de configurar un timer que cada 50 ms llama a la función *update_plot_data*.

Esta última función se encarga de actualizar la gráfica cada vez que se recibe un dato de *bridge.py*. Para la recepción de los datos, esta tarea actúa como subscriber ZMQ, conectándose a la dirección "tcp://raspberrypi-javi:5555". En cada caso, habrá que cambiar el nombre de la

Raspberry Pi por el del dispositivo concreto que se utilice. En cada actualización de la gráfica, se elimina el punto temporal más antiguo y se añade el nuevo dato recibido.

3.4 Programa *commands_publisher*.py

Este programa lee el valor de la consigna introducido en la consola del PC por el usuario y lo envía al programa *commands_subscriber.py* de la Raspberry Pi mediante ZMQ. Consta de una única tarea, la tarea 6.

3.4.1 Tarea 6: read setpoint + ZMQ publisher

Esta tarea consiste en pedir de manera continua una nueva consigna al usuario por la terminal de Python, y mandarla a través de un socket ZMQ al programa *commands_subscriber.py* de la Raspberry Pi. La tarea 6 actúa como publisher, y utiliza la dirección "tcp://*5556".

3.5 Programa *commands_subscriber.py*

Este programa (en la Raspberry Pi) se encarga de recibir los comandos enviados por *commands_publisher.py* (en el PC) y enviárselos al programa *control_motor.c* para actualizar la consigna. Consta de una única tarea, la tarea 7.

3.5.1 Tarea 7: ZMQ subscriber + socket client

Esta tarea, que actúa como subscriber ZMQ, se encarga de recibir la información referente a la consigna enviada por el publisher ubicado en el PC (*commands_publisher.py*). A continuación, la vuelve a codificar y la envía a través de un socket UNIX a la tarea 1 del programa *control_motor.c*. La tarea 7 se comporta como cliente del socket.

La dirección a la que se suscribe el *subscriber* es la dirección "tcp://169.254.1.2:5556", que corresponde a la conexión ethernet de la Raspberry con el PC. Para que esta dirección sea fija, hay que configurarla desde la configuración de ethernet del PC. Los seis primeros dígitos han de coincidir con la subred correspondiente a la Raspberry, que en este caso es la 169.254. Para los últimos dígitos, se asigna 1.2, ya que la dirección acabada en 1.1 ya está ocupada y corresponde a la propia Raspberry. Como máscara de subred, se utiliza la 255.255.255.0, y se utiliza el DNS de Google (8.8.8.8).

4 Verificación del diseño

En la Figura 8 se muestra el montaje del sistema completo. En la parte derecha se puede ver el ordenador, conectado con la Raspberry Pi que está en la parte izquierda. La Raspberry Pi está conectada a la tarjeta del puente en H (de color azul), que a su vez está conectada con el motor y el encoder.

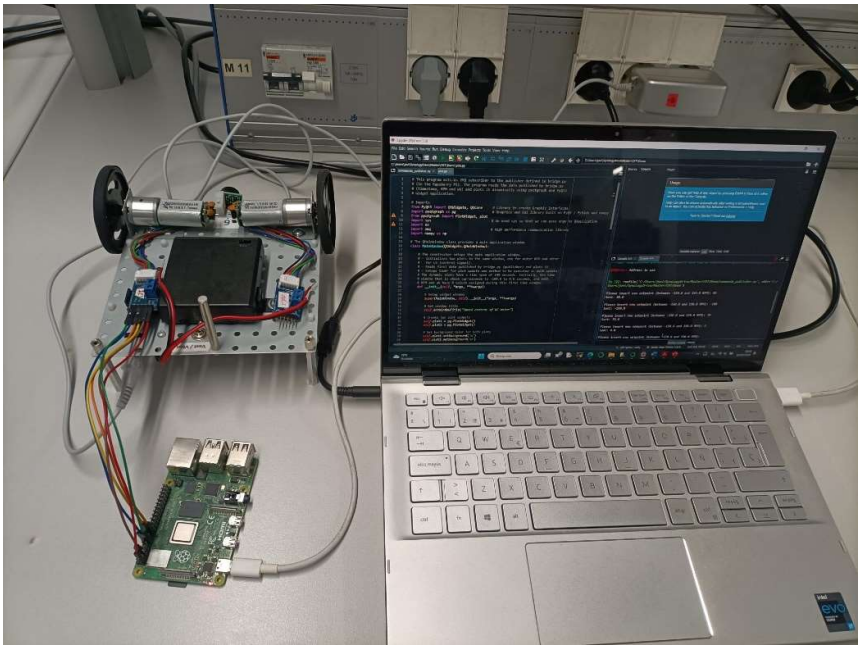

Figura 8. Montaje del sistema completo.

4.1 Verificación de la velocidad de giro

El correcto funcionamiento del diseño se ha comprobado de manera elemental con un punto blanco sobre la rueda a controlar y un cronómetro. Con el objetivo de comprobar que la rueda gira a la velocidad establecida en la consigna, se cuenta el número de veces que la marca pasa por la posición inicial en un tiempo determinado. Efectivamente, este sencillo procedimiento ha permitido comprobar que el sistema al completo funciona de manera correcta asignando una consigna de 60 rpm y comprobando que a lo largo de un minuto la marca pasa exactamente 60 veces por la posición inicial.

En un proyecto de mayor envergadura esta no hubiese sido la manera correcta de abordar esta etapa del diseño. Una opción preferible sería aislar las diferentes partes involucradas en el control (decoder, función de transferencia y PWM) y testar cada una de ellas por separado. Para ello una posibilidad es decantarse por los procedimientos de diseño y verificación basados en modelos que en la actualidad se están perfilando como los estándares en la industria.

4.2 Verificación del sistema de monitorización

Por último, se verifica el funcionamiento del sistema completo. Para ello, en el PC se abren los programas *commands_publisher.py* y *plot.py*. Mediante *commands_publisher.py*, se asigna la secuencia de consignas 0 rpm, 100 rpm, -40 rpm, 80 rpm, 0 rpm. En la Figura 9 se muestran las gráficas generadas por el programa *plot.py* en el PC. Se comprueba, que el motor gira a las velocidades indicadas por la consigna, y que se puede visualizar la señal de control *uk* en voltios.

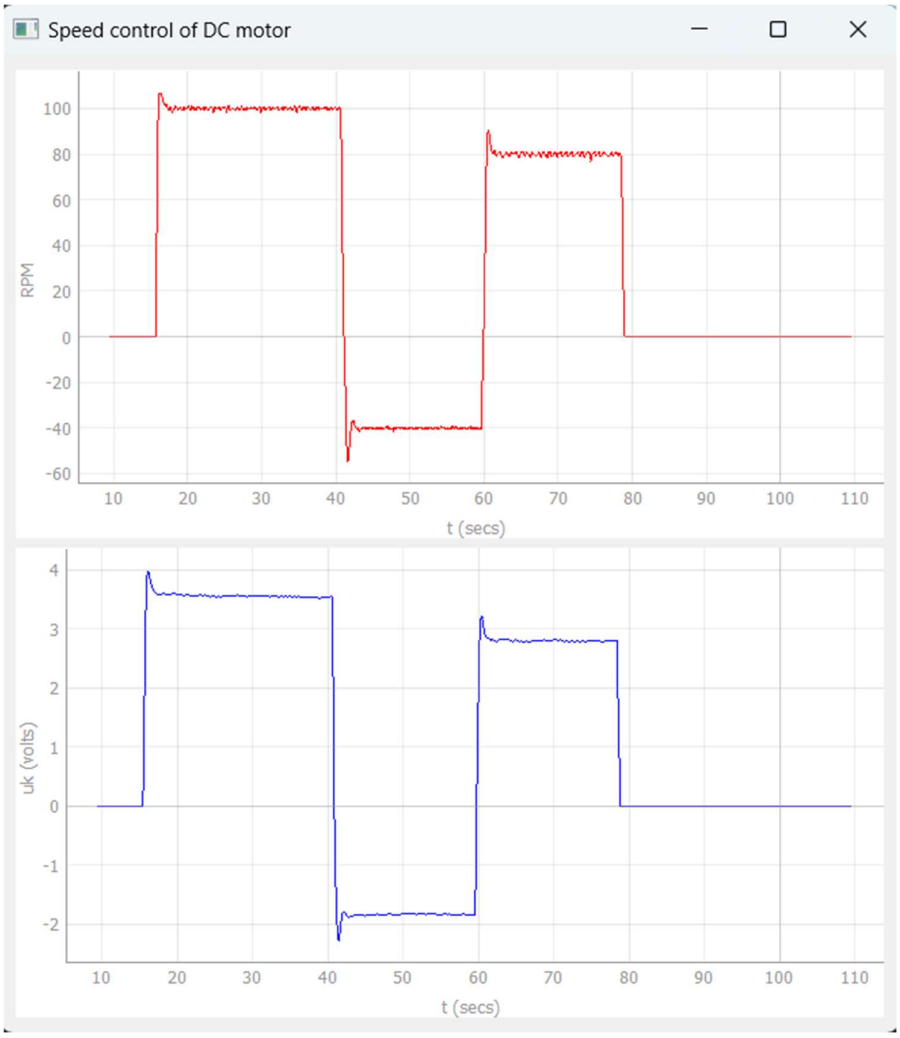

Figura 9. Gráficas de la velocidad de giro (superior) y la señal de control (inferior) generadas por *plot.py*.

5 Conclusiones

Este proyecto ha consistido en la implementación de un controlador para un motor DC en una Raspberry Pi con un sistema operativo real-time Linux. Esto permite realizar tareas en instantes concretos de tiempo, y permite implementar la periodicidad del periodo de muestreo que necesita un controlador. El proceso de diseño ha incluido todas las etapas de desarrollo que se precisan en un proyecto de estas características. Entre las tareas de este proceso, se incluye el diseño del *decoder*, la obtención de la función de transferencia discreta del controlador y su implementación computacional, la generación de la PWM y la señal de polaridad asociadas a la señal de control, que se encargan de conmutar los transistores del puente en H y, finalmente, la implementación de los protocolos de comunicación entre los diferentes componentes dedicados a la monitorización (gráfica) y cambio de consigna dinámico por parte del usuario.

El desarrollo de este proyecto ha constituido una primera toma de contacto con herramientas muy diversas que resultan de gran utilidad en el diseño de software para sistemas embebidos. Las principales a destacar son:

- Los hilos POSIX de UNIX (entre los cuales se destacan los hilos real-time).
- Distintos tipos de socket (sockets a bajo nivel de UNIX y sockets ZMQ de mayor nivel que permiten la comunicación con otras máquinas).
- Librerías para el acceso a los GPIOs de la Raspberry (*pigpio*).
- La herramienta *pyqtgraph*, que permite realizar aplicaciones con gráficas con un bajo coste computacional, lo cual permite la monitorización en tiempo real.

Finalmente, la verificación del control se ha realizado de manera elemental comprobando simplemente que el motor giraba a la velocidad que se le había asignado mediante el uso de un cronómetro y una muesca blanca sobre la rueda. También se ha comprobado que el sistema de monitorización gráfico y el programa para modificar la consigna funcionan adecuadamente.

6 Referencias

[1] K. Yaghmour, J. Masters, G. Ben-Yossef and P. Gerum, Building Embedded Linux Systems, edición 2. O'Reilly, 2008.

[2] Shayang Ye Industrial Co., IG220019X00015R datasheet, Diciembre 2013. https://digilent.com/reference/_media/motor_gearbox/290-006_ig220019x00015r_ds.pdf

[3] K. Basterretxea, Proiektua – Potentzia txikiko DC motor baten PI kontrola eta ibilgailu autonomoaren programazioa. Sistema Elektroniko Digitalak, Industria elektronikaren eta Automatikaren Ingeniaritzako Gradua, Euskal Herriko Unibertsitatea.

[4] Digilent, Unit 6: Analog I/O and Process Control. https://digilent.com/reference/learn/courses/unit-6/start

[5] Digilent, PmodHB5 2A H-Bridge Reference Manual, April 2016.

[6] Øyvind Nydal Dahl, What is an H-bridge? https://www.build-electronic-circuits.com/h-bridge/

[7] Raspberry Pi Documentation. https://www.raspberrypi.com/documentation/computers/raspberry-pi.html

[8] The Linux Foundation, HOWTO build a simple RT application. https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/application_base

[9] Pigpio C Interface. https://abyz.me.uk/rpi/pigpio/cif.html

[10] Educative.io, Unix socket programming in C. https://www.educative.io/answers/unix-socket-programming-in-c

[11] IBM, Example: Nonblocking I/O and select(). https://www.ibm.com/docs/en/i/7.3?topic=designs-example-nonblocking-io-select

[12] Infineon, Quadrature Decoder, November 2010. https://www.infineon.com/dgdl/Infineon-Quadrature_Decoder_(QuadDec)_Component_QuadDec_V1.50-Software%20Module%20Datasheets-v03_00-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e9659412050

[13] Python, Queue – A synchronized queue class. https://docs.python.org/es/3/library/queue.html

[14] RT-Linux, Ejemplos de uso de zeromq para mandar mensajes. https://nbviewer.org/github/josujugo/RT-Linux/blob/master/ejemplos_jupyter/uso_zmq.ipynb

[15] PythonGUIs, Plotting With PyQtGraph. https://www.pythonguis.com/tutorials/plotting-pyqtgraph/