ÍNDICE GENERAL

ÍNDICE DE FIGURAS

Capítulo 1

INTRODUCCIÓN

En este trabajo, durante varias semanas de experimentación con el microcontrolador ESP32 hemos tenido la oportunidad de aplicar y profundizar en diversos conceptos relacionados con el IoT.

Al inicio de este trabajo, comenzamos realizando pruebas con la librería uasyncio para gestionar tareas asincrónicas. Esto nos permitió aprender a trabajar con tareas simultáneas, sincronizadas y eventos.

Una vez familiarizados con el manejo de tareas, empezamos a experimentar con diferentes sensores. Primero, trabajamos con un sensor de temperatura y humedad, aunque terminaríamos por no usarlo en nuestra aplicación final. A continuación, nos centramos en el joystick, que fue el principal elemento de interacción en nuestro proyecto. Aprendimos a leer sus valores mediante los pines ADC del ESP32, lo que nos permitió mapear esos valores y utilizarlos para controlar dispositivos, como el LED RGB.

Tras estos primeros pasos, nos adentramos en los protocolos de comunicación. Comenzamos con ESP-NOW, donde exploramos cómo los dispositivos ESP32 pueden comunicarse entre sí sin la necesidad de infraestructura adicional como un router. Implementamos tanto el cliente como el servidor, y probamos cómo enviar los datos del joystick entre ambos dispositivos. También experimentamos con el protocolo WiFi y la comunicación mediante sockets, lo que nos permitió comparar ambas opciones.

Finalmente, reunimos todos estos elementos: los sensores, la librería asyncio, los protocolos de comunicación y el control del LED RGB. Logramos implementar un sistema donde los valores del joystick, leídos en tiempo real, controlaban el color del LED RGB.

Capítulo 2

ASYNCIO

Asyncio es una biblioteca para escribir código concurrente. asyncio se utiliza como base para una serie de frameworks asíncronos de Python que proporcionan un alto rendimiento, incluyendo web y servicios web, bibliotecas de conectividad de bases de datos, colas de tareas distribuidas, y mucho más.

En este proyecto, utilizamos asyncio para gestionar múltiples tareas simultáneamente, incluyendo:

1. Lectura de los datos de los ejes X e Y (entrada del sensor) del joystick.
2. Envío de datos mediante ESP-NOW o WiFi (comunicación de datos).
3. Asegurar la sincronización de la lectura y el envío de datos para evitar la pérdida de datos o el envío duplicado.

2.1 Tareas simultáneas

Queremos que el ESP32 realice varias tareas al mismo tiempo, por ejemplo:

- Tarea 1: Leer continuamente los datos del joystick (ejes X e Y).
- Tarea 2: Enviar datos al servidor o receptor (ESP-NOW o WiFi).

En la programación síncrona tradicional, una tarea termina antes de que empiece la otra, lo que puede provocar retrasos en los datos. Asyncio nos permite ejecutar múltiples tareas al mismo tiempo, de forma que las lecturas y transferencias de datos pueden hacerse en paralelo, aumentando la eficiencia.

Iniciamos ambas en la función main() para que el ESP32 pueda leer y enviar datos al mismo tiempo.

2.2 Tareas sincronizadas

Queremos ejecutarlas secuencialmente, de modo que una tarea se complete antes de que otra pueda comenzar. En nuestro trabajo, los datos del joystick se leen antes de que los datos puedan ser enviados.

Utilizamos await para asegurarnos de que una tarea se completa antes de que se ejecute la siguiente. En asyncio, await hace que el código espere a que la tarea se complete antes de continuar la ejecución sin afectar a las otras tareas.

En primer lugar

- Tarea 1: es responsable de leer los datos de los ejes X e Y para el joystick.
- Tarea 2: necesita esperar a que la tarea 1 se complete antes de enviar los datos al servidor.

Para terminar, Await permite que la lectura de datos esté completamente terminada antes de pasar a la tarea de envío de datos, asegurando la exactitud de los datos. De esta manera, no enviamos datos incorrectos o no actualizados.

2.3 Eventos

Las tareas deben esperar a que se cumplan determinadas condiciones antes de ejecutarse. Por ejemplo, enviar datos sólo cuando cambian los datos del joystick.

Usamos asyncio.Event() para hacer que las tareas se ejecuten sólo cuando son disparadas por un evento específico. Cree un objeto evento que no sea disparado por defecto. Cuando se leen los datos del joystick, data event.set() se dispara, indicando que hay nuevos datos para enviar. En lugar de ejecutar un bucle todo el tiempo, la tarea que envía los datos llama a await data_event.wait() y espera a

que los nuevos datos estén disponibles antes de enviarlos. Después de enviar los datos, llama a data event.clear() para esperar la siguiente actualización de datos.

Capítulo 3

SENSORES

3.1 Temperatura y humedad

En este caso se usó el sensor DHT11, el cual es una herramienta fundamental en el campo de la electrónica y la automatización debido a su capacidad para medir la temperatura y la humedad. Su popularidad radica en su simplicidad, bajo costo y precisión.

Este sensor es digital, lo que significa que su salida es una señal digital que puede ser fácilmente interpretada por el ESP32. Su rango de medición de temperatura va desde 0°C hasta 50°C con una precisión de ±2°C, mientras que su rango de medición de humedad varía entre el 20% y el 90% de humedad relativa, con una precisión de ±5%. Además, opera con un voltaje que oscila entre 3.5V y 5.5V, y tiene un consumo de corriente máximo de 2.5mA, lo cual lo hace ideal para aplicaciones de bajo consumo energético.

Para la conexión con el ESP32 se necesitan pocos materiales: el sensor DHT11, una placa ESP32 y cables de conexión. El sensor cuenta con tres pines: VCC, DATA y GND. Para conectarlo al ESP32, el pin VCC del DHT11 se conecta al pin de 3.3V del ESP32, el pin GND al pin GND del ESP32 y el pin DATA a uno de los pines digitales del ESP32, como por ejemplo el GPIO21.

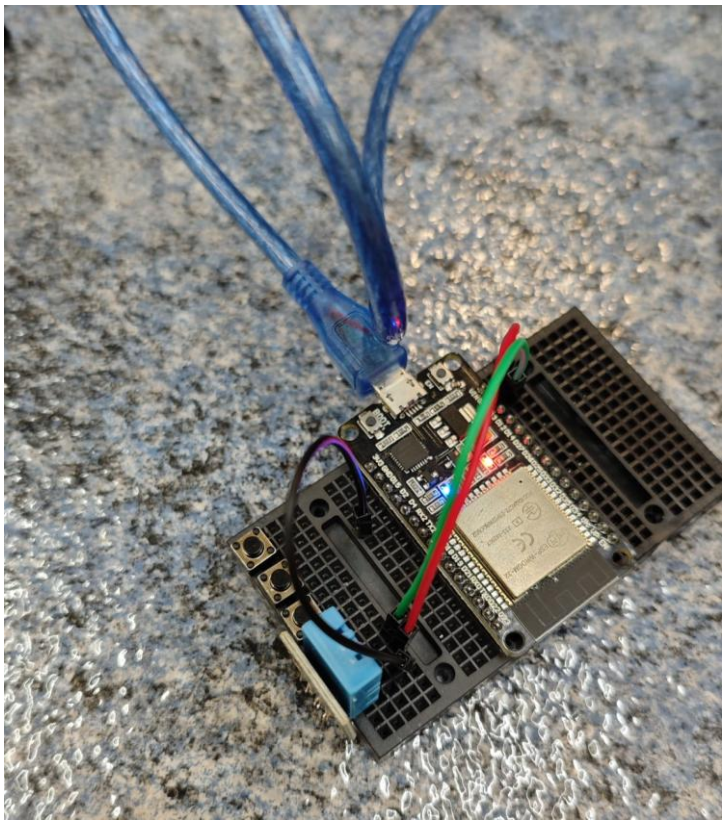

Figura 1: Sensor de temperatura y humedad

3.2 Joystick

El joystick analógico es un dispositivo de entrada esencial en la electrónica moderna, utilizado principalmente en aplicaciones de videojuegos y controladores de máquinas. Este instrumento permite detectar movimientos en dos ejes, conocidos como X e Y, y puede incluir también un botón integrado que amplía su funcionalidad, en este caso no fue usado, ya que solamente usamos los ejes X e Y. La conexión de un joystick analógico con un microcontrolador como el ESP32 se realiza mediante los pines ADC (Convertidor Analógico a Digital), lo que permite que el microcontrolador lea los valores de voltaje correspondientes a las posiciones del joystick y realice las acciones necesarias en consecuencia.

Las características principales del joystick analógico incluyen su capacidad para operar en un rango de voltaje de 0V a 5V y proporcionar salidas analógicas correspondientes a los movimientos en los ejes X e Y, en nuestro caso se usó entre 0V y 3.3V, ya que estos son los valores aceptados por el

ADC del ESP32. Adicionalmente, cuenta con una salida digital para el botón, que puede ser utilizada para detectar eventos de presión.

La conexión de este dispositivo con el ESP32 es bastante directa, requiriendo pocos componentes adicionales. El joystick posee cinco pines: GND, VCC, VRX, VRY y SW. Para establecer la conexión, se debe conectar el pin GND del joystick al pin GND del ESP32 y el pin VCC al pin de 3.3V del ESP32. Las salidas analógicas VRX y VRY se conectan a los pines ADC del ESP32.

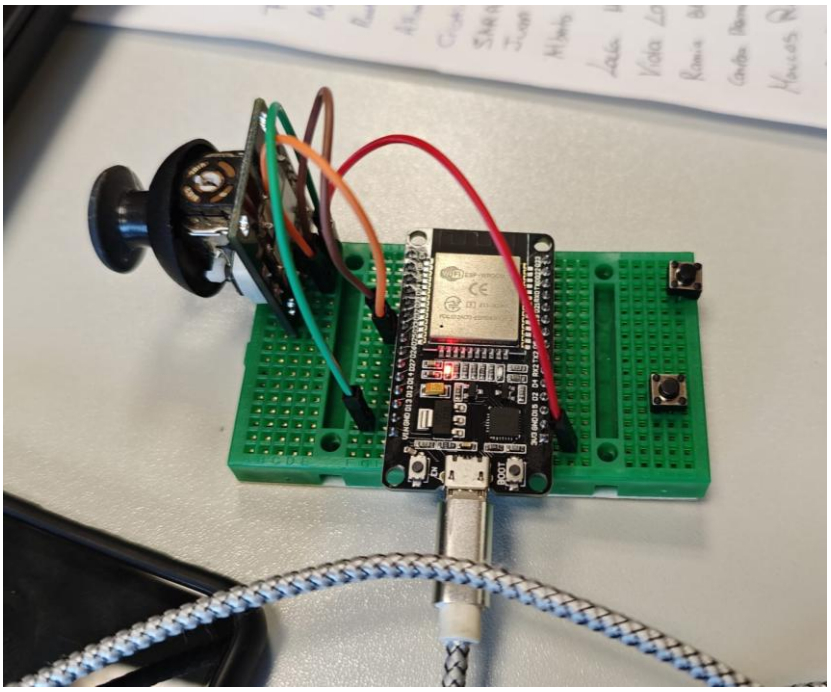

Figura 2: Joystick

3.3 LED

La conexión de un LED RGB con el ESP32 es relativamente sencilla. Generalmente se utilizan resistencias para limitar la corriente que pasa por cada LED, evitando así dañarlos, en nuestro caso no se usaron ya que la corriente de salida de los pines del microcontrolador es muy cercana a la corriente de funcionamiento del led. Esto permite que el microcontrolador controle cada color de manera independiente.

Para controlar la intensidad de los leds para crear los colores se utilizó PWM en el ESP32, esto se configuró en el microcontrolador para generar señales PWM en los pines GPIO correspondientes.

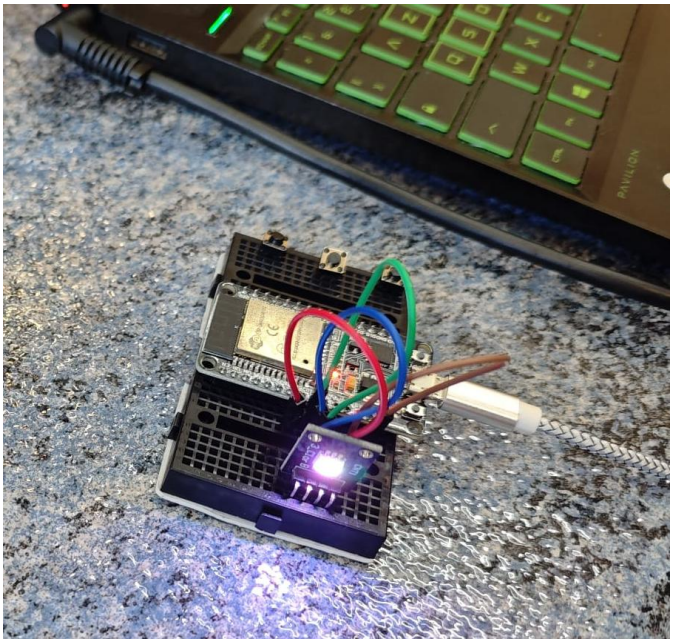

Figura 3: LED

3.4 ESP32

El ESP32 es un microcontrolador extremadamente versátil y ampliamente utilizado en aplicaciones del Internet de las Cosas debido a sus capacidades integradas de Wi-Fi y Bluetooth. Este dispositivo cuenta con un microprocesador dual-core Tensilica LX6, que puede operar a una frecuencia de reloj de hasta 240 MHz, lo que le permite manejar tareas complejas y realizar múltiples operaciones simultáneamente. Además, el ESP32 dispone de 520 KB de SRAM y 448 KB de memoria ROM, lo que le proporciona suficiente espacio para almacenar y ejecutar programas de diversa índole.

Una de las características más destacadas del ESP32 es su amplia gama de pines de entrada y salida de propósito general (GPIO), que suman 34 en total. Estos pines permiten la conexión de una variedad de sensores y actuadores, facilitando la creación de proyectos electrónicos interactivos. Asimismo, el ESP32 cuenta con 18 canales de convertidor analógico a digital (ADC) de 12 bits, lo que posibilita la lectura precisa de señales analógicas y su conversión a datos digitales procesables.

La conexión del ESP32 con sensores es relativamente sencilla y requiere pocos componentes adicionales. Los pines fundamentales para su funcionamiento son el pin GND, que se conecta a la tierra común del circuito, y el pin 3V3, que proporciona una fuente de alimentación de 3.3V. Los pines GPIO pueden utilizarse según las necesidades específicas del proyecto.

Capítulo 4

PROTOCOLOS DE COMUNICACIÓN

En este apartado se describen los protocolos de comunicación utilizados en el desarrollo del proyecto, destacando la implementación de la comunicación mediante Wi-Fi y ESP-Now para conectar dispositivos ESP32.

4.1 Wifi

La comunicación mediante Wi-Fi fue implementada utilizando los módulos integrados en la ESP32 para conectarse a una red local y establecer un canal de comunicación cliente-servidor. El protocolo TCP/IP se utilizó para garantizar una comunicación confiable entre los dispositivos.

4.1.1 Implementación del Servidor

El servidor, configurado en una ESP32, establece una conexión a la red Wi-Fi y escucha solicitudes en el puerto 8080. Además, se define una IP específica permitida para garantizar que solo el cliente autorizado pueda enviar datos al servidor. El servidor se mantiene en un bucle esperando conexiones entrantes y recibe los datos transmitidos por el cliente.

```
import network
import time
import socket
# Conexión a la red Wi-Fi
sta = network.WLAN(network.STA_IF)
sta.active(True)
sta.connect("IOTNET_2.4", "10T@ATC_")
# Esperar hasta que se conecte a Wi-Fi
while not sta.isconnected():
```

```
    print("Intentando conectar a la red...")
    time.sleep(1)
print("Conectado a la red con IP:", sta.ifconfig()[0])
# IP específica desde la que permitiremos conexiones
ip_permitida = "192.168.2.29"  # Cambia esta IP por la IP de la ESP32
cliente permitida
# Crear y configurar el servidor TCP/IP
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((sta.ifconfig()[0], 8080))  # Escuchar en la IP asignada de la ESP32
servidor y el puerto 8080
s.listen(1)
print("Esperando conexión en la IP:", sta.ifconfig()[0])
conn, addr = s.accept()

while True:
    # Verificar si la IP de la conexión entrante es la permitida
    if addr[0] == ip_permitida:
        # Recibir datos
        data = conn.recv(1024)
        print("Mensaje recibido:", data.decode())
    else:
        print("Conexión rechazada desde:", addr[0])
        conn.close()  # Cerrar conexión si no coincide con la IP permitida
```

4.1.2 Implementación del Cliente

El cliente, también configurado en una ESP32, se conecta a la misma red Wi-Fi y establece una comunicación con el servidor mediante el protocolo TCP/IP. Además, se utilizó un sensor DHT11 para obtener datos de temperatura y humedad que posteriormente son enviados al servidor en intervalos regulares.

```
import machine
import dht
import time
import network
import socket
# Configurar la ESP32 como cliente
sta = network.WLAN(network.STA_IF)
sta.active(True)
sta.connect("IOTNET_2.4", "10T@ATC_")
```

```
# Esperar hasta que se conecte
while not sta.isconnected():
    print("Intentando conectar a la red...")
    time.sleep(1)
print("Conectado a la red con IP:", sta.ifconfig()[0])
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('192.168.2.25', 8080))  # Cambia 'IP_del_Servidor' por la IP de
la ESP32 servidor
# Configurar el pin del sensor (cambiar el número de pin según tu conexión)
pin_sensor = machine.Pin(2)  # Aquí usamos el pin GPIO 2
sensor = dht.DHT11(pin_sensor)  # Crear una instancia del sensor DHT11
while True:
    try:
        # Leer el sensor
        sensor.measure()
        # Obtener la temperatura y la humedad
        temperatura = sensor.temperature()  # Obtener temperatura en °C
        humedad = sensor.humidity()  # Obtener humedad en %
        # Enviar los resultados al servidor
        s.send("Temperatura: " + str(temperatura) + " °C" + "\n" +
"Humedad: " + str(humedad) + "%")
    except OSError as e:
        print("Error al leer el sensor:", e)
    # Esperar 2 segundos antes de la próxima lectura
    time.sleep(1)
s.close()
```

4.2 ESP-Now

ESP-Now permite que un dispositivo actúe como servidor (receptor) y otro como cliente (emisor). En este caso, configuraremos un **servidor** para recibir mensajes y un **cliente** para enviar mensajes. Este modelo puede ser útil, por ejemplo, en aplicaciones IoT para recolectar datos desde varios sensores (clientes) hacia un nodo central (servidor).

4.2.1 Implementación del Servidor

El servidor estará escuchando mensajes de los clientes y procesará los datos recibidos. Su funcionamiento es el siguiente:

1. Se configura el servidor en modo estación y activa ESP-Now.
2. Se utiliza el método e.recv() para escuchar mensajes enviados por los clientes.
3. Al recibir un mensaje, imprime el contenido y la dirección MAC del cliente que lo envió.

```
import network
import espnow
# Configurar la interfaz WLAN en modo estación
sta = network.WLAN(network.STA_IF)
sta.active(True)
# Inicializar ESPNow
e = espnow.ESPNow()
e.active(True)
# Escuchar mensajes de clientes
print("Servidor en ESP-Now listo, esperando mensajes...")

while True:
    host, msg = e.recv()  # host es la MAC del emisor, msg es el mensaje recibido
    if msg:
        print(f"Mensaje recibido de {host}: {msg}")
```

4.2.2 Implementación del Cliente

El cliente se encargará de enviar mensajes al servidor utilizando ESP-Now.

1. Se configura el cliente en modo estación y activa ESP-Now.
2. Se agrega al servidor como "peer" utilizando su dirección MAC.
3. Utilizamos el método e.send() para enviar mensajes al servidor.

```
import network
import espnow
# Configurar la interfaz WLAN en modo estación
```

```
sta = network.WLAN(network.STA_IF)
sta.active(True)
# Inicializar ESPNow
e = espnow.ESPNow()
e.active(True)
# Agregar un peer (cambiar la dirección MAC por la del servidor)
server_mac = b'\xe0\x5a\x1b\x5f\xd6\x5c'  # Dirección MAC del servidor
e.add_peer(server_mac)
# Enviar un mensaje al servidor
while True:
    try:
        e.send(server_mac, "Mensaje desde el cliente ESP-Now")
        print("Mensaje enviado al servidor.")
    except Exception as e:
        print("Error al enviar mensaje:", e)
```

Capítulo 5

APLICACIÓN

Como aplicación final de todos los conocimientos adquiridos a lo largo de las sesiones de experimentación hemos decidido controlar el valor de un LED según la posición de un Joystick. A parte, lo hemos implementado con ambos protocolos de comunicación.

Se puede ver el funcionamiento del sistema en vídeo en el siguiente enlace: https://youtu.be/nUrDl4bbxNs.

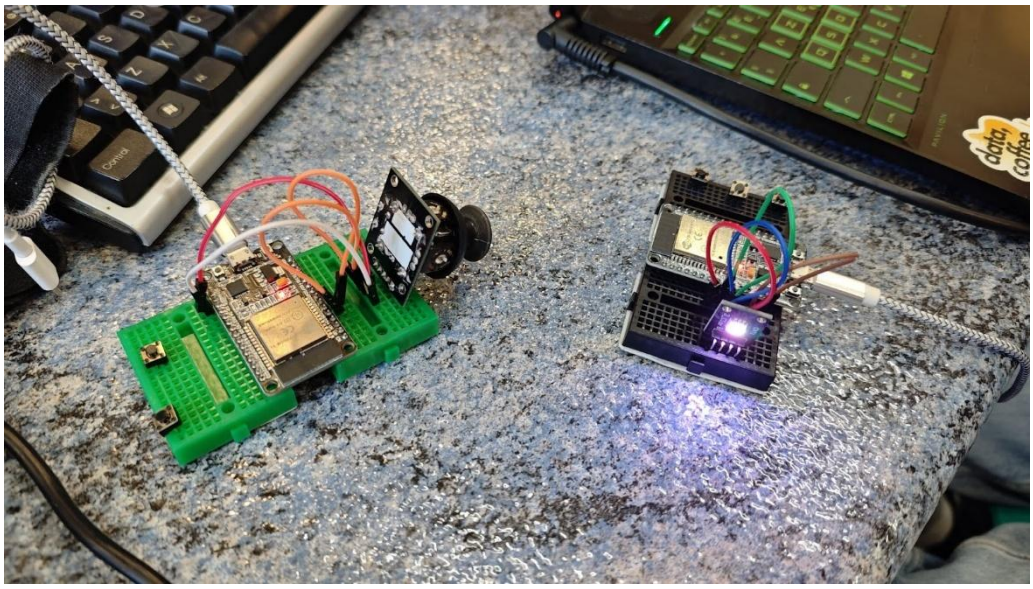

Figura 4: Joystick controlando LED

5.1 Cliente

5.1.1 Cliente usando ESPNow

Tomaremos como cliente la placa con el Joystick.

En esta aplicación, configuramos el cliente para capturar los datos del joystick y enviarlos al servidor. Primero, utilizamos los pines ADC del microcontrolador para leer las posiciones del joystick en los ejes X e Y. Mediante verificar_esquina, traducimos los valores obtenidos del joystick en una posición como: arriba, derecha, etc.

Para habilitar la comunicación ESPNow, inicializamos el módulo mediante e = espnow.ESPNow() y lo activamos con e.active(True). A continuación, configuramos el servidor como receptor de los datos añadiendo su dirección MAC con e.add_peer(peer), donde peer corresponde a la dirección del dispositivo receptor, especificada en formato binario.

El envío de datos se realiza en mensajes compactos utilizando e.send(peer, mensaje), donde mensaje contiene los valores actuales del joystick. Este mecanismo asegura que los datos sean transmitidos de manera eficiente y sin necesidad de infraestructura adicional.

Además, implementamos dos tareas asincrónicas: una para leer continuamente el estado del joystick y otra para enviar estos datos al servidor. Esto permite mantener una comunicación en tiempo real y gestionar de forma ordenada las operaciones del cliente.

```
import uasyncio as asyncio
from machine import ADC, Pin
import network
import espnow

# Configurar la interfaz WLAN en modo estación
sta = network.WLAN(network.STA_IF)
sta.active(True)

# Inicializar ESPNow
e = espnow.ESPNow()
e.active(True)

# Agregar un peer (cambia la MAC por la de tu dispositivo receptor)
peer = b'\xe0\x5a\x1b\x5f\xd6\x5c'  # MAC address en formato binario
e.add_peer(peer)

# Configurar los pines ADC para leer los valores del joystick
```

```
Verti_y = ADC(Pin(33))
Verti_y.atten(ADC.ATTN_11DB)

ps2_x = ADC(Pin(32))
ps2_x.atten(ADC.ATTN_11DB)

# Variables compartidas y evento para la sincronización
joystick_data = {"x": 0, "y": 0}
data_event = asyncio.Event()

# Función para verificar las esquinas
def verificar_esquina(x, y):
    if x > 4000 and y > 4000:
        return "Arriba-Izquierda"
    elif x < 100 and y < 100:
        return "Abajo-Derecha"
    elif x > 4000 and y < 100:
        return "Abajo-Izquierda"
    elif x < 100 and y > 4000:
        return "Arriba-Derecha"
    elif y < 100 :
        return "Abajo"
    elif y  > 4000 :
        return "Arriba"
    elif x < 100 :
        return "Derecha"
    elif x  > 4000 :
        return "Izquierda"
    else:
        return "Centro o en movimiento"

# Tarea para leer el joystick
async def leer_joystick():
    global joystick_data
    while True:
        rec_x = ps2_x.read()
        rec_y = Verti_y.read()
        joystick_data["x"] = rec_x
        joystick_data["y"] = rec_y
        print(f"Joystick -> x: {rec_x}, y: {rec_y}")
        print(verificar_esquina(rec_x, rec_y))
```

```
        # Disparar el evento indicando que hay nuevos datos
        data_event.set()
        await asyncio.sleep(0.1)  # Leer cada 100 ms

# Tarea para enviar datos por ESP-NOW
async def enviar_datos():
    while True:
        # Esperar hasta que haya datos nuevos
        await data_event.wait()
        data_event.clear()  # Limpiar el evento después de procesarlo

        # Enviar los datos por ESP-NOW

        mensaje = "x:{} y:{}".format(joystick_data["x"],
joystick_data["y"])
        #mensaje = verificar_esquina(rec_x, rec_y)
        e.send(peer, mensaje)
        #print(f"Enviado -> {mensaje}")

        await asyncio.sleep(0.1)  # Intervalo opcional

# Tarea principal
async def main():
    e.send(peer, "Starting...")  # Mensaje inicial
    print("Comenzando tareas...")

    # Crear tareas asincrónicas
    tarea_joystick = asyncio.create_task(leer_joystick())
    tarea_envio = asyncio.create_task(enviar_datos())

    # Mantener la ejecución
    await asyncio.gather(tarea_joystick, tarea_envio)


# Ejecutar el bucle principal de asyncio
try:
    asyncio.run(main())
except KeyboardInterrupt:
    e.send(peer, b'end')  # Envía el mensaje final
    e.close()
```

```
    print("Programa terminado.")
```

5.1.2 Cliente usando Wifi:

En este caso, configuramos el cliente para transmitir los datos del joystick al servidor mediante una conexión WiFi. El cliente inicia activando la interfaz WiFi mediante sta = network.WLAN(network.STA_IF) y sta.active(True). A continuación, se conecta a la red especificada con sta.connect("IOTNET_2.4", "10T@ATC_").

Una vez establecida la conexión, se implementa un sistema basado en sockets para enviar los datos al servidor. Este socket se configura con e = socket.socket(socket.AF_INET, socket.SOCK_STREAM) y se conecta al servidor utilizando su dirección IP y puerto con e.connect(('192.168.2.26', 8080)).

Todo el tratamiento de datos del joystick es el mismo que en el caso de ESPNow.

```
import uasyncio as asyncio
from machine import ADC, Pin
import network
import socket
import time


sta = network.WLAN(network.STA_IF)
sta.active(True)
sta.connect("IOTNET_2.4", "10T@ATC_")


# Esperar hasta que se conecte
while not sta.isconnected():
    print("Intentando conectar a la red...")
    time.sleep(1)


print("Conectado a la red con IP:", sta.ifconfig()[0])
```

```
e = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
e.connect(('192.168.2.26', 8080))


# Configurar los pines ADC para leer los valores del joystick
Verti_y = ADC(Pin(33))
Verti_y.atten(ADC.ATTN_11DB)


ps2_x = ADC(Pin(32))
ps2_x.atten(ADC.ATTN_11DB)


# Variables compartidas y evento para la sincronización
joystick_data = {"x": 0, "y": 0}
data_event = asyncio.Event()


# Función para verificar las esquinas
def verificar_esquina(x, y):
    if x > 4000 and y > 4000:
        return "Arriba-Izquierda"
    elif x < 100 and y < 100:
        return "Abajo-Derecha"
    elif x > 4000 and y < 100:
        return "Abajo-Izquierda"
    elif x < 100 and y > 4000:
        return "Arriba-Derecha"
    elif y < 100 :
        return "Abajo"
    elif y  > 4000 :
        return "Arriba"
    elif x < 100 :
        return "Derecha"
    elif x  > 4000 :
        return "Izquierda"
    else:
        return "Centro o en movimiento"
# Tarea para leer el joystick
async def leer_joystick():
    global joystick_data
```

```
    while True:
        rec_x = ps2_x.read()
        rec_y = Verti_y.read()
        joystick_data["x"] = rec_x
        joystick_data["y"] = rec_y
        print(f"Joystick -> x: {rec_x}, y: {rec_y}")
        print(verificar_esquina(rec_x, rec_y))

        # Disparar el evento indicando que hay nuevos datos
        data_event.set()
        await asyncio.sleep(1)  # Leer cada 100 ms


async def enviar_datos():
    while True:
        # Esperar hasta que haya datos nuevos
        await data_event.wait()
        data_event.clear()  # Limpiar el evento después de procesarlo

        mensaje = "x:{} y:{}".format(joystick_data["x"],
joystick_data["y"])
        #mensaje = verificar_esquina(rec_x, rec_y)
        e.send(mensaje)
        #print(f"Enviado -> {mensaje}")

        await asyncio.sleep(1)  # Intervalo opcional

# Tarea principal
async def main():
    e.send( "Starting...")  # Mensaje inicial
    print("Comenzando tareas...")

    # Crear tareas asincrónicas
    tarea_joystick = asyncio.create_task(leer_joystick())
    tarea_envio = asyncio.create_task(enviar_datos())

    # Mantener la ejecución
    await asyncio.gather(tarea_joystick, tarea_envio)


# Ejecutar el bucle principal de asyncio
```

```
try:
    asyncio.run(main())
except KeyboardInterrupt:
    e.send( b'end')  # Envía el mensaje final
    e.close()
    print("Programa terminado.")
```

5.2 Server

En esta sección, configuramos el servidor para recibir datos enviados por el cliente y utilizarlos para controlar el LED.

5.2.1 Servidor usando ESPNow

El servidor procesa los datos recibidos y ajusta el color del LED RGB en función de los valores de X e Y recibidos del cliente. Los pines del LED están controlados mediante PWM (Modulación por Ancho de Pulso) para ajustar la intensidad de cada color (Rojo, Verde y Azul) de acuerdo con la posición del joystick

En la función actualizar_rgb, utilizamos map_value para calcular las intensidades de los tres colores del LED RGB. El valor de x, que proviene del eje horizontal del joystick, controla el color rojo; el valor de y, que proviene del eje vertical, controla el color azul; y la media de ambos valores se usa para ajustar el verde. De esta forma, podemos cambiar el color del LED dependiendo de cómo movemos el joystick.

```
import network
import espnow
from machine import Pin, PWM


# Configurar los pines PWM para el LED RGB
led_r = PWM(Pin(25))  # Rojo
led_g = PWM(Pin(26))  # Verde
led_b = PWM(Pin(27))  # Azul
```

```
# Ajustar la frecuencia del PWM
led_r.freq(500)
led_g.freq(500)
led_b.freq(500)


# Función para mapear un valor de un rango a otro
def map_value(value, in_min, in_max, out_min, out_max):
    return int((value - in_min) * (out_max - out_min) / (in_max - in_min) +
out_min)


# Función para actualizar el color del LED RGB según x e y
def actualizar_rgb(x, y):
    # Mapear los valores de x e y a un rango de 0 a 1023 (intensidad de
PWM)
    r = map_value(x, 0, 4095, 0, 1023)  # Rojo depende de x
    b = map_value(y, 0, 4095, 0, 1023)  # Azul depende de y
    g = map_value((x + y) // 2, 0, 4095, 0, 1023)  # Verde depende de la
mezcla de x e y


    # Ajustar los colores del LED
    led_r.duty(r)
    led_g.duty(g)
    led_b.duty(b)


    print(f"RGB -> R: {r}, G: {g}, B: {b}")


# Configurar la interfaz WLAN en modo estación
sta = network.WLAN(network.STA_IF)
sta.active(True)
sta.disconnect()  # Desconectar de cualquier punto de acceso previo


# Inicializar ESPNow
e = espnow.ESPNow()
e.active(True)
```

```
print("Esperando mensajes...")


# Bucle principal para recibir datos y controlar el LED RGB
while True:
    host, msg = e.recv()  # Recibir datos
    if msg:  # Procesar el mensaje solo si no es `None`
        print(f"Mensaje recibido de {host}: {msg}")
        if msg == b'end':  # Detener el programa si se recibe "end"
            print("Finalizando programa...")
            break

        # Decodificar el mensaje y extraer los valores de x e y
        try:
            data = msg.decode('utf-8')  # Decodificar el mensaje
            if data.startswith("x:") and "y:" in data:
                # Parsear los valores de x e y
                valores = data.replace("x:", "").replace("y:", "").split()
                x = int(valores[0])
                y = int(valores[1])


                # Actualizar el color del LED RGB
                actualizar_rgb(x, y)
        except Exception as ex:
            print(f"Error al procesar el mensaje: {ex}")
```

5.2.2 Servidor usando Wifi

Por último, al igual que con el cliente, tenemos la misma implementación, pero utilizando sockets.

```
import network
import socket
from machine import Pin, PWM


# Conexión a la red Wi-Fi
```

```
sta = network.WLAN(network.STA_IF)
sta.active(True)
sta.connect("IOTNET_2.4", "10T@ATC_")


# Esperar hasta que se conecte a Wi-Fi
while not sta.isconnected():
    print("Intentando conectar a la red...")
    time.sleep(1)


print("Conectado a la red con IP:", sta.ifconfig()[0])


# IP específica desde la que permitiremos conexiones
ip_permitida = "192.168.2.29"  # Cambia esta IP por la IP de la ESP32
cliente permitida


# Crear y configurar el servidor TCP/IP


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((sta.ifconfig()[0], 8080))  # Escuchar en la IP asignada de la ESP32
servidor y el puerto 8080
s.listen(1)


print("Esperando conexión en la IP:", sta.ifconfig()[0])
conn, addr = s.accept()
while True:


    # Verificar si la IP de la conexión entrante es la permitida


        # Recibir datos
        data = conn.recv(1024)
        print("Mensaje recibido:", data.decode())

        #conn.close()  # Cerrar la conexión después de recibir el mensaje
```

```
        #break  # Salir después de recibir el primer mensaje


# Configurar los pines PWM para el LED RGB
led_r = PWM(Pin(25))  # Rojo
led_g = PWM(Pin(26))  # Verde
led_b = PWM(Pin(27))  # Azul


# Ajustar la frecuencia del PWM
led_r.freq(500)
led_g.freq(500)
led_b.freq(500)


# Función para mapear un valor de un rango a otro
def map_value(value, in_min, in_max, out_min, out_max):
    return int((value - in_min) * (out_max - out_min) / (in_max - in_min) +
out_min)


# Función para actualizar el color del LED RGB según x e y
def actualizar_rgb(x, y):
    # Mapear los valores de x e y a un rango de 0 a 1023 (intensidad de
PWM)
    r = map_value(x, 0, 4095, 0, 1023)  # Rojo depende de x
    b = map_value(y, 0, 4095, 0, 1023)  # Azul depende de y
    g = map_value((x + y) // 2, 0, 4095, 0, 1023)  # Verde depende de la
mezcla de x e y


    # Ajustar los colores del LED
    led_r.duty(r)
    led_g.duty(g)
    led_b.duty(b)


    print(f"RGB -> R: {r}, G: {g}, B: {b}")
```

```
print("Esperando mensajes...")


# Bucle principal para recibir datos y controlar el LED RGB
while True:
    if addr[0] == ip_permitida:
        msg = conn.recv(1024)
        print("Mensaje recibido:", msg.decode())


            # Decodificar el mensaje y extraer los valores de x e y
        try:
            data = msg.decode('utf-8')  # Decodificar el mensaje
            if data.startswith("x:") and "y:" in data:
                    # Parsear los valores de x e y
                valores = data.replace("x:", "").replace("y:", "").split()
                x = int(valores[0])
                y = int(valores[1])


                    # Actualizar el color del LED RGB
                actualizar_rgb(x, y)
        except Exception as ex:
            print(f"Error al procesar el mensaje: {ex}")
    else:
        print("Conexión rechazada desde:", addr[0])
        conn.close()  # Cerrar conexión si no coincide con la IP permitida
```

BIBLIOGRAFÍA

[1] Joy-IT. (2017). KY-023 Joystick module (XY-Axis). https://naylampmechatronics.com/img/cms/Datasheets/000036%20-%20datasheet%20KY-023-Joy-IT.pdf

[2] ESPIoradores. (2020) .MICROPYTHON ESP32 – Conversión Analógica-Digital ADC (Analog-Digital Conversion). https://www.esploradores.com/micropython_adc/ https://docs.micropython.org/en/latest/esp32/quickref.html

[3] Damien P. George, Paul Sokolovsky, and contributors. (2021). uasyncio — asynchronous I/O scheduler. https://docs.micropython.org/en/v1.14/library/uasyncio.html

[4] Python Software Foundation. (s.f.). aasyncio synchronization primitives https://docs.python.org/3/library/asyncio-sync.html

[5] Peter Hinch. (s.f.). uasyncio tutorial for MicroPython https://github.com/peterhinch/micropython-async/blob/master/v3/docs/TUTORIAL.md#2-asyncio-concept

[6] MicroPython. (2021). espnow — support for the ESP-NOW wireless protocol. https://docs.micropython.org/en/latest/library/espnow.html

[7] MicroPython. (2021). 5. Network - TCP sockets. https://docs.micropython.org/en/latest/esp8266/tutorial/network_tcp.html