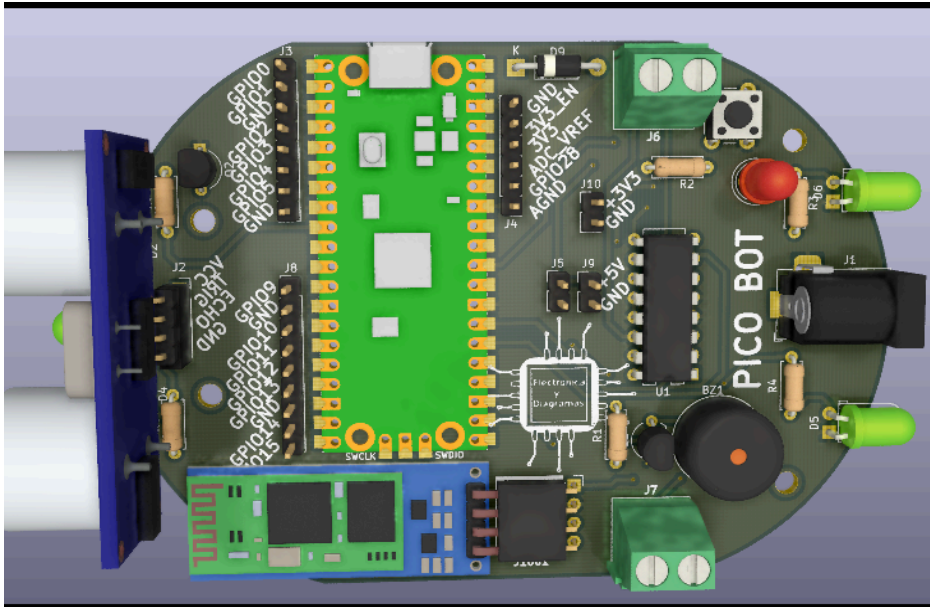


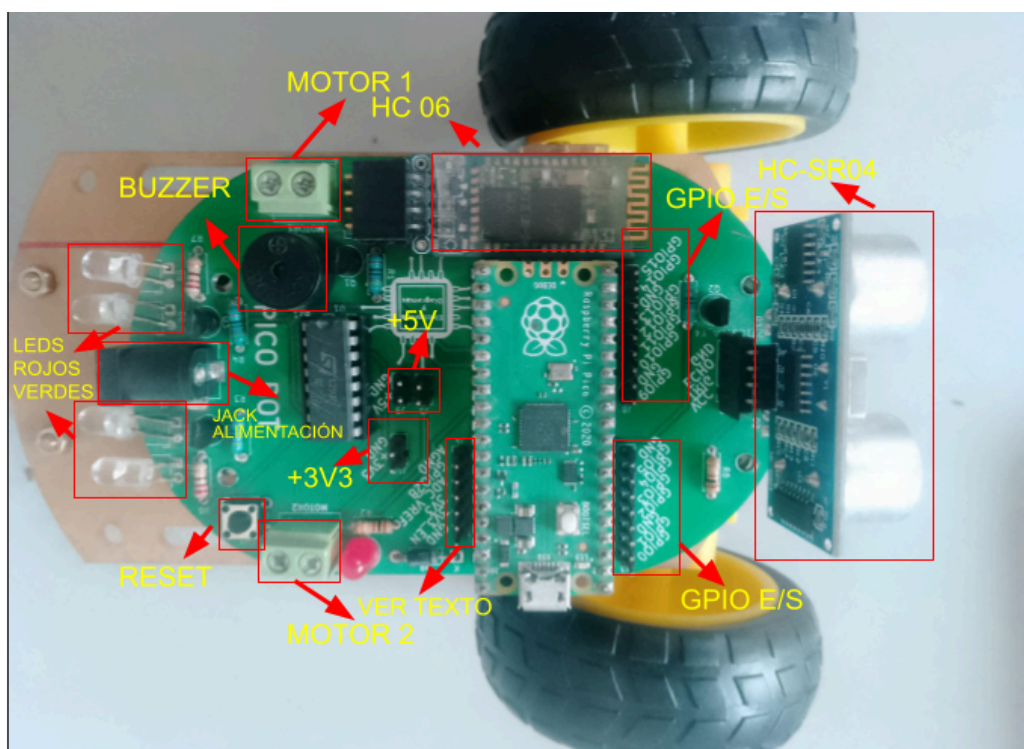
Guía de programación PICO BOT



Características del PICO BOT

El **PICO BOT** es una plataforma robótica educativa basada en la placa de desarrollo **Raspberry Pi Pico**. Está diseñado para facilitar el aprendizaje de electrónica, programación y robótica. Entre sus principales características se incluyen:

- **Leds de colores:** Incluye LEDs **verdes, rojos y blancos**, ideales para señalización, indicación de estados o para prácticas de programación básica.
- **Motores con engranajes:** Equipado con **motores reductores** que permiten un movimiento preciso y controlado del robot.
- **Buzzer pasivo:** Permite la generación de sonidos, tonos y melodías programables.
- **GPIO expuestos:** Acceso a los pines de entrada/salida del microcontrolador, útiles para conectar sensores, actuadores u otros módulos externos.
- **Conectores para módulos:** Entradas y salidas adicionales que permiten la expansión del sistema con módulos como sensores ultrasónicos, infrarrojos, pantallas, etc.
- **Compatibilidad con Raspberry Pi Pico:** Funciona con la popular placa **Raspberry Pi Pico** y su microcontrolador **RP 2040**, compatible con lenguajes como Micro Python y C/C++.
- **Salidas de alimentación de 3.3V y 5V:** Facilita la alimentación de diferentes tipos de módulos o periféricos.
- **Comunicación Bluetooth:** Integra un módulo Bluetooth hc 05



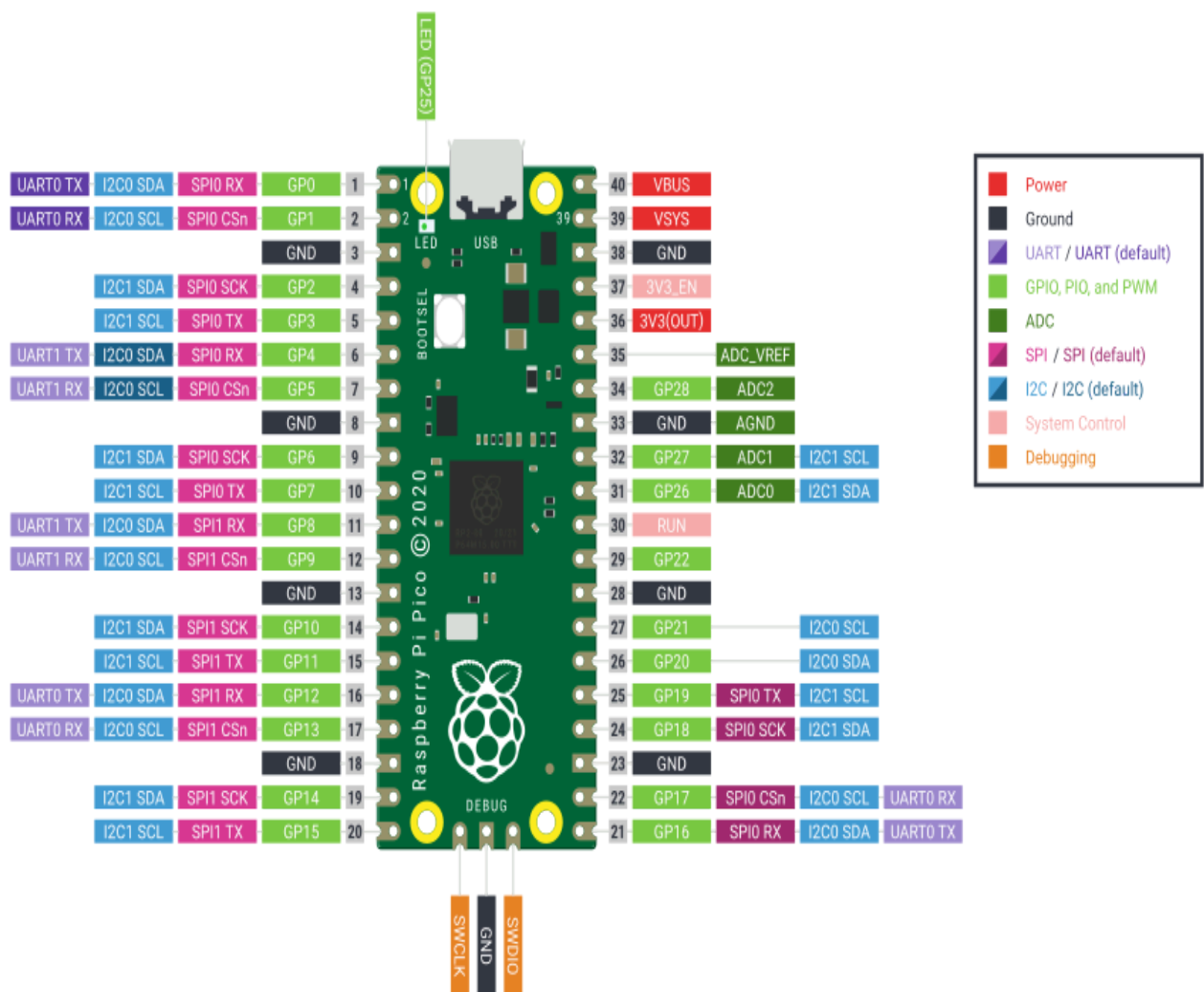
Pines clave de la Raspberry Pi Pico

La **Raspberry Pi Pico** es una placa de desarrollo basada en el microcontrolador **RP 2040**, que incluye un procesador **ARM Cortex-M0+ de doble núcleo**, 2 MB de memoria flash y múltiples interfaces periféricas (ADC, I2C, SPI, UART, PWM). Es ideal para proyectos educativos y de electrónica, y es compatible con **MicroPython** y C++.

🔌 Pines de alimentación y control

Pin	Descripción	Funcionamiento/Uso
3V3_EN (Pin 36)	Pin de control que habilita o desactiva el regulador interno de 3.3 V de la Pico	Si se conecta a GND, desactiva el regulador y apaga la placa. Ideal para sistemas con control total de energía.
3V3 (Pin 35)	Salida del regulador de voltaje de 3.3 V. Alimenta la lógica interna y circuitos externos.	Alimenta componentes que trabajen a 3.3 V. Voltaje típico: 3.3 V Corriente máx.: ~300 mA (depende de la fuente de Entrada)

ADC VREF (Pin 33)	Referencia de voltaje para el ADC.	Por defecto conectada a 3.3 V, pero puede usarse una fuente externa para mayor precisión.
AGND (Pin 30)	Tierra analógica.	Proporciona referencia limpia para señales sensibles del ADC, evitando interferencias digitales.



GPIO utilizados en el PICO BOT

La siguiente tabla detalla los pines GPIO utilizados por los distintos componentes del **PICO BOT**:

Indicadores LED

- **LEDs blancos:** GPIO 6
- **LEDs verdes:** GPIO 27
- **LEDs rojos:** GPIO 26

Sonido

- **Buzzer pasivo:** GPIO 22

Comunicación inalámbrica

- **Módulo Bluetooth HC-06:**
 - **TX:** GPIO 16
 - **RX:** GPIO 17

Sensor de distancia ultrasónico (HC-SR04)

- **TRIG:** GPIO 7
- **ECHO:** GPIO 8

Motores

- **Motor A:**
 - **a1:** GPIO 18
 - **a2:** GPIO 19
 - **Motor B:**
 - **b1:** GPIO 20
 - **b2:** GPIO 21
-

Funcionalidades de los GPIO en la Raspberry Pi Pico

Los pines GPIO del microcontrolador **RP 2040** de la Raspberry Pi Pico permiten una amplia variedad de funciones:

Digital I/O

- Todos los GPIO pueden configurarse como entradas o salidas digitales.

PWM (Pulse Width Modulation)

- Todos los pines GPIO son capaces de generar señales PWM.

- La Pico cuenta con **16 canales de PWM**, que pueden asignarse dinámicamente a los pines disponibles.

ADC (Convertidor Analógico a Digital)

- Pines con funcionalidad ADC:
 - **GPIO 26** → ADC0
 - **GPIO 27** → ADC1
 - **GPIO 28** → ADC2
- Rango de voltaje: **0 V a 3.3V**
- Resolución: **12 bits**

UART (Comunicación serie)

- Se pueden configurar hasta **dos UARTs** (UART0 y UART1).
- Ejemplo típico:
 - **TX:** GPIO 0
 - **RX:** GPIO 1

I2C (Inter-Integrated Circuit)

- Soporta **dos interfaces I2C** (I2C0 e I2C1).
- Ejemplo:
 - **SDA:** GPIO 4
 - **SCL:** GPIO 5

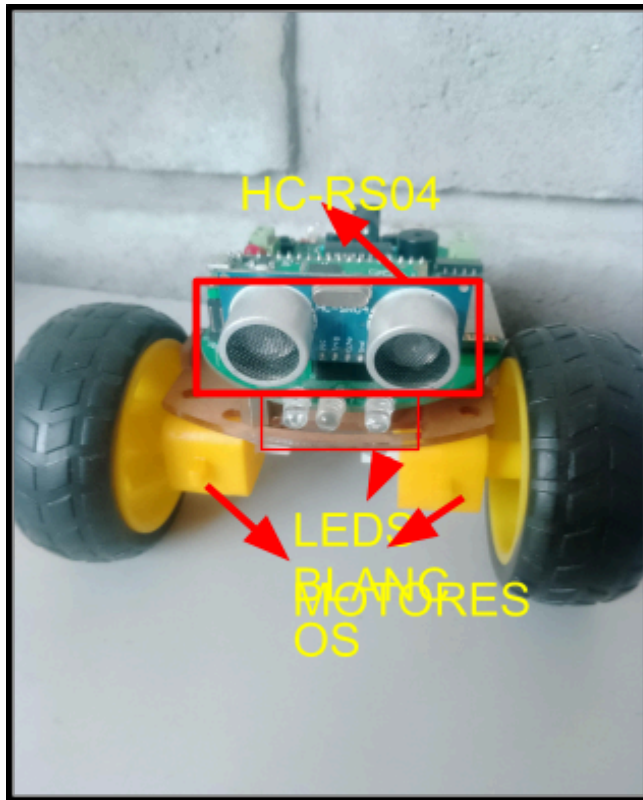
SPI (Serial Peripheral Interface)

- Se pueden configurar **hasta dos buses SPI** (SPI0 y SPI1).
- Ejemplo:
 - **SCK:** GPIO 18
 - **TX:** GPIO 19
 - **RX:** GPIO 16

Interrupciones

- Todos los GPIO soportan **interrupciones configurables**:
 - En flanco de subida
 - En flanco de bajada
 - En ambos flancos

HC-RS04:



Sensor Ultrasónico HC-SR04

El **HC-SR04** es un sensor ultrasónico muy popular para **medir distancias**. Su bajo costo, simplicidad y precisión lo convierten en una excelente opción para proyectos de **robótica**, **automatización** y **electrónica educativa**.

Características principales:

- **Método de medición:** Basado en el tiempo que tarda un pulso ultrasónico en reflejarse desde un objeto y regresar al sensor.
 - **Rango de medición:** 2 cm a 400 cm.
 - **Precisión:** ± 3 mm.
 - **Frecuencia de operación:** 40 kHz.
 - **Voltaje de operación:** 5V DC.
 - **Corriente típica:** ~ 15 mA.
- Interfaz:** 2 pines de control: TRIG (entrada) y ECHO (salida).

Principio de funcionamiento:

1. **Disparo del pulso:**
 - Se envía una señal de **10 microsegundos (μ s)** al pin **TRIG**.
 - El sensor emite **8 pulsos ultrasónicos** a 40 kHz a través del emisor.
2. **Recepción del eco:**
 - Si un objeto refleja el sonido, el **pin ECHO** genera un pulso cuya duración es proporcional al tiempo de ida y vuelta del sonido.

3. Cálculo de la distancia:

- La distancia al objeto se calcula con la fórmula:
- $\text{Distancia} = (\text{Tiempo_del_pulso_en_segundos} \times 343 \text{ m/s}) / 2$
- Se divide por 2 porque el tiempo medido incluye **ida y vuelta**.

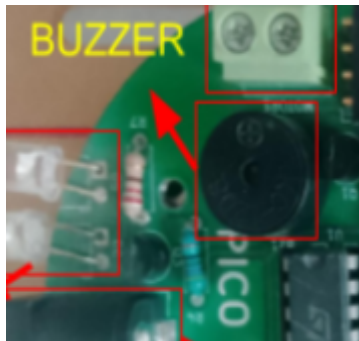
Componentes internos:

- **Emisor ultrasónico:** Genera los pulsos de sonido.
- **Receptor ultrasónico:** Detecta los ecos reflejados por los objetos.

Aplicaciones comunes

- Evitación de obstáculos en robots móviles.
- Sistemas de medición de nivel o distancia.
- Automatización de tareas que requieran detección de presencia o proximidad.

El buzzer:



Buzzer Pasivo

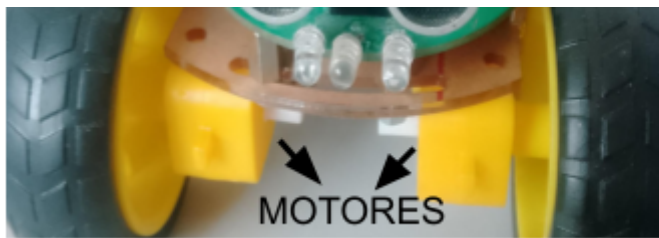
El **buzzer pasivo** es un dispositivo piezoeléctrico que genera sonido **cuando se le aplica una señal eléctrica de frecuencia variable**. A diferencia del buzzer activo, que incluye un oscilador interno y emite un tono fijo al conectarse, el buzzer pasivo **requiere una señal externa** para funcionar, lo que lo hace más versátil.



Características principales:

- **Requiere señal externa:**
Necesita una señal de corriente alterna (AC) o una serie de pulsos de voltaje para generar sonido. La frecuencia de esa señal determina el tono emitido.
- **Versatilidad en la generación de sonidos:**
Permite producir **diferentes tonos y melodías**, ya que el sonido depende de la frecuencia de entrada. Ideal para efectos de sonido personalizados.
- **Consumo de energía variable:**
El consumo depende directamente de la **frecuencia y amplitud** de la señal aplicada.
- **Estructura interna simple:**
Generalmente está compuesto por un **elemento piezoeléctrico o electromagnético** que vibra al recibir la señal.
- **Silencioso cuando está inactivo:**
No produce ningún sonido si no se le aplica señal, a diferencia del buzzer activo, que emite un tono fijo al conectarlo a la alimentación.

Motores:



Motores con Reducción Mecánica:

Los **motores con reducción mecánica** combinan un motor eléctrico con un sistema de **engranajes reductores**. Esta combinación permite **disminuir la velocidad de rotación del eje de salida** y, al mismo tiempo, **aumentar el torque (fuerza de giro)** disponible.

⚙️ ¿Cómo funciona?

- El motor genera rotación a alta velocidad pero con bajo torque.
- El sistema de engranajes **reduce la velocidad de salida** y **multiplica la fuerza**, permitiendo mover objetos más pesados o vencer la fricción con facilidad.

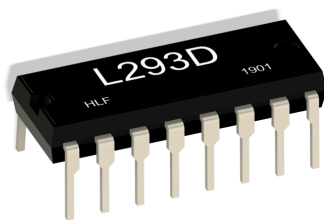
🎯 Ventajas principales:

- **Mayor torque:** Ideal para mover ruedas, levantar cargas o superar obstáculos.
- **Control más preciso:** La reducción de velocidad permite movimientos más suaves y controlados.
- **Aplicaciones a bajas velocidades:** Perfectos para robots móviles que requieren tracción en lugar de velocidad.
- **Eficiencia en robótica educativa:** Son económicos, fáciles de controlar y muy usados en proyectos escolares o de hobby.

🤖 Aplicaciones comunes:

- Robots seguidores de línea o evitadores de obstáculos.
- Sistemas de automatización con movimiento lento y preciso.
- Brazo robótico con articulaciones controladas.
- Vehículos robóticos como el **PICO BOT**.

L293D:



Controlador de Motores L293D

El **L293D** es un **circuito integrado de puente H dual**, diseñado para controlar motores **DC (corriente continua)** y **motores paso a paso**. Gracias a su diseño, permite controlar **hasta dos motores de forma independiente**, incluyendo su **dirección de giro y velocidad**.

⚙️ Características principales

- **Tipo de controlador:** Doble puente H integrado.
- **Cantidad de canales:** 2 (permite controlar 2 motores DC o 1 motor paso a paso).
- **Voltaje de operación del motor (Vs):** Hasta 36 V.
- **Corriente por canal:** Hasta 600 mA (con picos de hasta 1.2 A por corto tiempo).
- **Protección interna:** Diodos de protección integrados contra corrientes inversas (flyback).
- **Control digital:** Compatible con microcontroladores como la **Raspberry Pi Pico**, a través de sus pines GPIO.

🚗 Funcionalidad:

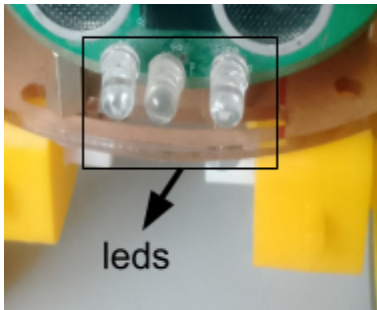
El L293D permite

- Encender o apagar motores de forma digital.
- Cambiar la **dirección** de rotación de cada motor.
- Regular la **velocidad** del motor utilizando señales **PWM** desde los pines GPIO.

🤖 Aplicaciones típicas

- Robótica móvil (vehículos como el **PICO BOT**).
- Control de ruedas, poleas o motores en sistemas automatizados.
- Proyectos educativos con microcontroladores.

Leds:



Diodo LED (Light Emitting Diode)

Un **LED** es un **dispositivo semiconductor** que emite luz cuando se le aplica corriente eléctrica en la **polarización directa**. Este fenómeno se conoce como **electroluminiscencia**.

💡 ¿Cómo funciona?

Cuando la corriente eléctrica atraviesa el LED:

- Los **electrones** en la banda de conducción se recombinan con **huecos** en la banda de valencia.
- Durante esta recombinación, se libera **energía en forma de fotones**, que se manifiesta como **luz visible** o **infrarroja**, según el material del LED.

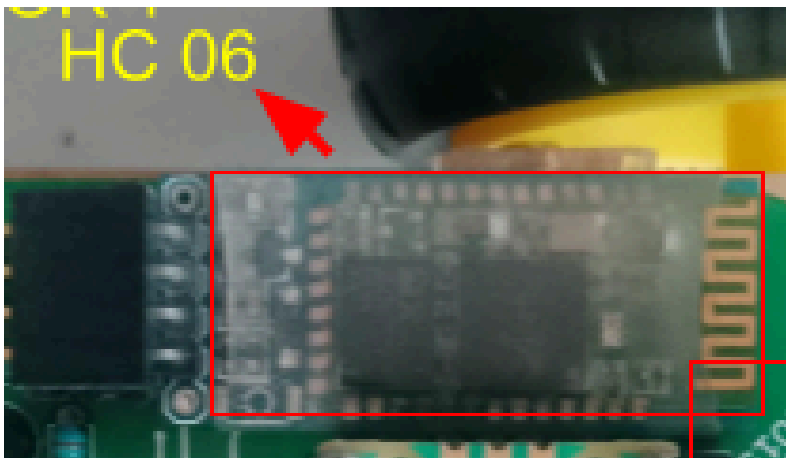
⚙️ Características principales

- **Alta eficiencia energética**
- **Larga vida útil**
- **Bajo consumo de corriente**
- **Alta resistencia a golpes y vibraciones**
- **Tamaño compacto**
- **Está disponible en varios colores y niveles de brillo**, dependiendo del material semiconductor (como GaAs, GaN, InGaN, etc.)

📌 Aplicaciones comunes

- Indicadores de estado en placas electrónicas
- Iluminación LED en general
- Pantallas (como los displays de 7 segmentos o matrices LED)
- Dispositivos electrónicos como tu **PICO BOT**

HC 06:



Módulo Bluetooth HC-06:

El **HC-06** es un módulo Bluetooth ampliamente utilizado para establecer **comunicación inalámbrica** entre microcontroladores y dispositivos como teléfonos móviles, PCs o placas de desarrollo. Es ideal para aplicaciones donde se desea controlar o monitorear un sistema de forma remota, como en el **PICO BOT**.

🔧 Características principales

- **Modo de operación:** Sólo opera como **esclavo** (espera ser emparejado).
 - **Comunicación:** UART (RX y TX).
 - **Velocidad por defecto:** 9600 baudios (bps).
 - **Voltaje de operación:** 3.6V a 6V en VCC.
 - **Compatibilidad lógica:** Pines RX/TX trabajan a 3.3V (ideal para la Raspberry Pi Pico).
 - **Configuración:** Mediante comandos **AT** (para cambiar nombre, baud rate, PIN, etc.).
-

¿Cómo funciona?

El HC-06 actúa como un **punto serial inalámbrico**. Una vez emparejado, **transmite datos UART** entre la Raspberry Pi Pico y un dispositivo Bluetooth (como un teléfono móvil).

Estado del módulo

- **Parpadeo rápido del LED:** En espera de emparejamiento
- **Parpadeo lento:** Conexión establecida (emparejado).

Conexión básica con la Pico

- **HC-06 TX** → **GPIO RX de la Pico**
 - **HC-06 RX** → **GPIO TX de la Pico** (usualmente con divisor resistivo si la Pico trabaja a 5V)
 - **VCC** → **5V**
 - **GND** → **GND**
-

UART (Universal Asynchronous Receiver-Transmitter)

La UART es un protocolo de comunicación **serial asincrónico**, muy simple y eficiente, usado para enviar y recibir datos entre dispositivos electrónicos.

Conceptos clave:

- **Asincrónico:** No necesita señal de reloj. Ambos dispositivos deben tener la **misma velocidad de transmisión (baud rate)**.
- **Solo dos líneas necesarias:**
 - **TX (Transmitter):** Envía datos.
 - **RX (Receiver):** Recibe datos.

Ejemplo de conexión:

- Pico TX → HC-06 RX
- Pico RX → HC-06 TX



Formato típico de datos transmitidos:

1. **Bit de inicio:** Siempre 0 (nivel bajo)
2. **Bits de datos:** Generalmente 8
3. **Bit de paridad (opcional):** Verificación de errores
4. **Bit de parada:** Siempre 1 (nivel alto)



Ejemplo práctico

- La Pico transmite el carácter "A" (ASCII 65, binario 01000001)
- Secuencia transmitida:
 - Bit de inicio: 0
 - Datos: 01000001
 - Bit de parada: 1

El receptor (HC-06) detecta el bit de inicio y **lee los datos** a la velocidad configurada, reconstruyendo el carácter original.



Ventajas de UART:

- **Simplicidad:** Solo requiere dos líneas (TX y RX).
- **Alta compatibilidad:** Presente en casi todos los microcontroladores.
- **Sin reloj compartido:** Ahorra pines.



Limitaciones de UART:

- **Comunicación punto a punto:** No apto para múltiples dispositivos sin hardware adicional.
 - **Distancia limitada:** Idealmente menor a 15 metros sin amplificación.
 - **Velocidad limitada:** Menor que SPI o I2C, aunque suficiente para la mayoría de aplicaciones.
-







Este módulo es ideal para enviar comandos desde el celular al **PICO BOT**, recibir datos de sensores o incluso controlar motores remotamente.

Introducción a MicroPython en la Raspberry Pi Pico

MicroPython es una implementación ligera del lenguaje **Python**, pensada especialmente para **microcontroladores y sistemas embebidos**. Es ideal para proyectos de electrónica, automatización y robótica, gracias a su **sintaxis simple**, su **versatilidad**, y una **amplia biblioteca de módulos integrados**.

¿Qué permite hacer MicroPython en la Raspberry Pi Pico?

Con Micro Python podés controlar fácilmente los recursos de la **Raspberry Pi Pico**, como:

-  Encender y apagar **LEDs**.
-  Leer entradas de **botones** y sensores.
-  Manejar **motores**, **relés** y actuadores.
-  Comunicarse con otros dispositivos usando:
 - **UART** (comunicación serie)
 - **I2C** (sensores, displays, etc.)
 - **SPI** (módulos de memoria, pantallas, etc.)
-  En modelos como la **Raspberry Pi Pico W**, podés crear **servidores web básicos** mediante Wi-Fi.
-  Desarrollar interfaces interactivas con **displays**, teclados o entradas táctiles.

Desarrollo rápido con REPL

MicroPython incluye el modo **REPL** (*Read–Eval–Print Loop*), que permite:

- Escribir código en tiempo real.
- Ejecutar líneas de Python directamente.
- Probar funciones rápidamente sin necesidad de grabar un archivo.

Es una herramienta muy útil para **experimentar y aprender**, sobre todo en el aula o en prototipos rápidos.

Thonny + MicroPython: una combinación ideal

Thonny es un entorno de desarrollo integrado (IDE) diseñado para facilitar la programación, especialmente para quienes están comenzando con Python. Es **liviano, intuitivo** y ofrece una interfaz limpia que facilita el uso de Micro Python con la Raspberry Pi Pico.

¿Por qué usar Thonny?

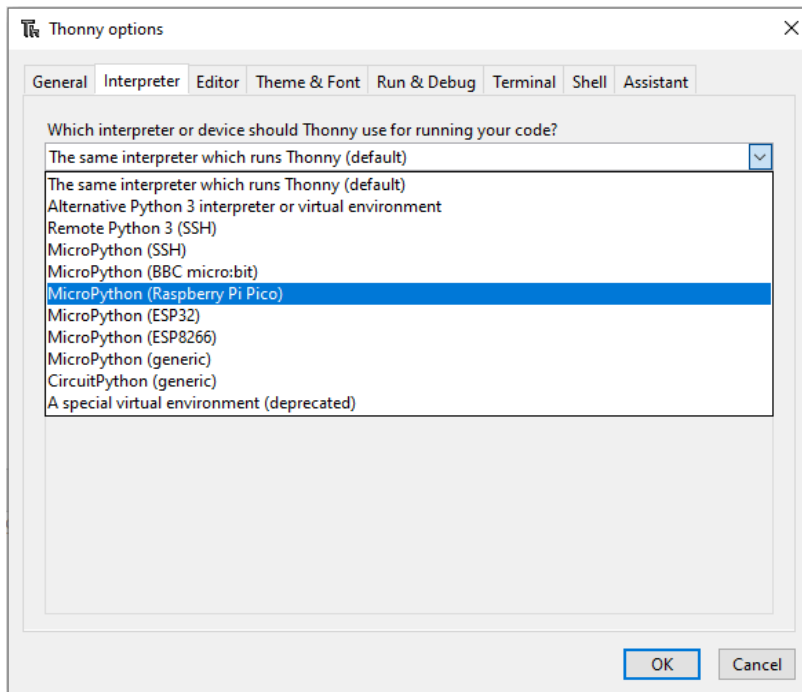
- Instalación sencilla y gratuita.
- Reconoce automáticamente la Raspberry Pi Pico cuando se conecta por USB.
- Permite cargar scripts fácilmente en la placa.
- Ofrece acceso directo al **REPL**.
- Ideal para la enseñanza y la práctica interactiva.

¿Cómo configurar Thonny para MicroPython?

1. **Descargá Thonny** desde su sitio oficial: <https://thonny.org>
Disponible para **Windows, macOS y Linux**.
2. **Instalá y abrí a Thonny**.
3. En el menú, seleccioná:
Herramientas > Configuración > Intérprete

4. En el menú desplegable, selecciona:
 - **MicroPython (Raspberry Pi Pico)**
5. En la opción "**Puerto**", elegí el puerto que detectó tu placa Pico.
(Thonny lo suele detectar automáticamente cuando conectas la placa por USB).

¡Listo! Ya podés comenzar a escribir y cargar código en tu Raspberry Pi Pico usando MicroPython.



Escribir y ejecutar código:

En la ventana del editor, escribe tu código MicroPython.



Ejemplo básico: Parpadeo de un LED

Un ejemplo típico para comenzar con Micro Python en la Raspberry Pi Pico es hacer **parpadear un LED**. En la Pico, el LED integrado está conectado al **GPIO 25**.

Código MicroPython

```
from machine import Pin
from time import sleep

led = Pin(25, Pin.OUT) # LED integrado en la Pico

while True:
    led.toggle() # Cambia el estado del LED
    sleep(0.5)   # Espera medio segundo
```

Guardar el archivo

Guarda el archivo en Raspberry Pi Pico como `main.py`.
Esto permite que el programa se **ejecute automáticamente al encender la placa**.






Uso del REPL y depuración

En **Thonny**, el panel inferior te da acceso al **REPL** (Read–Eval–Print Loop), donde podés:

- Ejecutar comandos de forma interactiva.
- Probar líneas de código sin guardar un archivo.
- Ver resultados y errores en tiempo real.

Ideal para hacer pruebas rápidas mientras desarrollas tu proyecto.

✨ Ventajas de usar Thonny con MicroPython

-  **Interfaz amigable** para principiantes.
-  **Soporte nativo** para Raspberry Pi Pico y Micro Python.
-  **Depurador visual** para encontrar errores fácilmente.
-  Compatible con módulos y bibliotecas de Micro Python.
-  Permite cargar scripts y usar el REPL en un solo entorno.

Thonny hace que programar en Micro Python sea **rápido, cómodo y efectivo**, permitiéndote concentrarse en tus ideas y no en la configuración.

Comentarios en MicroPython

Los comentarios son secciones de texto dentro del código que **no se ejecutan**. Sirven para:

- Explicar el propósito del código.
- Dejar notas para uno mismo o para otros.
- Mejorar la claridad y el mantenimiento del programa.



Tipos de comentarios

1. Comentarios de una sola línea

Usan el símbolo #. Todo lo que sigue será ignorado por el intérprete.

```
# Esto es un comentario
led = Pin(25, Pin.OUT) # También se puede comentar al final de una línea de código
```

2. Comentarios de varias líneas

No existe una sintaxis especial. Se utilizan varias líneas con #:

```
# Este es un comentario extenso
# que explica algo complejo en varias líneas.
```

⚠ Aunque se pueden usar comillas triples (""") para bloques, en MicroPython **no se recomienda**, ya que se interpretan como cadenas de texto (docstrings).



Buenas prácticas para comentar

- Sé **claro y conciso**: Evita comentarios innecesarios.
- Explica el “**por qué**”, no solo el “**qué**”.
- No repitas lo que ya es obvio:

```
led = Pin(25, Pin.OUT) # Bien, si se necesita aclarar el propósito
led = Pin(25, Pin.OUT) # Configura el pin 25 como salida → innecesario si el nombre lo dice todo
```

- Comentá solo lo necesario para **entender el código** sin sobrecargarlo.
- Comentá funciones complejas o secciones críticas.

La indentación en MicroPython

La **indentación** es la forma en que se organiza el código mediante espacios o tabulaciones para indicar la estructura jerárquica. En Micro Python, al igual que en Python, la indentación **no es opcional**, sino fundamental para definir bloques de código y la relación entre ellos.

Reglas clave de la indentación en MicroPython

- **Bloques de código:** Cada bloque —como el cuerpo de un bucle, función o condicional— debe estar indentado.

```
if True:
    print("Este código está dentro del bloque")
```

- **Consistencia:** Usá siempre el mismo número de espacios o tabulación para cada nivel de indentación. Se recomiendan **4 espacios por nivel**.
 - **No mezclar espacios y tabulaciones**, ya que puede causar errores difíciles de detectar.
-

Errores comunes por mala indentación

- **Falta de indentación:**

```
if True:
print("Esto causará un error")
```

- **Inconsistencia en los niveles de indentación:**

```
if True:
    print("Nivel 1")
print("Esto tiene un nivel distinto y causará un error")
```

Ejemplo correcto de indentación en MicroPython

```
from machine import Pin
from time import sleep

led = Pin(25, Pin.OUT)

for i in range(5): # Bucle que se ejecuta 5 veces
    led.toggle()   # Cambia el estado del LED
    sleep(0.5)     # Espera medio segundo
```

La indentación asegura que el código sea **claro y estructurado**. Si no se respeta, Micro Python generará errores y no ejecutará el programa.

Lección 1: Control de LEDs con Raspberry Pi Pico

Este ejemplo controla tres LEDs (blancos, verdes y rojos) conectados a la Raspberry Pi Pico.

Configuración de los pines GPIO

Se importan los módulos necesarios:

```
import time
from machine import Pin
```

- `Pin` permite configurar y controlar los pines GPIO.
 - El `tiempo` se usa para funciones relacionadas con el tiempo, como `sleep`.
-

Definición de pines para los LEDs

```
blancos = Pin(6, Pin.OUT) # LED blancos en GPIO 6
verdes = Pin(27, Pin.OUT) # LED verdes en GPIO 27
rojos = Pin(26, Pin.OUT)  # LED rojos en GPIO 26
```

Ciclo infinito que controla los LEDs

```
while True:
    verdes.value(0) # Apaga LEDs verdes
    blancos.value(1) # Enciende LEDs blancos
    time.sleep(1)   # Espera 1 segundo
```

```
blancos.value(0)    # Apaga LEDs blancos
rojos.value(1)      # Enciende LEDs rojos
time.sleep(1)       # Espera 1 segundo

rojos.value(0)      # Apaga LEDs rojos
verdes.value(1)     # Enciende LEDs verdes
time.sleep(1)       # Espera 1 segundo antes de reiniciar el ciclo
```

Resumen

- Los LEDs se encienden uno por uno: **blancos** → **rojos** → **verdes**.
- Cada LED permanece encendido durante **1 segundo**.
- El patrón se repite indefinidamente.

Lección 2: Parpadeo secuencial de LEDs con ciclo for

Explicación general del código

Este código controla tres LEDs (blancos, verdes y rojos) para que **parpadeen en secuencia**, utilizando un **bucle infinito** y ciclos **for** para repetir el encendido y apagado de cada LED un número específico de veces.

Funcionamiento del código

Configuración de LEDs

Se asignan los pines GPIO correspondientes a cada LED:

- `blancos` en GPIO 6
 - `verdes` en GPIO 27
 - `rojos` en GPIO 26
-

Bucle infinito (`while True`)

El programa se ejecuta indefinidamente, repitiendo la secuencia completa de parpadeo.

Uso del ciclo `for`

El ciclo `for` permite repetir un bloque de código un número definido de veces.

Ejemplo para encender y apagar el LED blanco 2 veces:

```
for _ in range(2): # Repite 2 veces
    blancos.value(1) # Enciende el LED blanco
    time.sleep(0.5) # Espera 0.5 segundos
    blancos.value(0) # Apaga el LED blanco
    time.sleep(0.5) # Espera 0.5 segundos
```

- El guión bajo se usa como variable de iteración cuando no necesitamos su valor.
 - `range(2)` indica que el bloque se repite 2 veces.
-

Secuencia completa

- Los LEDs **blancos** parpadean 2 veces.
- Los LEDs **verdes** parpadean 3 veces.
- Los LEDs **rojos** parpadean 4 veces.

Luego, el ciclo infinito repite esta secuencia continuamente.

Código completo

```
from machine import Pin
import time

# Configuración de pines GPIO para LEDs
blancos = Pin(6, Pin.OUT)
verdes = Pin(27, Pin.OUT)
rojos = Pin(26, Pin.OUT)

while True:
    # Parpadeo LED blanco 2 veces
    for _ in range(2):
```

```
    blancos.value(1) # Enciende LED blanco
    time.sleep(0.5)
    blancos.value(0) # Apaga LED blanco
    time.sleep(0.5)

# Parpadeo LED verde 3 veces
for _ in range(3):
    verdes.value(1) # Enciende LED verde
    time.sleep(0.5)
    verdes.value(0) # Apaga LED verde
    time.sleep(0.5)

# Parpadeo LED rojo 4 veces
for _ in range(4):
    rojos.value(1) # Enciende LED rojo
    time.sleep(0.5)
    rojos.value(0) # Apaga LED rojo
    time.sleep(0.5)
```

Resumen

- El ciclo `for` facilita repetir acciones un número definido de veces.
- La secuencia hace que cada LED parpadee una cantidad distinta de veces antes de pasar al siguiente.
- El ciclo `while True` asegura que esta secuencia se repita indefinidamente.

Lección 3: Parpadeo de LEDs sin ciclo infinito

Explicación general del código

En esta lección se controla el parpadeo de tres LEDs (blancos, verdes y rojos) sin utilizar un bucle infinito (`while True`). Esto significa que:

- La secuencia de parpadeo se ejecuta **una sola vez**.
 - Al terminar, el programa finaliza y no repite la secuencia.
-

Configuración de LEDs

Se asignan tres pines GPIO como salidas para controlar los LEDs:

- blancos → GPIO 6
 - verdes → GPIO 27
 - rojos → GPIO 26
-

Secuencia de parpadeo

Cada LED parpadea una cantidad diferente de veces utilizando ciclos `for`:

- LED blancos: 2 parpadeos (encendido y apagado con pausas de 0.5 s)
- LED verdes: 3 parpadeos
- LED rojos: 4 parpadeos

La diferencia principal respecto a la lección anterior es que **no se usa `while True`**, por lo que el programa no se repite indefinidamente.

¿Qué ocurre sin `while True`?

- El programa ejecuta la secuencia de parpadeo **una sola vez**.
 - Después de terminar, el código **deja de ejecutarse** y los LEDs permanecen apagados.
 - Esto contrasta con el uso de `while True`, que hace que la secuencia se repita constantemente.
-

Código completo

```
from machine import Pin
import time

# Configuración de pines GPIO para LEDs
blancos = Pin(6, Pin.OUT)
verdes = Pin(27, Pin.OUT)
rojos = Pin(26, Pin.OUT)

# Parpadeo LED blanco 2 veces
for _ in range(2):
    blancos.value(1) # Enciende LED blanco
```

```
time.sleep(0.5)
blancos.value(0) # Apaga LED blanco
time.sleep(0.5)

# Parpadeo LED verde 3 veces
for _ in range(3):
    verdes.value(1) # Enciende LED verde
    time.sleep(0.5)
    verdes.value(0) # Apaga LED verde
    time.sleep(0.5)

# Parpadeo LED rojo 4 veces
for _ in range(4):
    rojos.value(1) # Enciende LED rojo
    time.sleep(0.5)
    rojos.value(0) # Apaga LED rojo
    time.sleep(0.5)
```

Lección 4: Modularización con funciones para controlar LEDs

Explicación general

Este código controla tres LEDs (blancos, verdes y rojos) conectados a pines GPIO, haciendo que parpadeen una cantidad específica de veces en una secuencia repetitiva.

La modularización mediante funciones facilita:

- Organizar el código por tareas específicas (control de cada color).
- Ajustar fácilmente el comportamiento (número de parpadeos o tiempos).
- Mejorar la legibilidad y mantenimiento.

Funciones definidas

- **leds blancos()**: Hace parpadear los LEDs blancos 2 veces, con pausas de 0.5 segundos.
 - **leds verdes()**: Hace parpadear los LEDs verdes 3 veces.
 - **leds rojos()**: Hace parpadear los LEDs rojos 4 veces.
-

Flujo del programa

El bucle infinito (`while True`) llama a cada función en secuencia:

1. `leds blancos()` → 2 parpadeos
2. `leds verdes()` → 3 parpadeos
3. `leds rojos()` → 4 parpadeos

Luego repite la secuencia indefinidamente.

```
from machine import Pin
import time

# Configuración de pines GPIO para LEDs
blancos = Pin(6, Pin.OUT)
verdes = Pin(27, Pin.OUT)
rojos = Pin(26, Pin.OUT)

# Función para parpadear LED blanco 2 veces
def leds_blancos():
    for _ in range(2):
        blancos.value(1)
        time.sleep(0.5)
        blancos.value(0)
        time.sleep(0.5)

# Función para parpadear LED verde 3 veces
def leds_verdes():
    for _ in range(3):
        verdes.value(1)
        time.sleep(0.5)
        verdes.value(0)
        time.sleep(0.5)

# Función para parpadear LED rojo 4 veces
def leds_rojos():
    for _ in range(4):
        rojos.value(1)
        time.sleep(0.5)
        rojos.value(0)
        time.sleep(0.5)

# Bucle principal que ejecuta la secuencia indefinidamente
while True:
    leds_blancos()
    leds_verdes()
    leds_rojos()
```

Lección 5: Sensor de luz con módulo LDR

Introducción

Este código utiliza un módulo con un LDR (Resistencia Dependiente de la Luz) para detectar la cantidad de luz ambiental. Según esta detección, controla el encendido y apagado de los LEDs blancos.

¿Qué es un LDR?

Un **LDR (Light Dependent Resistor)** es un componente electrónico cuya resistencia varía en función de la luz que recibe:

- **Más luz → menor resistencia.**
- **Menos luz → mayor resistencia.**

Esto permite medir la intensidad luminosa en un entorno.

Cómo funciona un módulo con LDR

El módulo típico incluye varios componentes:

1. **LDR:** El sensor que detecta la luz.
2. **Resistencia fija:** Forma un divisor de voltaje junto con el LDR.
3. **Comparador (opcional):** Como el LM393, que genera una salida digital cuando se supera un umbral.
4. **Trimpot (potenciómetro ajustable):** Permite ajustar la sensibilidad del sensor.
5. **Salidas:**
 - **Salida analógica (A0):** Voltaje proporcional a la intensidad luminosa.
 - **Salida digital (D0):** Señal ON/OFF según el umbral configurado.

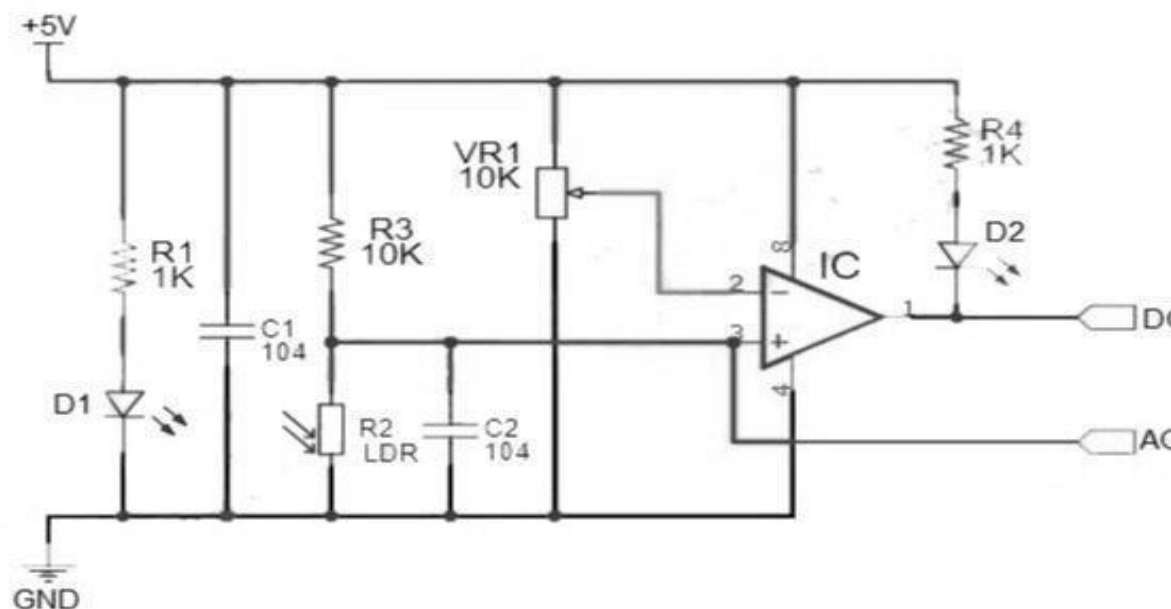
Divisor de voltaje

El LDR y la resistencia fija forman un divisor de voltaje que genera un voltaje de salida proporcional a la luz recibida:

$$V_{out} = V_{in} \times \frac{R_{LDR}}{R_{LDR} + R_{fija}}$$

Donde:

- R_{LDR} : Resistencia variable del LDR (depende de la luz).



Aplicaciones del módulo LDR

- Encender luces automáticamente en condiciones de poca luz.
 - Detectores de paso en sistemas de seguridad.
 - Medición de intensidad lumínica en proyectos electrónicos.
 - Seguidores solares, que ajustan paneles según la luz recibida.
-

Uso con MicroPython

Puedes conectar la salida analógica o digital del módulo LDR a tu microcontrolador y leer la señal para tomar decisiones o controlar dispositivos (como LEDs).

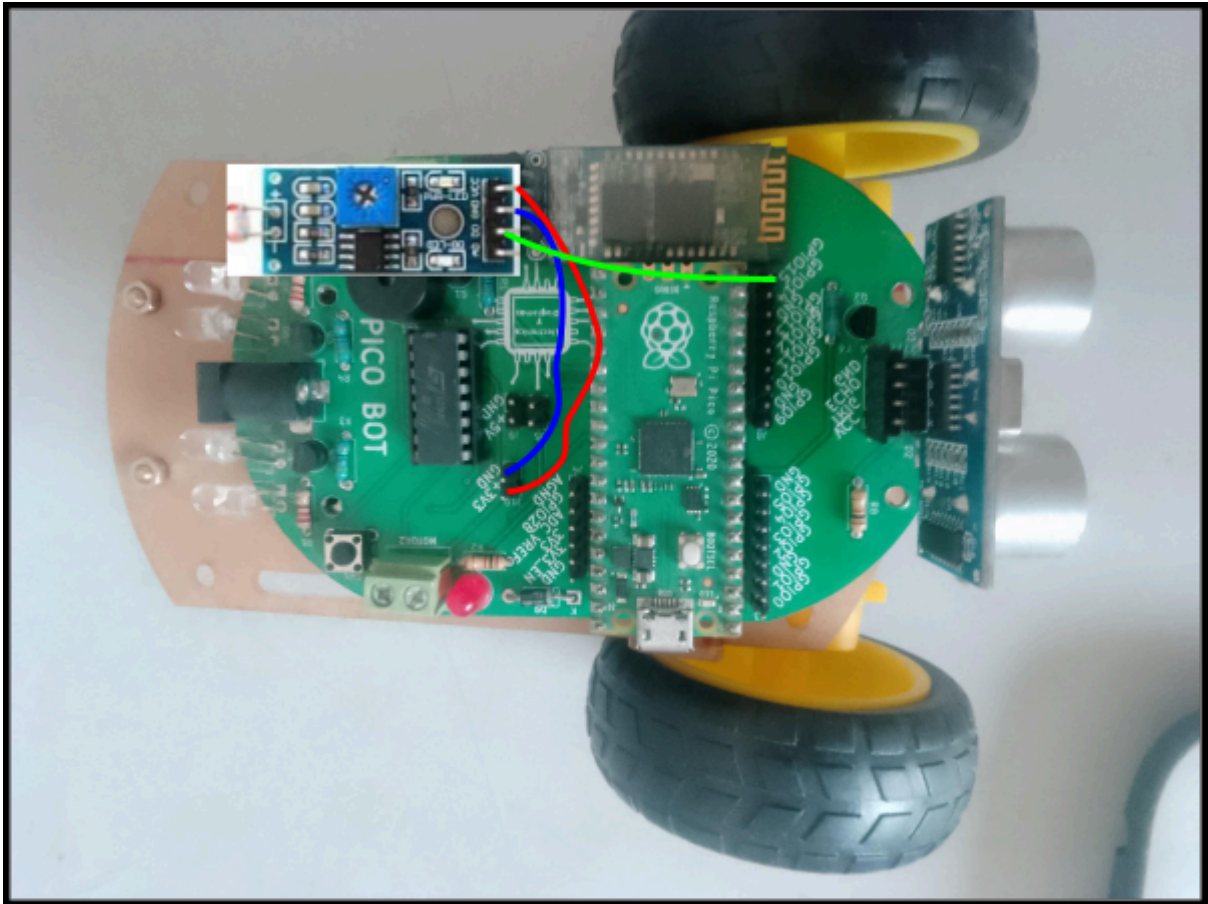
Configuración del hardware

- **LEDs:** conectados al pin GPIO 6, configurados como salida (`Pin.OUT`).
 - **Módulo LDR:** conectado al pin GPIO 15, configurado como entrada con resistencia pull-down interna (`Pin.IN`, `Pin.PULL_DOWN`). Esto asegura que el valor leído sea 0 cuando no hay señal (ausencia de luz).
-

Lógica de funcionamiento

- Cuando el sensor detecta luz, la entrada en GPIO 15 cambia a **1** (alto).
- Cuando no hay luz o es baja, la entrada se mantiene en **0** (bajo).
- Según este valor, el código puede encender o apagar los LEDs conectados.

Conexión del módulo ver imagen:



El código está dentro de un bucle infinito (`while True`) para leer continuamente el estado del módulo LDR.

- Cuando el módulo detecta luz, su pin D0 genera 3.3 V, y en el código se lee como `modulo_ldr.value() == 1`. En ese caso, los LEDs se apagan: `blancos.value(0)`.
- Cuando no detecta luz, el pin D0 tiene un voltaje cercano a 0V, y el código lee `modulo_ldr.value() == 0`. Entonces, los LEDs se encienden: `blancos.value(1)`.

En resumen, los LEDs se apagan cuando hay luz y se encienden en la oscuridad, todo controlado por el sensor LDR y el pin GPIO 6 de la Raspberry Pi Pico.

```
from machine import Pin
import time

# Configuración del pin para el LED blanco
blancos = Pin(6, Pin.OUT) # LED conectado al GPIO 6 configurado como
salida

# Configuración del pin para el sensor LDR
modulo_ldr = Pin(15, Pin.IN, Pin.PULL_DOWN)

while True:
    if modulo_ldr.value() == 1: # Detecta luz
        blancos.value(0)        # Apaga el LED
    else:                        # No detecta luz
        blancos.value(1)        # Enciende el LED
    time.sleep(0.1) # Pequeña pausa para evitar lecturas muy rápidas
```

Lección 6: Uso de un buzzer con PWM

Configuración inicial del buzzer:

```
buzzer = PWM(Pin(22))
```

Esto configura el buzzer en el pin GPIO 22 utilizando modulación por ancho de pulso (PWM), lo cual permite generar distintos tonos.

Funcionamiento del código:

Bucle infinito (`while True`):

El buzzer alterna entre dos frecuencias, generando un efecto similar al de una sirena.

1. Frecuencia alta:

- Se establece la frecuencia en 1000 Hz con `buzzer.freq(1000)`.
- Se activa el buzzer con `buzzer.duty_u16(10000)`. Este valor representa la amplitud de la señal (el volumen). El máximo es 65535.
- El buzzer suena durante 0.3 segundos.

2. Frecuencia baja:

- La frecuencia cambia a 600 Hz con `buzzer.freq(600)`.
- Se mantiene el volumen (`duty_u16(10000)`).
- Suena durante 0.3 segundos.

3. Silencio (opcional):

- Se apaga el buzzer con `buzzer.duty_u16(0)`, eliminando el sonido.
 - Se pausa 0.1 segundos.
-

Funcionamiento continuo:

Este ciclo se repite indefinidamente, alternando entre tonos altos y bajos, generando una sirena continua hasta que se detenga el programa manualmente.

¿Qué es PWM?

PWM (Pulse Width Modulation) o modulación por ancho de pulso, es una técnica digital para controlar dispositivos electrónicos como:

- LEDs (para variar el brillo),
- motores (para variar la velocidad),
- buzzers (para generar tonos).

Conceptos clave:

- **Frecuencia:** número de ciclos por segundo (Hz).
Ej: 1000 Hz = 1000 pulsos por segundo.
 - **Duty cycle (ciclo de trabajo):** porcentaje de tiempo que la señal está en alto ("encendida") en cada ciclo.
Ej: 50% significa que la señal está activa la mitad del tiempo.
-

Código completo:

```
from machine import Pin, PWM
import time

# Configuración del buzzer en el pin GPIO 22
buzzer = PWM(Pin(22))

while True:
    # Frecuencia alta
    buzzer.freq(1000)
    buzzer.duty_u16(10000)
    time.sleep(0.3)

    # Frecuencia baja
    buzzer.freq(600)
    buzzer.duty_u16(10000)
    time.sleep(0.3)

    # Silencio
    buzzer.duty_u16(0)
    time.sleep(0.1)
```

Lección 7: Reproducción de melodías con PWM y diccionarios

1. Configuración del buzzer

```
buzzer = PWM(Pin(22))
```

Se inicializa el buzzer en el pin GPIO 22 y se configura para trabajar con **PWM** (modulación por ancho de pulso), lo cual permite generar sonidos a diferentes frecuencias.

2. Diccionario de notas musicales

```
notas = {  
    'Do': 261, 'Do*': 523, ..., 'Silencio': 0  
}
```

Se crea un **diccionario** que relaciona cada nota musical con su frecuencia en Hz. La clave 'Silencio' está asociada a la frecuencia 0 para representar pausas sin sonido.

3. Melodía (Tema de Mario Bros)

```
melodia = [  
    ('Mi*', 0.15), ('Mi*', 0.15), ..., ('Si', 0.15)  
]
```

La melodía es una lista de **tuplas**, donde cada elemento contiene una nota y su duración. Ejemplo: ('Mi*', 0.15) indica que la nota Mi* debe sonar durante 0.15 segundos.

4. Función para reproducir una nota

```
def reproducir_nota(nota, duracion):  
    if nota in notas:  
        frecuencia = notas[nota]  
        if frecuencia == 0:  
            buzzer.duty_u16(0) # Silencio  
        else:  
            buzzer.freq(frecuencia)  
            buzzer.duty_u16(10000) # Volumen medio  
            time.sleep(duracion)
```

```
buzzer.duty_u16(0)
time.sleep(0.05) # Pausa entre notas
```

Funcionamiento:

- Busca la frecuencia en el diccionario.
- Si la frecuencia es 0, apaga el buzzer (silencio).
- Si es distinta de cero, reproduce la nota con su frecuencia.
- Luego se apaga y hace una pequeña pausa antes de la siguiente nota.

5. Bucle principal

```
while True:
    for nota, duracion in melodia:
        reproducir_nota(nota, duracion)
        time.sleep(1) # Pausa entre repeticiones
```

Este bucle reproduce la melodía completa y espera un segundo antes de repetirla.



Diferencia entre diccionario y tupla en Python

Característica	Diccionario (dict)	Tupla (tuple)
Estructura	{clave: valor}	(elemento1, elemento2, ...)
Acceso	Por clave	Por índice
Mutabilidad	Mutable (se puede modificar)	Inmutable (no se puede cambiar)
Orden	Desde Python 3.7, mantiene orden	Siempre ordenada
Uso típico	Mapear relaciones clave/valor	Agrupar datos relacionados



Código completo – Melodía de Mario Bros

```
from machine import Pin, PWM
import time

# Configuración del buzzer
buzzer = PWM(Pin(22))

# Diccionario de notas musicales
notas = {
    'Do': 261, 'Do*': 523, 'Re': 293, 'Re*': 587, 'Mi': 329, 'Mi*': 659,
    'Fa': 349, 'Fa*': 698, 'Sol': 392, 'Sol*': 784, 'La': 440, 'La*': 880,
    'Si': 493, 'Sib': 466, 'Sol#': 831, 'Re#': 622, 'Fa#': 740, 'La#': 932,
    'Silencio': 0
}
```

```

}

# Melodía: lista de tuplas (nota, duración en segundos)
melodia = [
    ('Mi*', 0.15), ('Mi*', 0.15), ('Silencio', 0.1), ('Mi*', 0.15),
    ('Silencio', 0.1), ('Do*', 0.15), ('Mi*', 0.15), ('Sol*', 0.3),
    ('Silencio', 0.3), ('Sol', 0.3), ('Silencio', 0.3),
    ('Do*', 0.15), ('Silencio', 0.1), ('Sol', 0.15), ('Mi', 0.15),
    ('La', 0.15), ('Si', 0.15), ('Sib', 0.15), ('La', 0.15),
    ('Sol*', 0.15), ('Mi*', 0.15), ('Sol*', 0.15), ('La*', 0.3),
    ('Fa*', 0.15), ('Sol*', 0.15), ('Mi*', 0.15), ('Do*', 0.15),
    ('Re', 0.15), ('Si', 0.15)
]

# Función que reproduce una nota
def reproducir_nota(nota, duracion):
    if nota in notas:
        frecuencia = notas[nota]
        if frecuencia == 0:
            buzzer.duty_u16(0)
        else:
            buzzer.freq(frecuencia)
            buzzer.duty_u16(10000)
        time.sleep(duracion)
        buzzer.duty_u16(0)
        time.sleep(0.05)
    else:
        print(f'Nota "{nota}" no definida.')

# Reproduce la melodía en bucle
while True:
    for nota, duracion in melodia:
        reproducir_nota(nota, duracion)
    time.sleep(1)

```

Lección 8: Reproducción de melodía con luces sincronizadas (Jingle Bells)

🎵 1. Estructura del código

✅ Configuración de pines y variables

- `buzzer = PWM(Pin(22))`: Configura el buzzer en el GPIO 22 usando modulación PWM (Pulse Width Modulation).
 - LEDs:
 - blancos → GPIO 6
 - verdes → GPIO 27
 - rojos → GPIO 26
 - Se define un **diccionario** `notas` que asocia cada nota musical con su frecuencia en Hz. El valor 0 representa un silencio.
-

✅ Melodía: Jingle Bells

- Representada como una lista de **tuplas** (`nota, duración`).
 - Por ejemplo: `('Mi', 0.3)` indica que la nota Mi sonará durante 0.3 segundos.
 - Las tuplas permiten especificar ritmo y pausas entre las notas.
-

✅ Función `reproducir_nota_con_luces()`

Esta función hace tres cosas:

1. **Reproducir la nota**
 - Busca la frecuencia en el diccionario `notas`.
 - Si es `'Silencio'`, apaga el buzzer.
 - Si no, ajusta la frecuencia y el volumen del buzzer.
2. **Sincronizar luces**
 - Enciende los LEDs **en secuencia**: blancos → verdes → rojos.
 - Divide el tiempo total de la nota en tres partes, una para cada LED.
3. **Finalizar**
 - Apaga el buzzer y todos los LEDs.

- Espera brevemente antes de la siguiente nota.

✓ Bucle principal

- Recorre toda la melodía.
- Reproduce cada nota llamando a `reproducir_nota_con_luces()`.
- Espera 2 segundos antes de volver a empezar.

🧠 Resumen del funcionamiento

- El buzzer reproduce las notas de la canción.
- Las luces LED se encienden sincronizadas con el ritmo.
- Las pausas, los silencios y la melodía están perfectamente representadas.
- ¡Una excelente forma de combinar sonido y luz!

✓ Código completo (Jingle Bells con luces sincronizadas)

```
from machine import Pin, PWM
import time

# Configuración del buzzer y los LEDs
buzzer = PWM(Pin(22))
blancos = Pin(6, Pin.OUT)
verdes = Pin(27, Pin.OUT)
rojos = Pin(26, Pin.OUT)

# Diccionario de notas con frecuencias (Hz)
notas = {
    'Do': 261, 'Do*': 523, 'Re': 293, 'Re*': 587, 'Mi': 329, 'Mi*': 659,
    'Fa': 349, 'Fa*': 698, 'Sol': 392, 'Sol*': 784, 'La': 440, 'La*':
880,
    'Si': 493, 'Sib': 466, 'Sol#': 831, 'Re#': 622, 'Fa#': 740, 'La#':
932,
    'Silencio': 0
}

# Melodía de Jingle Bells
melodia = [
    ('Mi', 0.3), ('Mi', 0.3), ('Mi', 0.6),
```

```

('Mi', 0.3), ('Mi', 0.3), ('Mi', 0.6),
('Mi', 0.3), ('Sol', 0.3), ('Do', 0.3), ('Re', 0.3), ('Mi', 0.6),
('Fa', 0.3), ('Fa', 0.3), ('Fa', 0.3), ('Fa', 0.3),
('Fa', 0.3), ('Mi', 0.3), ('Mi', 0.3), ('Mi', 0.3), ('Mi', 0.3),
('Mi', 0.3), ('Re', 0.3), ('Re', 0.3), ('Mi', 0.3), ('Re', 0.6),
('Sol', 0.6),
('Mi', 0.3), ('Mi', 0.3), ('Mi', 0.6),
('Mi', 0.3), ('Sol', 0.3), ('Do', 0.3), ('Re', 0.3), ('Mi', 0.6),
('Sol', 0.3), ('Sol', 0.3), ('Fa', 0.3), ('Re', 0.3), ('Do', 0.6)
]

```

Función que reproduce una nota con luces sincronizadas

```
def reproducir_nota_con_luces(nota, duracion):
```

```
    if nota in notas:
```

```
        frecuencia = notas[nota]
```

```
        # Encender LED blanco
```

```
        blancos.value(1)
```

```
        verdes.value(0)
```

```
        rojos.value(0)
```

```
        if frecuencia == 0:
```

```
            buzzer.duty_u16(0)
```

```
        else:
```

```
            buzzer.freq(frecuencia)
```

```
            buzzer.duty_u16(10000)
```

```
        time.sleep(duracion / 3)
```

```
        # Encender LED verde
```

```
        blancos.value(0)
```

```
        verdes.value(1)
```

```
        time.sleep(duracion / 3)
```

```
        # Encender LED rojo
```

```
        verdes.value(0)
```

```
        rojos.value(1)
```

```
        time.sleep(duracion / 3)
```

```
        # Apagar buzzer y LEDs
```

```
        buzzer.duty_u16(0)
```

```
        rojos.value(0)
```

```
        time.sleep(0.05)
```

```
    else:
```

```
        print(f'Nota "{nota}" no definida.')
```

```
# Bucle principal
while True:
    for nota, duracion in melodia:
        reproducir_nota_con_luces(nota, duracion)
    time.sleep(2) # Espera entre repeticiones
```

Lección 9: Reproducción paralela de melodía y luces usando los dos núcleos de la Raspberry Pi Pico

1. Explicación general del código

Configuración inicial

- Se configura un buzzer en el **GPIO 22** usando **PWM**, lo que permite reproducir tonos musicales variando la frecuencia.
 - Se configuran **tres LEDs**:
 - Blanco en **GPIO 6**
 - Verde en **GPIO 27**
 - Rojo en **GPIO 26**
 - Se define un **diccionario de notas** con frecuencias en Hertz. El valor `'Silencio': 0` permite insertar pausas.
-

Melodía y luces

- La **melodía** es una lista de tuplas `(nota, duración)`, indicando qué nota tocar y durante cuánto tiempo.
 - La **secuencia de luces** también es una lista de tuplas `(LED, duración)`, donde se especifica qué LED se enciende y por cuánto tiempo.
-

2. Funciones principales

`reproducir_melodia()`

- Recorre la lista de la melodía.
- Si la nota es `'Silencio'`, apaga el buzzer.
- En caso contrario, ajusta la frecuencia y activa el buzzer.
- Espera la duración indicada para cada nota.



controlar_luces()

- Recorre indefinidamente la lista de luces.
 - Enciende cada LED durante su duración, lo apaga, y pasa al siguiente.
-



3. Ejecución paralela con los dos núcleos



Cómo trabaja cada núcleo:

Núcleo	Función asignada	Descripción
Core 0 (principal)	controlar_luces()	Controla los LEDs con una secuencia repetitiva.
Core 1 (secundario)	reproducir_melodia()	Ejecuta la melodía de manera independiente.

Se utiliza `_thread.start_new_thread()` para ejecutar la melodía en paralelo.



Ventajas del paralelismo

- Permite que **sonido y luces funcionen simultáneamente** sin bloquearse entre sí.
 - Mejora la **fluidez**, ya que no es necesario alternar manualmente entre tareas.
 - **Aprovecha los dos núcleos del microcontrolador**, mejorando el rendimiento general.
-



Código completo

```
from machine import Pin, PWM
import time
import _thread

# Configuración del buzzer
buzzer = PWM(Pin(22))

# Configuración de los LEDs
blancos = Pin(6, Pin.OUT)
verdes = Pin(27, Pin.OUT)
rojos = Pin(26, Pin.OUT)

# Diccionario de notas musicales
```

```

notas = {
    'Do': 261, 'Re': 293, 'Mi': 329, 'Fa': 349, 'Sol': 392,
    'La': 440, 'Si': 493, 'Do*': 523, 'Re*': 587, 'Mi*': 659,
    'Silencio': 0
}

# Melodía extendida (nota, duración en segundos)
melodia = [
    ('Do', 0.5), ('Re', 0.5), ('Mi', 0.5), ('Fa', 0.5),
    ('Sol', 0.5), ('La', 0.5), ('Si', 0.5), ('Do*', 0.5),
    ('Si', 0.5), ('La', 0.5), ('Sol', 0.5), ('Fa', 0.5),
    ('Mi', 0.5), ('Re', 0.5), ('Do', 0.5),
    ('Silencio', 0.5),
    ('Do', 0.25), ('Re', 0.25), ('Mi', 0.25), ('Re', 0.25),
    ('Do', 0.5), ('Sol', 0.5), ('Fa', 0.5), ('Mi', 0.5),
    ('Re', 0.5), ('Do', 0.5), ('Silencio', 0.5)
]

# Secuencia de luces (LED, duración en segundos)
luces = [
    (blancos, 0.5), (verdes, 0.5), (rojos, 0.5),
    (blancos, 0.25), (verdes, 0.25), (rojos, 0.25),
    (verdes, 0.5), (blancos, 0.5), (rojos, 0.5)
]

# Función para reproducir la melodía
def reproducir_melodia():
    for nota, duracion in melodia:
        frecuencia = notas[nota]
        if frecuencia == 0: # Silencio
            buzzer.duty_u16(0)
        else:
            buzzer.freq(frecuencia)
            buzzer.duty_u16(10000)
            time.sleep(duracion)
    buzzer.duty_u16(0) # Apagar buzzer al final

# Función para controlar las luces
def controlar_luces():
    while True:
        for led, duracion in luces:
            led.on()
            time.sleep(duracion)
            led.off()

# Ejecutar funciones en paralelo con los dos núcleos

```

```
_thread.start_new_thread(reproducir_melodia, ()) # Core 1
controlar_luces() # Core 0
```

Lección 10: Movimiento autónomo con sensor ultrasónico y LEDs

Objetivo:

Controlar al **PICO BOT** utilizando un sensor ultrasónico para detectar obstáculos y tomar decisiones de movimiento (avanzar, retroceder, girar), con indicación visual mediante LEDs.

1. Configuración del hardware

- **Sensor ultrasónico:**
 - **TRIG** (GPIO 7): Envía un pulso.
 - **ECHO** (GPIO 8): Recibe el pulso reflejado y calcula la distancia.
 - **LEDs indicadores:**
 - **Blanco (GPIO 6) y Verde (GPIO 27):** Indican que el robot avanza.
 - **Rojo (GPIO 26):** Indica que el robot está detenido.
 - **Motores:**
 - Motor A: GPIO 18 y 19.
 - Motor B: GPIO 20 y 21.
-

2. Función del sensor ultrasónico

- 10 microsegundos por el pin TRIG.
- Mide el tiempo que tarda el pulso en volver por ECHO.
- Calcula la distancia en centímetros usando la fórmula:

$$\text{distancia (cm)} = \frac{\text{duración del pulso} \times 0.0343}{2}$$

- Donde $0.0343 \frac{cm}{\mu s}$ es la velocidad del sonido en μs .
- Retorna True si hay un obstáculo a menos de 20cm.

3. Funciones de movimiento del robot

Función	Descripción
<code>adelante()</code>	Avanza, encendiendo LEDs blancos y verdes.
<code>atras()</code>	Retrocede.
<code>derecha()</code>	Gira hacia la derecha.
<code>parar()</code>	Detiene el robot y enciende el LED rojo.

4. Bucle principal

El robot evalúa continuamente si hay un obstáculo:

- **Si detecta uno:**
 - Se detiene.
 - Retrocede por 0.5 segundos.
 - Gira a la derecha 0.5 segundos.
 - Luego reintentar avanzar.
 - **Si no hay obstáculo:**
 - Avanza normalmente.
-

Código completo

```
from machine import Pin, PWM
import time

# --- Configuración de pines ---

# Sensor ultrasónico
trig = Pin(7, Pin.OUT)
echo = Pin(8, Pin.IN)

# LEDs indicadores
blancos = Pin(6, Pin.OUT)
verdes = Pin(27, Pin.OUT)
rojas = Pin(26, Pin.OUT)

# Motores
motorA1 = Pin(18, Pin.OUT)
motorA2 = Pin(19, Pin.OUT)
```

```

motorB1 = Pin(20, Pin.OUT)
motorB2 = Pin(21, Pin.OUT)

# --- Función del sensor ultrasónico ---
def ultrasonico_sensor():
    """Mide la distancia y devuelve True si hay un obstáculo a menos de 20
    cm."""
    trig.value(1)
    time.sleep_us(10)
    trig.value(0)

    while echo.value() == 0:
        start_time = time.ticks_us()
    while echo.value() == 1:
        end_time = time.ticks_us()

    pulse_duration = time.ticks_diff(end_time, start_time)
    distance = (pulse_duration * 0.0343) / 2
    print("Distancia:", distance, "cm")
    time.sleep(0.1)

    return distance < 20

# --- Funciones de movimiento ---
def adelante():
    rojas.value(0)
    blancos.value(1)
    verdes.value(1)
    motorA1.high()
    motorA2.low()
    motorB1.low()
    motorB2.high()

def atras():
    motorA1.low()
    motorA2.high()
    motorB1.high()
    motorB2.low()

def derecha():
    motorA1.high()
    motorA2.low()
    motorB1.high()
    motorB2.low()

def parar():
    rojas.value(1)
    blancos.value(0)
    verdes.value(0)
    motorA1.low()
    motorA2.low()

```



```
motorB1.low()
motorB2.low()

# --- Bucle principal ---
while True:
    if ultrasonico_sensor():
        parar()
        time.sleep(0.5)
        atras()
        time.sleep(0.5)
        parar()
        time.sleep(0.5)
        derecha()
        time.sleep(0.5)
        parar()
        time.sleep(0.5)
    else:
        adelante()
```

Lección 11: Control del PICO BOT por Bluetooth

Aplicación: *BT Car Controller - Arduino/ESP*

Objetivo:

Controlar el PICO BOT desde el celular a través de Bluetooth, usando la app *BT Car Controller*, incluyendo:

- Movimiento con motores
 - Encendido de LEDs
 - Bocina con buzzer
 - Evitación de obstáculos con sensor ultrasónico
-

Hardware usado:

- Bluetooth HC-05 o HC-06
- Raspberry Pi Pico / RP2040
- Motores conectados a GPIO 18–21
- Sensor ultrasónico HC-SR04 (Trig: GPIO 7, Echo: GPIO 8)
- LEDs: blanco (GPIO 6), verde (GPIO 27), rojo (GPIO 26)

- **Buzzer:** GPIO 22 (PWM)
 - **Comunicación UART:** TX GPIO 16, RX GPIO 17
-

Descripción del funcionamiento:

Comunicación Bluetooth UART

El módulo Bluetooth está conectado por UART (9600 baudios) y recibe comandos desde el celular.

Comandos Bluetooth recibidos:

Comando	Acción
F	Avanzar
B	Retroceder
L	Giro derecha
R	Giro izquierda
S	Detenerse
W/w	Encender/apagar led blanco
U/u	Encender/apagar led verde
X	Reproducir melodía de inicio
V/v	Encender /apagar buzzer



Recomendación de App:

BT Car Controller - Arduino/ESP



Google Play Store



Seguridad:

Si el sensor ultrasónico detecta un obstáculo a menos de 15 cm mientras avanza (F), el robot:

- Se detiene
- Emite sonido con buzzer
- Enciende LED blanco y rojo



Código completo:

```
from machine import UART, Pin, PWM
import time, _thread

# Comunicación Bluetooth (UART0)
modulo = UART(0, 9600, tx=Pin(16), rx=Pin(17))

# Pines de sensores, motores, LEDs y buzzer
trig = Pin(7, Pin.OUT)
echo = Pin(8, Pin.IN)
motora1 = Pin(18, Pin.OUT)
motora2 = Pin(19, Pin.OUT)
motorb1 = Pin(20, Pin.OUT)
motorb2 = Pin(21, Pin.OUT)
blancas = Pin(6, Pin.OUT)
verdes = Pin(27, Pin.OUT)
```

```
rojas = Pin(26, Pin.OUT)
buzzer = PWM(Pin(22))

# Funciones de movimiento
def atras():
    motora1.high()
    motora2.low()
    motorb1.low()
    motorb2.high()

def adelante():
    motora1.low()
    motora2.high()
    motorb1.high()
    motorb2.low()

def derecha():
    motora1.low()
    motora2.high()
    motorb1.low()
    motorb2.high()

def izquierda():
    motora1.high()
    motora2.low()
    motorb1.high()
    motorb2.low()

def parar():
    motora1.low()
    motora2.low()
    motorb1.low()
    motorb2.low()
    rojas.value(1)
    time.sleep(0.3)
    rojas.value(0)

# Funciones auxiliares
def bncs():
    blancas.value(1)

def ver():
    verdes.value(1)

def bocina():
    buzzer.freq(500)
    buzzer.duty_u16(10000)

def detecta():
    parar()
    bocina()
```

```

bncs()
utime.sleep(0.3)
    blancas.value(0)
    rojas.value(0)
    buzzer.duty_u16(0)

# Melodía de inicio
def inicio():
    def playNote(frequency, duration, pause):
        buzzer.duty_u16(10000)
        buzzer.freq(frequency)
        time.sleep(duration)
        buzzer.duty_u16(0)
        time.sleep(pause)

    notes = [440, 494, 523, 587, 659, 698, 784]

    for note in notes:
        playNote(note, 0.1, 0.1)
        blancas.value(1)
        time.sleep(0.02)
        blancas.value(0)
        time.sleep(0.02)
        verdes.value(1)
        time.sleep(0.02)
        verdes.value(0)
        time.sleep(0.02)
        rojas.value(1)
        time.sleep(0.02)
        rojas.value(0)

# Sensor ultrasónico
def ultrasonido():
    trig.low()
    utime.sleep_us(2)
    trig.high()
    utime.sleep_us(10)
    trig.low()

    while echo.value() == 0:
        pulse_start = utime.ticks_us()

    while echo.value() == 1:
        pulse_end = utime.ticks_us()

    pulse_duration = utime.ticks_diff(pulse_end, pulse_start)
    distancia = pulse_duration * 0.0343 / 2
    return distancia

# Ejecutar la melodía de inicio en el segundo núcleo
_thread.start_new_thread(inicio, ())

```

```
# Bucle principal: escucha Bluetooth y sensor
while True:
    if modulo.any() > 0:
        dato = modulo.read(1).decode().strip()
        print("Dato recibido:", dato)

        if dato == "F":
            adelante()
        elif dato == "B":
            atras()
        elif dato == "R":
            izquierda()
        elif dato == "L":
            derecha()
        elif dato == "S":
            parar()
        elif dato == "W":
            bnca()
        elif dato == "w":
            blancas.value(0)
        elif dato == "U":
            ver()
        elif dato == "u":
            verdes.value(0)
        elif dato == "X":
            inicio()
        elif dato == "V":
            bocina()
        elif dato == "v":
            buzzer.duty_u16(0)

# Verificar obstáculo
distancia_actual = ultrasonido()
if distancia_actual < 15 and dato == "F":
    detecta()
```

