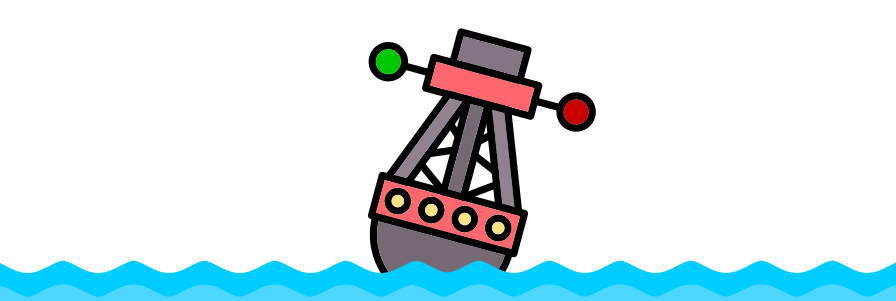
El trabajo con los microcontroladores, de una manera muy simplificada, se puede reducir a una gestión de señales: el microcontrolador recibe datos (las señales de entrada) y los procesa, convirtiéndolos en un tipo diferente de datos (las señales de salida). Las señales con que trabajan los microcontroladores pueden ser analógicas o digitales. En este capítulo nos centraremos en las digitales; en concreto veremos cómo “escribirlas”. Esta es una de las primeras habilidades que debemos dominar cuando trabajamos con microcontroladores ya que nos permitirá, entre otras cosas, algo tan fundamental como encender y apagar un dispositivo.

El material que usaremos, además de la placa, es:

* Dos ledes (uno verde y otro rojo, o de otros colores cualesquiera).
* Dos resistencias de 220 Ω.
* Una placa de pruebas o *breadboard*.
* Cables de conexión o *jumper wires*.

5.1 Proyecto “Señalización marítima”

En la navegación marítima las boyas de señalización son una guía esencial para los navegantes. Estas estructuras flotantes, situadas en puntos específicos de interés para las embarcaciones, delimitan canales de circulación, marcan zonas de peligro, indican áreas de baño… Para reforzar la señalización en zonas críticas, o para la navegación nocturna o en condiciones de baja visibilidad, algunas boyas llevan integrada una luz. Una luz verde indica el lado de estribor de un canal y una roja el lado de babor; estas señales luminosas, además, pueden ponerse en estado intermitente para indicar bifurcaciones en el canal.

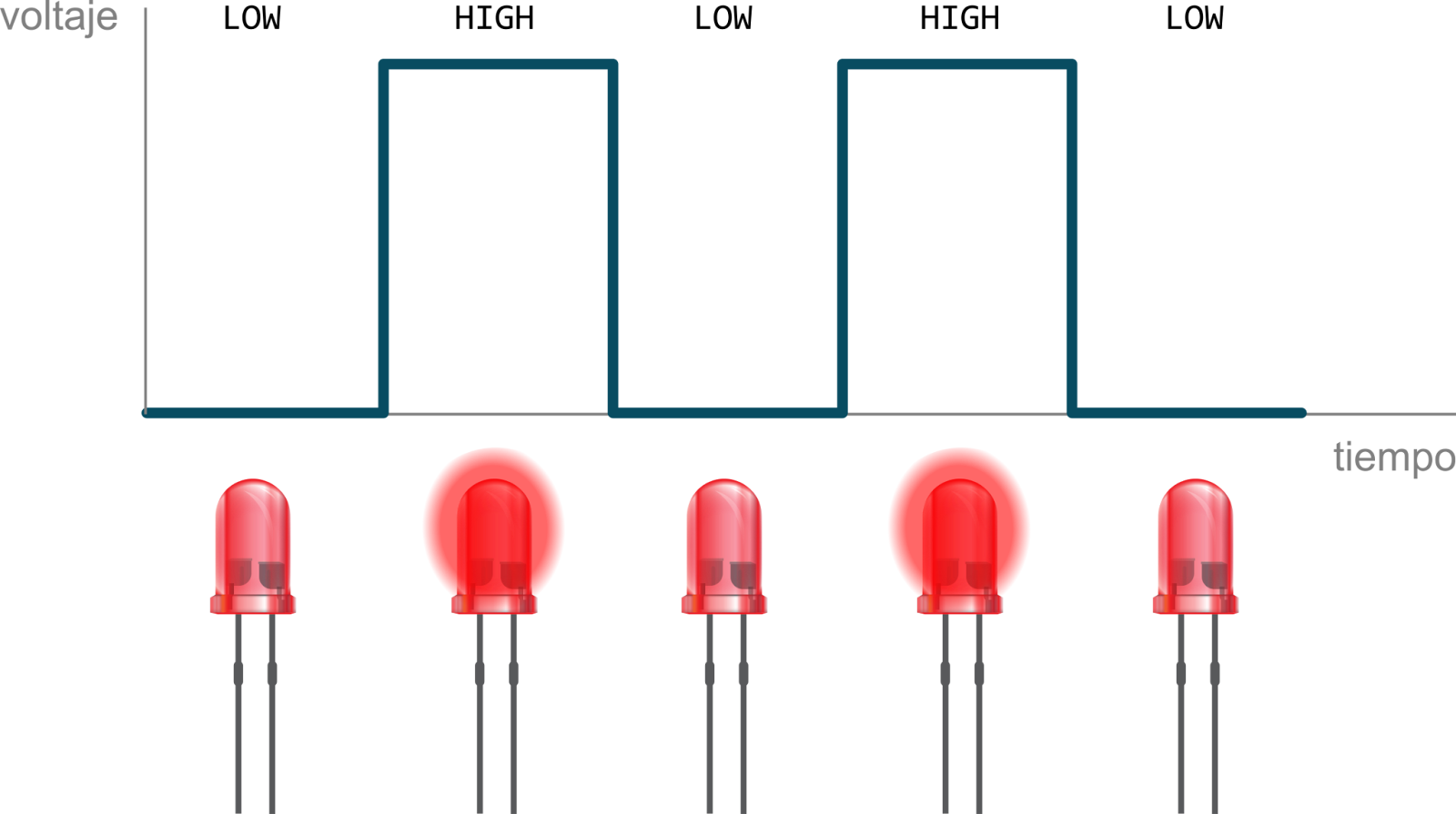


En este capítulo, utilizando dos ledes que se encenderán y apagarán de manera periódica, construiremos nuestro primer dispositivo electrónico: un prototipo de boya luminosa de señalización marítima.

5.2 Señales digitales

Dado que este capítulo está dedicado a las señales digitales, comenzaremos introduciendo brevemente estas señales tan especiales. En el mundo de los microcontroladores consideramos que una señal es digital si solo puede estar en dos estados diferentes: encendido –ON, HIGH o 1– o apagado –OFF, LOW o 0–. En una señal digital no hay valores intermedios posibles; es todo o nada.

El estado de un led es un buen ejemplo de señal digital. Cuando el led recibe un voltaje (conectándolo a una pila, por ejemplo) se enciende; cuando el voltaje se pone a cero, el led se apaga. Son las dos únicas posibilidades.



Ejemplo de señal digital. El voltaje solo puede estar en estado HIGH o LOW; análogamente, el led solo puede estar encendido o apagado.

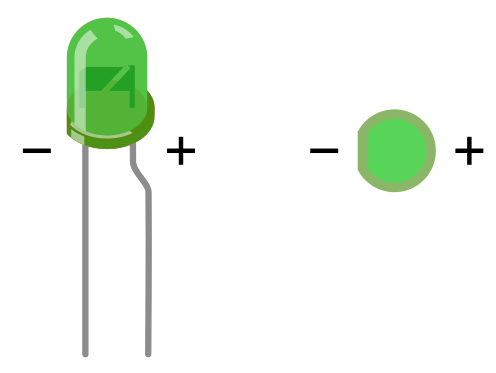
Aunque hemos dicho que el led únicamente puede estar encendido o apagado, en realidad sí lo podemos encender con un valor intermedio de la luminosidad, pero esto lo veremos más adelante porque ya no se trataría de usar una señal digital.

5.3 El diodo emisor de luz

Un diodo emisor de luz –o led, por sus siglas en inglés (de *Light Emitting Diode*)– es un dispositivo que emite luz cuando lo atraviesa una corriente eléctrica. El color de la luz generada depende del material semiconductor con que esté fabricado el led; aunque son muy habituales los rojos (de hecho, fue el primer color conseguido), hoy en día los podemos encontrar en multitud de colores.

Los diodos son dispositivos electrónicos que tienen polaridad, lo que quiere decir que la corriente circula a través de ellos en un único sentido. Sucede igual que con las pilas: si se conectan al revés, el aparato no funciona. En las pilas es fácil, ya que un polo está marcado como positivo (+) y el otro como negativo (−). Pero, en un led, ¿cómo los distinguimos?

Aunque parezcan iguales, una de las dos “patitas” del led es más larga. Pues bien, la más larga es el terminal positivo (o ánodo); la otra, obviamente, el negativo (o cátodo). Otra manera de distinguirlos es que, si nos fijamos bien, la carcasa plástica del led no es totalmente circular, sino que tiene una zona plana; el terminal adyacente a este corte plano es el negativo.

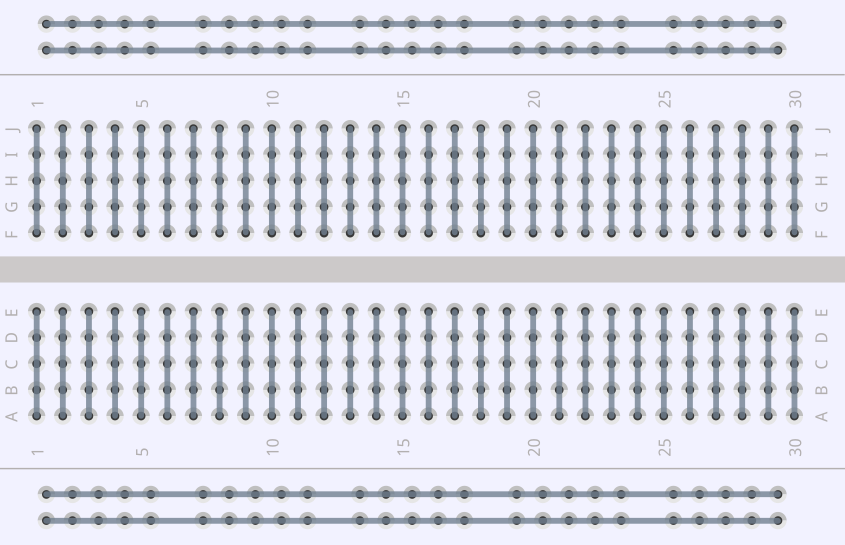


Izquierda: la pata más larga corresponde a la conexión positiva. Derecha: vista del led desde arriba; la parte plana en la carcasa indica la conexión negativa.

5.4 La placa de pruebas

Para facilitar la conexión entre los diferentes dispositivos, en todos los montajes que hagamos vamos a utilizar una placa de pruebas, también conocida con el nombre de tablero de prototipado, *protoboard* o *breadboard*. Sin entrar en detalles, la placa de pruebas consiste en un tablero de plástico que tiene una serie de orificios, organizados en filas y columnas, que están eléctricamente conectados. Estas conexiones eléctricas internas reducen el número de cables y facilitan enormemente la tarea de prototipado de los proyectos.

Aunque las placas de prueba se presentan en diferentes tamaños, la estructura interna es prácticamente la misma en todas ellas. En la imagen se señalan los orificios que están unidos eléctricamente.



5.5 El circuito

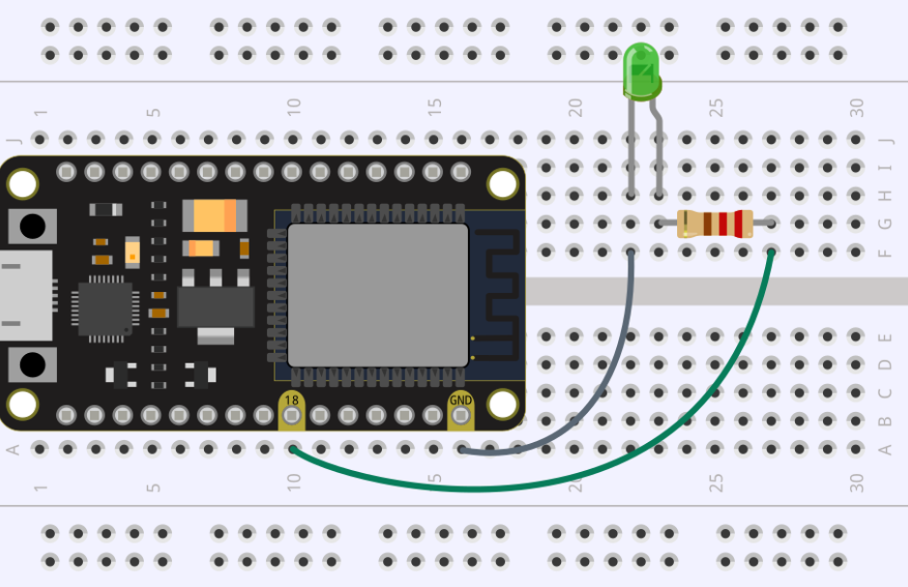
En capítulos anteriores hemos programado el microcontrolador con MicroPython, pero hasta ahora no habíamos añadido ningún componente de hardware externo. Ha llegado, por fin, el momento de conectar cosas a la placa. Empezaremos por conectar un único led.

El pin del microcontrolador al que conectemos el terminal positivo del led será el encargado de emitir las órdenes de encendido y apagado. Este pin, por tanto, funcionará como una salida digital. Hemos escogido el GPIO18 como pin de conexión –en el Arduino Nano ESP32 el GPIO18 recibe el nombre D9, mientras que en la Raspberry Pi Pico es el pin rotulado con el número 24–, pero se puede usar otro pin digital de la placa (en ese caso hay que cambiar en los programas el número del pin).

A su vez, en pin rotulado como GND (de *ground*, o tierra) cierra el circuito; es donde conectaremos el terminal negativo del led.

Una vez localizados los pines GPIO18 y GND en la placa, y con la placa desconectada del ordenador (los montajes se deben realizar siempre con la placa desconectada), se utiliza una placa de pruebas para realizar las siguientes conexiones:

* El terminal positivo del led (el ánodo) se conecta al GPIO18 de la placa.
* El terminal negativo del led (el cátodo) se conecta a GND.
* Se añade una resistencia de 220 Ω en serie con el led. Es indiferente si se conecta a través del terminal positivo o negativo del led; las resistencias no tienen polaridad.



¡Listo! Una vez finalizado el montaje conectaremos la placa al ordenador mediante el cable USB para poder manejar el led desde el entorno de programación.

5.6 Encender y apagar el led

Como comentamos cuando hablamos de los módulos integrados, las funciones relacionadas con el hardware de la placa están incluidas en el módulo machine. Nuestro primer contacto con este módulo será a través de la clase Pin, que contiene las funciones necesarias para controlar los pines de entrada y salida de la placa (los GPIO).

El primer objetivo que nos vamos a marcar es ser capaces de encender y apagar el led. Dado que este está conectado a un pin de la placa (en nuestro caso, el GPIO18), para encender el led simplemente debemos suministrar un voltaje en dicho pin; de manera análoga, para que el led se apague hay que anular el voltaje en el pin. Esto es equivalente a poner el pin en estado ON para encender el led o en estado OFF para apagarlo.

Bien, pero ¿cómo se le puede dar el valor deseado al estado del pin? Los pasos que hay que seguir son los que se recogen a continuación. No importa si por ahora no los entendemos muy bien; los explicaremos posteriormente con detalle. Ejecutemos las instrucciones en el REPL para ver su efecto.

Empezamos por importar la clase Pin del módulo machine.

>>> from machine import Pin

Se crea ahora una instancia del objeto Pin y se configura, estableciendo los valores de ciertas propiedades. A este objeto, que está asociado con el pin de la placa que queremos controlar, lo llamamos LED (se le puede poner cualquier otro nombre). Si el pin elegido para la conexión no es el GPIO18 se debe cambiar este número en la instrucción.

>>> LED = Pin(18, Pin.OUT)

Finalmente se especifica el valor de salida del pin. Aplicando a nuestro objeto el método on() se establece en ON el nivel lógico del pin.

>>> LED.on()

Si todo ha ido bien, al ejecutar esta última instrucción el led se habrá encendido. ¿Y si queremos apagarlo? Para ello recurrimos al método off(), que pone en OFF el estado del pin.

>>> LED.off()

Primer objetivo cumplido: podemos encender y apagar el led a nuestro antojo.

5.7 La clase Pin del módulo machine

Para entender mejor las instrucciones anteriores, en este apartado explicaremos con detalle cómo se crean y se utilizan los objetos Pin de MicroPython. Un objeto Pin (más rigurosamente deberíamos decir “una instancia del objeto Pin”) no es más que la representación, en lenguaje MicroPython, de un pin de la placa. Podemos decir que el objeto Pin es el equivalente en software al pin físico. En consecuencia, actuando sobre el objeto Pin en el programa lograremos controlar el pin de la placa.

El constructor

Para crear un objeto Pin se utiliza el constructor Pin(). En programación orientada a objetos, un constructor es un conjunto de instrucciones diseñado para inicializar instancias de objetos de una clase. Dicho de manera más simple, un constructor es un tipo especial de función que se utiliza para crear objetos y darles ciertos valores a sus propiedades. La sintaxis y los argumentos del constructor se muestran en la siguiente tabla.

|  |
| --- |
| Pin(id, mode=x, value=x) |
| Crea una instancia de un objeto Pin. Argumentos:   * id: es el primer argumento e indica el número del pin de la placa con el que se asocia el objeto. Es el único argumento obligatorio en el constructor. * mode: determina el modo –entrada o salida– en el que se configura el pin. Para un pin de salida (un pin en el que se suministra un voltaje) el valor es Pin.OUT; si se configura como pin de entrada toma el valor Pin.IN. Su posición está determinada en el constructor, después de id. * value: si el pin se ha configurado como salida (con Pin.OUT), este parámetro indica el valor inicial del voltaje suministrado en el pin. Toma el valor 0 o False si el pin se inicializa en estado apagado (0 V), y el valor 1 o True si se inicializa en estado encendido (3,3 V). Siempre es necesario poner el nombre de este parámetro. |

Los métodos

Los métodos son funciones que se pueden aplicar a un objeto Pin previamente creado para actuar sobre él y modificar su comportamiento. Para aplicar un método a un objeto se escribe el nombre del objeto seguido de un punto y a continuación el nombre del método.

A continuación se recogen los principales métodos de la clase Pin. No es un listado exhaustivo, sino que nos hemos limitado a los necesarios para el problema que nos ocupa (para más información se puede recurrir a la documentación del lenguaje MicroPython que está disponible *online*).

|  |  |
| --- | --- |
| init(mode=x, value=x) | Se configura el pin con los valores de mode y value suministrados. |
| value(x) | Se establece el valor lógico de un pin configurado como salida. Si el argumento es 0 o False se pone en estado apagado (0 V); si es 1 o True se pone en estado encendido (3,3 V). |
| value() | Sin argumento, devuelve el valor lógico en que se encuentra un pin: 0 si no hay ningún voltaje aplicado en el pin y 1 en caso contrario. |
| on() | Se establece el valor de salida 1 en el pin (3,3 V). Es equivalente a value(1). |
| off() | Se establece el valor de salida 0 en el pin (0 V). Es equivalente a value(0). |

Encendiendo y apagando el led

Volvamos al código con el que hemos encendido y apagado el led.

>>> from machine import Pin

>>> LED = Pin(18, Pin.OUT)

>>> LED.on()

>>> LED.off()

Después de importar la clase Pin hemos utilizado el constructor Pin() para crear un objeto Pin asociado al GPIO18. A este objeto le hemos dado el nombre LED y lo hemos configurado como pin de salida, Pin.OUT. Finalmente hemos puesto el pin en estado encendido aplicando el método on() al objeto LED, y lo hemos apagado con off().

Pues bien, existen varias maneras alternativas de conseguir lo mismo. Por ejemplo, podemos usar value(1) en lugar de on() para encender el led, y value(0) para apagarlo.

>>> LED.value(1)

>>> LED.value(0)

También podemos encender el led en el propio constructor.

>>> LED = Pin(18, Pin.OUT, value=1)

O podemos crear el objeto sin dar ningún valor a los parámetros y lo configuramos posteriormente.

>>> LED = Pin(18)

>>> LED.init(Pin.OUT, value=1)

Escogeremos una u otra en función de la funcionalidad del programa y de nuestras preferencias.

5.8 La configuración y el bucle

Hemos encendido y apagado el led escribiendo las instrucciones en el REPL y ejecutándolas una a una. Para conseguir un led intermitente, sin embargo, esta manera de proceder no es en absoluto adecuada; debemos escribir un programa que se encargue de encenderlo y apagarlo automáticamente. Pero, antes de proceder, vamos a analizar la estructura que utilizaremos habitualmente en la programación con microcontroladores: la configuración y el bucle.

La configuración

La configuración recoge las instrucciones que se ejecutan en primer lugar una única vez y lo preparan todo para funcionar correctamente. Si pretendemos conseguir que el led parpadee, debemos importar la clase Pin del módulo machine y la función sleep del módulo time para controlar el tiempo que el led estará encendido y apagado. Así mismo, crearemos un objeto Pin asociado al GPIO18 y lo inicializaremos como salida. Estas son las instrucciones de configuración.

El bucle

Una vez terminada la fase de configuración, normalmente habrá una serie de instrucciones que deseamos que se repitan una y otra vez. La manera más habitual de conseguirlo es usando el bucle infinito while True. Así, dentro del cuerpo del bucle escribiremos las instrucciones de encendido y apagado del led que se repetirá una y otra vez hasta que interrumpamos la ejecución del programa.

5.9 El led intermitente

Siguiendo la estructura que acabamos de comentar, una posible implementación del programa que hace parpadear el led de manera intermitente es la siguiente. Escribamos el programa en el editor y comprobemos que el led se comporta de la manera deseada. Cuidado con los espacios: en Python el cuerpo del bucle debe ir correctamente indentado –lo más habitual es poner cuatro espacios o un tabulador–, de lo contrario se obtiene un error.

from machine import Pin

from time import sleep

LED = Pin(18, Pin.OUT)

while True:

LED.on()

sleep(1)

LED.off()

sleep(1)

Al ejecutar el programa, el led se enciende de manera intermitente.

Una vez dentro del bucle, la ejecución del programa queda “atrapada” en su interior. Se puede parar la ejecución desde el editor pulsando el botón de Stop o simplemente desconectando la placa del ordenador.

5.11 El operador not

Una manera más compacta de escribir el programa de encendido y apagado del led es recurriendo al método value(). Como hemos visto, este método, sin argumento, lee el valor del pin y devuelve 1 si el pin está encendido o 0 si está apagado; con argumento, establece el valor lógico del pin. Por otro lado, el operador lógico not de Python (negación) convierte un 1 en un 0 y un 0 en un 1. Usando el operador not y el método value() obtenemos la siguiente versión del mismo programa; haciéndolo así necesitamos especificar el estado lógico inicial del led (por ejemplo con el parámetro value=0 en el constructor).

from machine import Pin

from time import sleep

LED = Pin(18, Pin.OUT, value=0)

while True:

LED.value(not LED.value())

sleep(1)

5.12 Cambiando la frecuencia

Si se quiere que el led parpadee más rápida o lentamente solo hay que cambiar el número de segundos dentro de la función sleep. Por ejemplo, para aumentar diez veces la frecuencia de parpadeo del led el tiempo de espera sería 0,1 s en lugar de 1 s. En tal caso, en vez de sleep utilizaremos sleep\_ms, poniendo el tiempo en milisegundos. Hemos añadido además la variable INTERVALO, cuyo valor es el tiempo de encendido y apagado; así, si se desea cambiar el intervalo de parpadeo solo hay que modificar el valor de esta variable.

from machine import Pin

from time import sleep\_ms

LED = Pin(18, Pin.OUT, value=0)

INTERVALO = 100

while True:

LED.value(not LED.value())

sleep\_ms(INTERVALO)

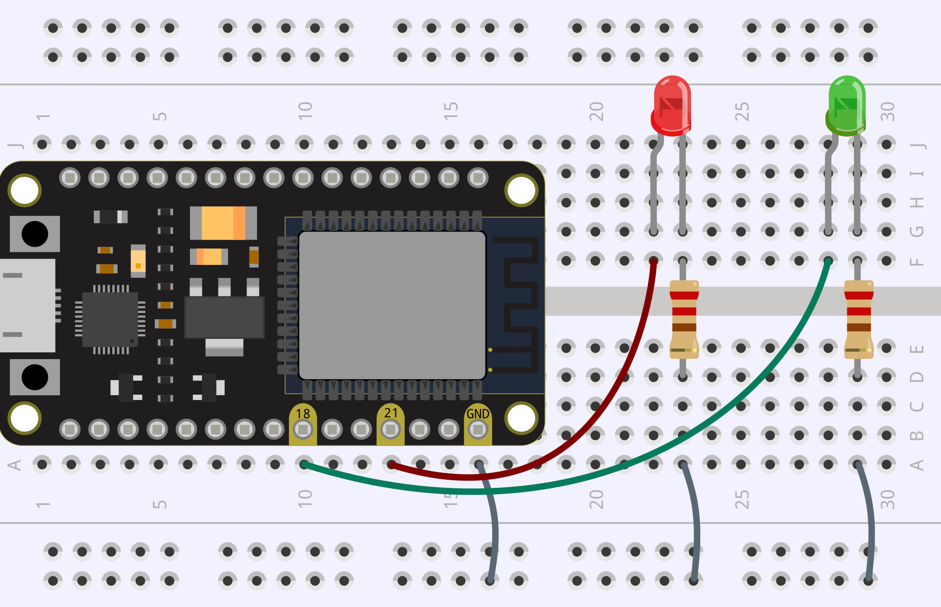
5.13 Encendiendo dos ledes

Sabemos ya cómo conectar y controlar un led. Sin embargo, la señal luminosa de las boyas que queremos simular implica el uso de dos ledes, uno verde y otro rojo. Empecemos con el montaje del circuito. Además del microcontrolador, la placa de pruebas y los cables de conexión usaremos dos ledes, uno verde y uno rojo, cada uno con su respectiva resistencia de 220 Ω.

Dado que debemos controlar dos ledes, necesitamos conectarlos a sendos GPIOs de la placa. Hemos escogido el GPIO18 para conectar el led verde y el GPIO21 para el led rojo (se pueden usar otros, en cuyo caso hay que cambiar los números en los programas). En la tabla siguiente se recoge el nombre que tienen estos pines en diferentes placas (en caso de duda se debe consultar la documentación).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Led | GPIO | Arduino | ESP32 | RaspberryPi Pico |
| Verde | GPIO18 | D9 | D18 / G18 | 24 |
| Rojo | GPIO21 | D10 | D21 / G21 | 27 |

El montaje con los dos ledes es totalmente análogo al de un único led; se puede utilizar la imagen como referencia. En este caso hemos usado el bus inferior de la placa de pruebas como conexión común a GND.



Conectamos ahora la placa al ordenador y para comprobar que el circuito está bien montado escribimos estas instrucciones en el REPL.

>>> from machine import Pin

>>> LED\_VERDE = Pin(18, Pin.OUT, value=1)

>>> LED\_ROJO = Pin(21, Pin.OUT, value=1)

Dado que en el constructor hemos puesto el parámetro value=1 deberían encenderse los dos ledes. Si alguno no se enciende se deben comprobar las conexiones.

5.13 Señal intermitente

Una vez que el circuito está correctamente montado vamos a programarlo para que se enciendan los dos ledes de manera intermitente. Primero hay que tomar ciertas decisiones sobre la manera en que van a parpadear. Por ejemplo, ¿cuál va a ser la frecuencia de parpadeo? ¿Esta frecuencia será siempre la misma? ¿Se encenderán y apagarán de manera simultánea o consecutivamente? Después de contestar estas preguntas, y dado que no hay ningún concepto de programación nuevo, animamos a los lectores a que intenten escribir el programa sin mirar la respuesta.

Mostramos una posible solución al problema planteado.

from machine import Pin

from time import sleep

LED\_VERDE = Pin(18, Pin.OUT, value=1)

LED\_ROJO = Pin(21, Pin.OUT, value=0)

# Tiempo entre destellos (s)

INTERVALO = 1

while True:

# Los ledes parpadean alternativamente

LED\_VERDE.value(not LED\_VERDE.value())

LED\_ROJO.value(not LED\_ROJO.value())

sleep(INTERVALO)

Con este programa hemos dado respuesta al reto de simular la señalización luminosa intermitente para los navegantes; como ejercicio proponemos variar la manera en la que las luces parpadean.

Llegados a este punto nos invade una sensación de orgullo muy grande al ver cómo se encienden y apagan los ledes, especialmente si es la primera vez que utilizamos un microcontrolador. Pero este programa se puede mejorar. Para descubrir cómo, sigamos leyendo.

5.14 Los *timers*

La función sleep que hemos usado para controlar el tiempo de encendido y apagado del led es una función bloqueante. Esto quiere decir que, mientras se está ejecutando, el programa se queda bloqueado; por tanto, durante el tiempo especificado el microcontrolador no puede realizar ninguna otra tarea. Nada. Lo cual, se mire como se mire, es un verdadero desperdicio de recursos. ¿Y si, en vez de estar de brazos cruzados esperando a que pase el tiempo, quisiésemos que el programa avanzase? La respuesta es que con sleep no se puede. La buena noticia es que hay diferentes maneras de controlar el tiempo sin que el microcontrolador se quede bloqueado. Una de ellas es utilizando *timers*.

Los *timers* son mecanismos del hardware de la placa que se ocupan de gestionar períodos de tiempo. Aunque son elementos que varían mucho entre diferentes placas (de hecho, el ESP32 tiene cuatro *timers* mientras que la Raspberry Pi pico solo dispone de uno), MicroPython ofrece funciones básicas que son comunes a todas ellas.

Las funciones para controlar este elemento están en la clase Timer del módulo machine. Mostramos a continuación la sintaxis del constructor.

|  |
| --- |
| Timer(id) |
| Crea una instancia de un objeto Timer. El argumento id es el número del *timer* de la placa con el que se asocia el objeto. Puede tomar los siguientes valores, dependiendo de la placa:   * En ESP32 y Arduino indica el timer que se está utilizando: 0, 1, 2… Si se usa el timer virtual (de software), el valor será −1. * En la Raspberry Pi Pico se debe usar el valor −1, o bien se usa el constructor sin argumento. |

Los métodos básicos que se pueden aplicar a un objeto Timer son los siguientes:

|  |  |
| --- | --- |
| init(mode=x, period=x, callback=x) | Se configura el objeto Timer. Argumentos:   * period: El período del *timer*, en milisegundos. * mode: Puede tomar los valores Timer.ONE\_SHOT (el *timer* se ejecuta una sola vez, cuando haya transcurrido el tiempo indicado en period) o Timer.PERIODIC (el *timer* se ejecuta de manera periódica cada vez que finaliza el tiempo indicado en period). * callback: Función que se ejecuta al finalizar el tiempo indicado en period. En su definición, esta función debe llevar necesariamente un argumento, y uno solo.   Los argumentos deben ir precedidos por el nombre del parámetro, por tanto se pueden situar en el orden que se desee. |
| deinit() | Para el *timer* y lo desactiva. |

Reescribamos el programa del parpadeo de los dos ledes, pero esta vez controlando el tiempo con *timers*. Para ello el encendido y apagado lo debemos poner dentro de una función, a la que hemos llamado parpadeo(), que será la función que se ejecute cada vez que pase el tiempo indicado en el período del *timer*.

Si trabajamos con las placas ESP32 o Arduino se puede usar por ejemplo el *timer* 0, por lo que la sintaxis del constructor sería Timer(0); sin embargo, con la Raspberry Pi Pico la instrucción debe ser Timer(). Otra opción es recurrir a un *timer* virtual (no está disponible en todas las placas), manejado por software, en cuyo caso la instrucción sería Timer(-1). Se debe poner en el programa el *timer* adecuado a la placa con la que estemos trabajando.

from machine import Pin, Timer

# Ledes

LED\_VERDE = Pin(18, Pin.OUT, value=1)

LED\_ROJO = Pin(21, Pin.OUT, value=0)

# Se usa el timer 0

TIMER = Timer(0)

# Intervalo de parpadeo (ms)

INTERVALO = 500

def parpadeo(t):

# Los ledes parpadean alternativamente

LED\_VERDE.value(not LED\_VERDE.value())

LED\_ROJO.value(not LED\_ROJO.value())

# Se inicializa el timer

TIMER.init(period=INTERVALO, mode=Timer.PERIODIC, callback=parpadeo)

El *timer* que hemos creado, al que hemos llamado TIMER, llama a la función parpadeo de manera periódica cada vez que pasan los milisegundos indicados en period.

Si se ejecuta el programa, los ledes parpadean igual que antes, pero hemos evitado el uso de la función bloqueante sleep. De hecho, si nos fijamos en el editor, mientras los ledes parpadean el *prompt* del REPL (>>>) está activo, cosa que no sucedía con la versión anterior del programa.

Para conseguir que los ledes dejen de parpadear pulsamos Stop en el editor o llamamos en en el REPL al método deinit() para desactivar el *timer*.

>>> TIMER.deinit()

5.15 Último paso: subirlo a placa

Aunque ya tenemos un prototipo funcional de boya luminosa intermitente, hay un problema muy obvio: el programa lo estamos ejecutando desde el entorno de programación en nuestro ordenador y eso es, cuando menos, poco deseable. Una característica esperable en la gran mayoría de proyectos con microcontroladores es que sean autosuficientes, en el sentido de que no necesiten estar conectados a un ordenador para funcionar.

Los programas con los que hemos trabajado hasta ahora estaban ubicados en el disco duro del ordenador y los ejecutábamos desde el editor, por eso era imprescindible que la placa estuviese conectada al ordenador. Pues bien, el primer paso para la autonomía del proyecto es guardar el programa en la placa, para que se ejecute desde allí.

La manera de guardar el programa a la placa la hemos visto en un capítulo anterior, y consiste en subir el programa a la placa con el nombre main.py. Haciéndolo así el programa se ejecutará desde el propio microcontrolador, con lo que ya se puede desconectar del ordenador. Eso sí, ahora la placa ya no está recibiendo algo que es imprescindible para su funcionamiento: corriente. Se necesita algo que la alimente; lo más rápido y cómodo es conectarla a un *power bank* como los que se utilizan para cargar el móvil. Y, ahora sí, el proyecto es totalmente independiente y podemos llevarlo a donde queramos.

En resumen

En este capítulo nos hemos centrado en la escritura de señales digitales, uno de los aspectos básicos de la programación de microcontroladores. Hemos conectado un led a la placa y, para controlarlo, hemos tenido nuestro primer contacto con el módulo machine de MicroPython a través de la clase Pin. Hemos analizado el flujo habitual de trabajo en un microcontrolador, distinguiendo las instrucciones de configuración de las que se repiten en bucle. Con el bucle while True hemos escrito instrucciones cuya ejecución se repite sin parar. Nos hemos ayudado de las funciones sleep y sleep\_ms para controlar el tiempo y hemos introducido los *timers* como una herramienta alternativa a estas funciones. Finalmente hemos subido el programa a la placa para poder instalar donde deseemos la boya luminosa que hemos fabricado.