En los capítulos anteriores hemos tocado los tres pilares de la programación de microcontroladores: hemos hablado del lenguaje de programación, hemos presentado la placa que utilizaremos en los proyectos y hemos introducido el entorno de programación con que escribiremos los programas que se ejecutarán en la placa. Este último capítulo de introducción lo dedicaremos a un elemento que vamos a utilizar en todos nuestros proyectos con MicroPython: los módulos integrados. No añadiremos por ahora elementos de hardware externos (eso llegará a partir del próximo capítulo); de esta manera no corremos el riesgo de perdernos entre cables y resistencias y podremos mantener el foco en la placa y el lenguaje. En concreto trabajaremos con el módulo time, que proporciona funciones que vamos a usar con mucha frecuencia en los proyectos posteriores.

4.1 Los módulos integrados

Como ya hemos comentado previamente, el lenguaje MicroPython trae incorporadas una serie de librerías que amplían la sintaxis básica del lenguaje. Algunas son librerías estándar iguales que las de Python; otras son también librerías de Python pero que han sido “miniaturizadas”, y en otros casos son librerías específicas de MicroPython cuyo cometido es facilitar la interacción con el microcontrolador. Pues bien, las funciones y clases implementadas en estas librerías están recogidas en los llamados módulos integrados (en inglés, *built-in modules*).

Los módulos integrados

Veamos en primer lugar cuáles son los módulos disponibles en la distribución de MicroPython de nuestro microcontrolador. Para interactuar con la placa seguimos los siguientes pasos, tal y como se indicó en el capítulo dedicado al entorno de programación:

* Abrimos el entorno de programación (Thonny).
* Conectamos la placa al ordenador con el cable USB.
* Comprobamos que el intérprete que aparece a la derecha en la barra inferior de Thonny sea el adecuando para nuestra placa (MicroPython ESP32, Raspberry Pi Pico o genérico). Si no es así, lo escogemos en el menú desplegable que aparece al hacer clic en las tres barras horizontales.
* Si la placa no se conecta automáticamente pulsamos el botón Stop para reiniciarla.

Una vez que la placa está conectada escribimos help("modules") en el REPL.

>>> help("modules")

Al pulsar Intro, aparece una lista parecida a la siguiente (solo incluimos un extracto).

\_\_main\_\_ bluetooth heapq select

\_asyncio btree inisetup socket

\_boot builtins io ssl

\_espnow cmath json struct

\_thread math machine time

\_webrepl deflate micropython tls

...

Plus any modules on the filesystem

El listado anterior recoge todos los módulos que vienen instalados por defecto en nuestra distribución de MicroPython.

En la inmensa mayoría de los programas que escribamos en MicroPython (por no decir en todos) vamos a recurrir a alguno de estos módulos. Tomemos como ejemplo uno con el que trabajaremos en innumerables ocasiones a lo largo del libro: el módulo machine. El nombre ya nos da una idea del tipo de funciones que nos encontraremos en él: contiene funciones específicas relacionadas con el hardware de una placa determinada. Esto quiere decir que para controlar cualquier elemento de hardware usaremos las funciones que se incluyen en este módulo. Análogamente, el módulo time proporciona las funciones necesarias para trabajar con intervalos temporales, os ofrece servicios básicos relacionados con el sistema operativo, math contienen funciones matemáticas y network facilita la configuración de red, entre otros muchos.

Los módulos externos

Al final del listado de los módulos integrados aparecía la frase Plus any modules on the filesystem. Esto se debe a que existe otro tipo de librerías, que no vienen “de fábrica” en la instalación de MicroPython, cuya misión es facilitarnos el trabajo cuando utilicemos elementos como sensores o pantallas. Dado que estas librerías sirven a propósitos muy específicos, por motivos de espacio no vienen incluidas en el propio *firmware*, y para usarlas hay que instalarlas manualmente en el sistema de archivos de la placa. De ahí que, además de los módulos de la lista, podemos tener otros instalados. Pero de estos módulos externos no nos ocuparemos por ahora.

4.2 ¿Qué hay en un módulo?

Exactamente, ¿qué hay en un módulo? Analicemos el módulo math. Los módulos integrados, como hemos visto, se instalaron en la placa en el momento en que se instaló el *firmware* de MicroPython. Para poder acceder a ellos hay que importarlos, que es algo así como cargarlos en la memoria del microcontrolador para que estén disponibles. Para importar un módulo escribimos en el REPL la palabra import seguida del nombre del módulo que queremos importar. Por ejemplo, importemos el módulo math.

>>> import math

Al ejecutar esta instrucción parece que no ha sucedido nada. En realidad, al importar el módulo se han cargado las funciones, clases y constantes que contiene, para que las podamos usar en los programas que escribamos. El módulo estará disponible a lo largo de toda la sesión, mientras no se reinicie la placa.

Una vez importado el módulo, escribimos en el REPL help(math).

>>> help(math)

object <module 'math'> is of type module

\_\_name\_\_ -- math

e -- 2.718282

pi -- 3.141593

inf -- inf

sqrt -- <function>

pow -- <function>

exp -- <function>

log -- <function>

...

Como se puede ver en el listado anterior, el módulo contiene funciones que se encargan de realizar tareas específicas: sqrt para calcular raíces cuadradas, log para hallar logaritmos, etc. Hay también definidas constantes, como los números π y e. Finalmente un módulo también puede contener clases, que sirven para agrupar objetos con características similares y que, a su vez, contienen sus propios atributos (propiedades) y métodos (funciones).

4.3 Cómo usar los módulos

Para usar una función o una constante definida en un módulo se puede proceder de dos maneras diferentes: importando el módulo completo o importando únicamente el elemento en cuestión.

Importando todo el módulo

Primero se importa el módulo completo, escribiendo la palabra import seguida del nombre del módulo a importar.

>>> import math

Para usar una función (por ejemplo la función raíz cuadrada) de este módulo se escribe el nombre del módulo y a continuación, separado por un punto, el nombre de la función. El nombre de la función debe ir seguido de paréntesis (aunque pueden estar vacíos).

>>> math.sqrt(2)

1.414214

Si lo que se quiere usar es una constante (como puede ser el número pi) se procede de manera análoga, pero esta vez no se utilizan paréntesis.

>>> math.pi

3.141593

Cada vez que usemos una función o constante, esta debe ir precedida del nombre del módulo al que pertenece. Si nos olvidamos de añadir el nombre del módulo obtenemos un error.

>>> math.log(e)

NameError: name 'e' isn't defined

>>> math.log(math.e)

1.0

Importando solo los elementos necesarios

En lugar de importar todo el módulo podemos importar únicamente los elementos que vayamos a utilizar. Para ello se usa la siguiente sintaxis: from, seguido del nombre del módulo, import, seguido del nombre del elemento a importar; se pueden poner varios elementos, separados por comas. El nombre de las funciones se escribe sin los paréntesis.

>>> from math import sqrt, pi

Para usar uno de estos elementos, simplemente se escribe su nombre; no hay necesidad de hacer referencia al módulo.

>>> sqrt(2)

1.414214

>>> pi

3.141593

Importar las funciones de una u otra manera es una decisión personal. Eso sí, es importante recalcar que solo podemos usar aquellas funciones, clases o constantes que hayamos importado. Si intentamos utilizar alguna que no haya sido importada previamente se obtendrá un error. Por otro lado, si estamos trabajando en el REPL y desconectamos la placa, al volverla a conectar es necesario volver a importar los módulos con que vayamos a trabajar.

4.4 El módulo time

En prácticamente todos los programas de este manual vamos a necesitar llevar cuenta del tiempo. Por ello, y para poner en práctica lo aprendido sobre los módulos, el resto del capítulo lo vamos a dedicar al módulo time. Dentro este módulo se engloban, entre otras, funciones para medir intervalos de tiempo y para establecer tiempos de espera. En la siguiente tabla recogemos algunas de las funciones más importantes del módulo.

|  |  |
| --- | --- |
| ticks\_ms() | Devuelve el número de milisegundos pasados, con respecto a un instante arbitrario. |
| ticks\_us() | Igual que ticks\_ms(), pero en microsegundos. |
| ticks\_diff(t1, t2) | Mide la diferencia entre t1 y t2. Es equivalente a t1 − t2. |
| sleep(tiempo) | Interrupción durante el tiempo indicado, en segundos. Algunas placas solo aceptan valores enteros como argumento. |
| sleep\_ms(tiempo) | Igual que la anterior, pero en milisegundos. |
| sleep\_us(tiempo) | Igual que las anteriores, pero en microsegundos. |

4.5 La función sleep()

Hay una función relacionada con el tiempo que usaremos con mucha frecuencia a lo largo de este manual: sleep(). La función sleep(), junto con sleep\_ms() y sleep\_us(), se utilizan para introducir esperas o retardos en el código. Las tres actúan igual, solo cambia la unidad en que se mide el tiempo: segundos, milisegundos y microsegundos, respectivamente.

Cuando el programa entra en una de estas funciones, la ejecución se queda bloqueada durante el tiempo que se estableció en el argumento. Hagamos la prueba: veamos qué sucede al ejecutar la función sleep(5) en el REPL.

>>> from time import sleep

>>> sleep(5)

De manera alternativa, y solo por recordar cómo se utilizan las funciones de los módulos, podemos importar el módulo completo, en cuyo caso debemos indicar el nombre del módulo antes de la función.

>>> import time

>>> time.sleep(5)

De una manera u otra, al pulsar Intro el prompt (>>>) desaparece. Durante cinco segundos no sucede nada; pasado este tiempo reaparece. Por este motivo se dice que sleep() es una función bloqueante: cuando se ejecuta esta instrucción el programa se queda bloqueado esperando a que pase el tiempo indicado, y mientras tanto no se ejecuta ninguna otra instrucción.

Una importante característica que hay que hacer notar es que el tiempo de espera es aproximado; si el procesador necesita ocuparse de otras tareas que tienen más prioridad (como gestionar interrupciones, por ejemplo), el tiempo de espera puede ser algo mayor que el indicado.

4.6 Cuenta atrás

Hasta este momento, todas las instrucciones las hemos ejecutado desde la consola. Hacerlo así ayuda a entender cómo funciona cada instrucción, pero llega un momento en que esta manera de trabajar se vuelve poco práctica. Lo que haremos ahora es escribir un programa en el editor de *scripts* y ejecutaremos este programa, en lugar de ir instrucción a instrucción.

Para ilustrar el uso de sleep() vamos a escribir un programa muy simple: una cuenta atrás. El programa imprimirá los números 5, 4, 3, 2, 1, 0 con un intervalo de un segundo entre ellos. Escribimos el siguiente programa en el editor de *scripts* y pulsamos la tecla Run para ejecutarlo.

from time import sleep

i = 5

while i >= 0:

print(i)

sleep(1)

i -= 1

La variable i es el contador que inicializamos en 5. Cuando la ejecución entra en el bucle while se imprime el valor del contador y se espera un segundo; a continuación el contador se reduce en uno. Estas instrucciones se volverán a ejecutar mientras el contador sea mayor o igual que cero. En consecuencia, se muestra en el REPL la cuenta atrás esperada.

Visto así parece poca cosa, pero más adelante podremos añadir un pulsador para iniciar la cuenta atrás y “adornarla” con pitidos o señales luminosas para darle un toque muy profesional.

4.7 La función ticks\_ms()

Cuando se necesita medir tiempos con precisión es muy habitual recurrir a la función ticks\_ms(). Esta función accede a un contador de tiempos y devuelve el número de milisegundos que han pasado desde que dicho contador se puso en marcha (lo mismo sucede con ticks\_us(), pero en microsegundos). El inicio del contador es arbitrario, es decir, no tenemos referencia de cuándo se puso en marcha. Esto, que parece un problema, en realidad no nos preocupa, como veremos enseguida. Lo importante es que cada vez que se ejecuta ticks\_ms() se accede al valor que marca el contador en ese instante.

Hagamos una prueba. Con la placa conectada, importamos la función ticks\_ms(), que está en el módulo time, y la ejecutamos.

>>> from time import ticks\_ms

>>> ticks\_ms()

273105

El valor que aparece es el número de milisegundos que han pasado desde que se inició el contador hasta el momento en que se ejecutó ticks\_ms(). Si la ejecutamos de nuevo obtendremos un valor mayor porque ha pasado más tiempo.

>>> ticks\_ms()

288435

El inicio del contador está relacionado con el momento en que se conectó la placa. Si se reinicia la placa y se vuelve a ejecutar ticks\_ms() (en ese caso es necesario volver a importar la función), comprobamos que el contador se ha reinicializado.

4.8 Un cronómetro

En el trabajo con microcontroladores muchas veces necesitamos controlar el tiempo transcurrido entre dos sucesos, así que vamos a usar la función ticks\_ms() para programar una especie de cronómetro, muy rudimentario pero que nos servirá para sentar las bases de cómo medir el tiempo de manera muy precisa.

Los cronómetros no son más que relojes que miden con precisión el tiempo transcurrido entre dos instantes, desde el momento en que se ponen en marcha hasta que se paran. A grandes rasgos, nuestro cronómetro funcionará de la siguiente manera. La función ticks\_ms() –o ticks\_us() en el caso de que necesitemos trabajar con más precisión– se ejecutará en dos instantes diferentes: el de “inicio” (cuando se pone en marcha el cronómetro) y el de “fin” (cuando se para). Para saber cuánto tiempo ha pasado entre esos instantes, el intervalo transcurrido será, simplemente, la diferencia entre el tiempo marcado por ticks\_ms() en ambos instantes; para hallar esa diferencia se usaremos la función ticks\_diff(). Y eso es todo.

Para ilustrarlo importamos las funciones (si ya hemos importado previamente la función ticks\_ms() no es necesario volver a hacerlo).

>>> from time import ticks\_ms, ticks\_diff

Determinaremos los instantes inicial y final con ticks\_ms(). Cada vez que llamemos a esta función debemos guardar el valor devuelto en una variable para poder utilizarlo posteriormente. Escribimos la siguiente instrucción en el REPL.

>>> inicio = ticks\_ms()

En el momento que hemos pulsado Intro, ticks\_ms() ha leído el valor del contador y lo ha adjudicado a la variable inicio (podemos ponerle cualquier otro nombre). Este es nuestro instante inicial.

Escribimos ahora la instrucción para el instante final.

>>> fin = ticks\_ms()

Igual que antes, el instante final, que corresponde al momento en que se ejecuta la instrucción anterior, está guardado en la variable fin. Podemos ver el valor de estas variables sin más que escribir su nombre en el REPL.

>>> inicio

824507

>>> fin

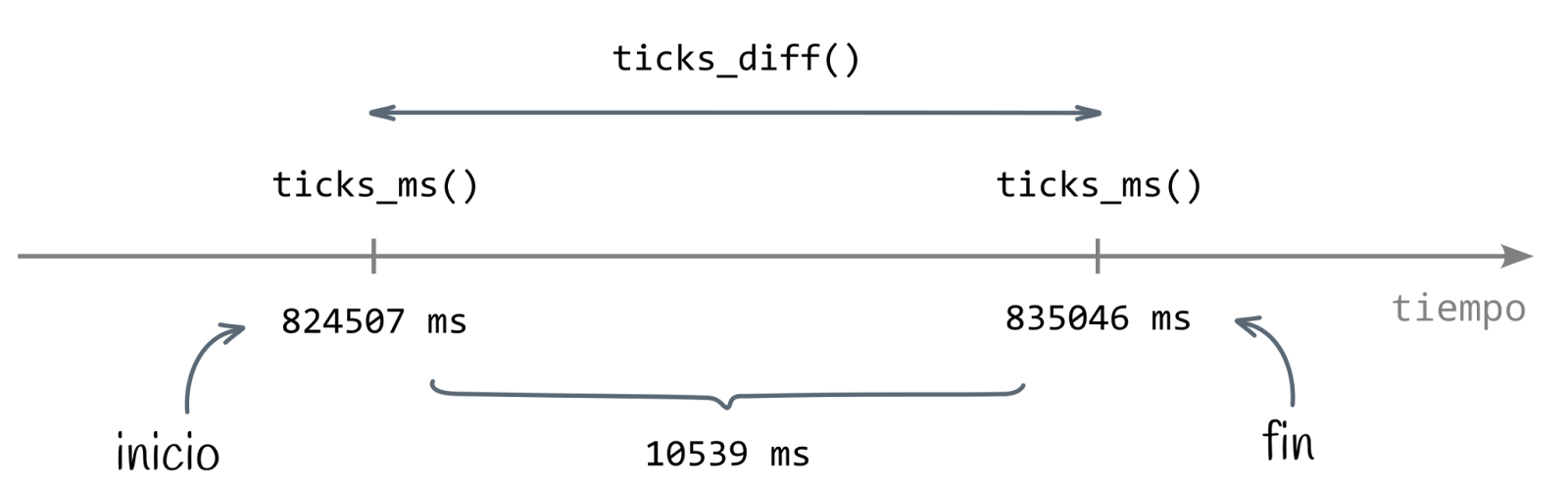
835046

Por último, para saber cuánto tiempo ha pasado entre un instante y otro usamos la función ticks\_diff(). Esta función actúa de la siguiente manera: si t1 y t2 son los dos valores devueltos por ticks\_ms(), entonces ticks\_diff(t1, t2) calcula el valor de t1 − t2. En consecuencia, si t1 ocurrió antes que t2 la función devolverá un resultado negativo (porque t1 es menor que t2), y si t1 ocurrió después de t2 entonces el resultado es positivo (ya que ahora t1 es mayor que t2). Dado que sabemos con certeza que inicio fue antes que fin, para obtener un intervalo de tiempo positivo nos interesa hacer la operación fin − inicio. Por tanto, la instrucción a ejecutar es la siguiente:

>>> ticks\_diff(fin, inicio)

10539

Es decir, han pasado exactamente 10 539 milisegundos –o 10,539 segundos– entre la ejecución de las dos instrucciones ticks\_ms().



4.7 El cronómetro, mejorado

Para tener un cronómetro funcional, en lugar de ejecutar las instrucciones desde el REPL vamos a escribir un programa.

Para poder iniciar y para el cronómetro necesitamos interactuar con el programa. Aunque más adelante podremos hacerlo de maneras mucho más sofisticadas, por ahora vamos a conformarnos con utilizar el teclado. Para ello usaremos la función input() de Python. Simplificando mucho su uso, cuando se ejecuta input() el programa queda a la espera de que el usuario pulse la tecla Intro en el teclado. Una vez pulsada, el programa continúa. Eso es todo lo que necesitaremos para poner en marcha y parar el cronómetro.

Algo importante a tener en cuenta es que, para que el entorno de programación detecte que se ha pulsado Intro, primero hay que hacer clic sobre la consola (el REPL) con el ratón para activarla.

Escribamos el siguiente programa en el editor. Para ejecutarlo hacemos clic en la tecla Run.

from time import ticks\_ms, ticks\_diff

# Inicio

input()

inicio = ticks\_ms()

print("Instante inicial:", inicio)

# Fin

input()

fin = ticks\_ms()

print("Instante final:", fin)

# Tiempo

tiempo = ticks\_diff(fin, inicio)

print("\nTiempo (s):", tiempo/1000)

El programa empieza por importar las funciones necesarias del módulo time. Cuando el programa llega al primer input() se queda a la espera de que se pulse Intro. En cuanto se pulsa, se guarda el instante inicial en inicio y se muestra su valor. A continuación el programa entra en el siguiente input(). De nuevo se queda esperando y, una vez pulsado Intro, se guarda el instante final en fin. Se calcula el tiempo que ha pasado y se divide entre mil para mostrarlo en segundos.

Aunque se trata de un cronómetro muy limitado, este programa nos ha servido para mostrar cómo medir intervalos de tiempo de una manera muy precisa.

En resumen

En este capítulo hemos presentado los módulos integrados de MicroPython, que facilitan enormemente la programación gracias a las funciones que ponen a nuestra disposición. En particular hemos trabajado con el módulo time, ya que será muy habitual que trabajemos con él en los proyectos que desarrollemos más adelante. Usando este módulo hemos diseñado un cronómetro para medir con gran precisión el tiempo transcurrido entre dos sucesos y hemos programado una sencilla cuenta atrás.