Una aplicación muy extendida de los microcontroladores es la toma de datos mediante sensores. En el capítulo anterior, por ejemplo, programamos un sensor de temperatura. Los datos medidos los leímos en el REPL, que no está mal para empezar pero tiene sus limitaciones. Uno de los inconvenientes de esta forma de trabajar es que necesitamos el ordenador para ver los datos. Este problema tiene fácil solución: conectemos una pequeña pantalla al microcontrolador y ya nos lo podemos llevar a donde queramos. Pero esta solución no resuelve otro problema. Imaginemos que deseamos monitorizar la temperatura de un parque infantil en verano para estudiar posibles medidas de protección ante el calor. No sería nada práctico tener que estar leyendo la temperatura en la pantalla. Ese es precisamente el problema que queremos solucionar en este capítulo.

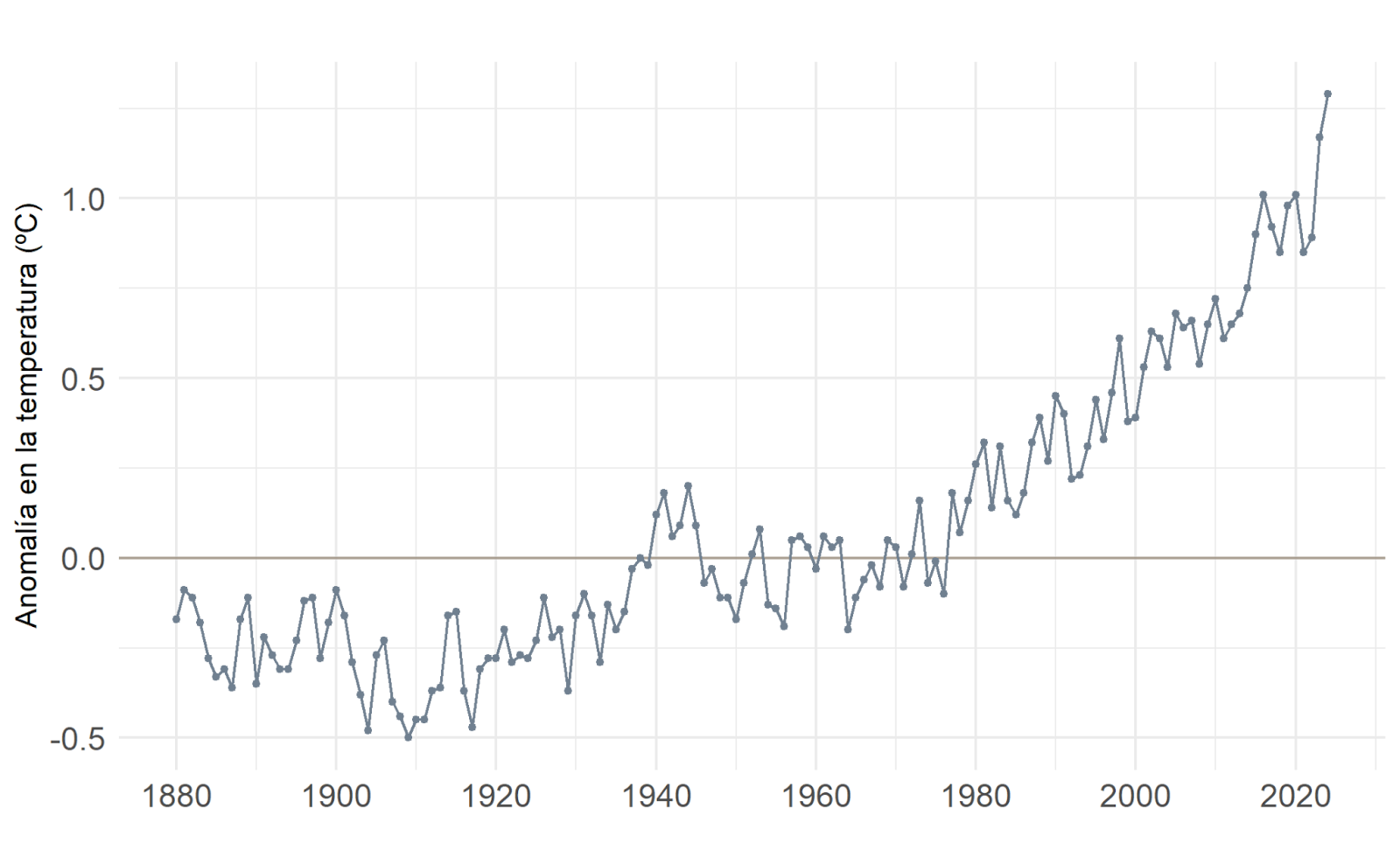
Una posible solución (aunque no es la única) consiste en guardar los datos en el dispositivo a medida que se van obteniendo. La manera “tradicional” de hacerlo pasa por añadir un componente externo, como un lector de tarjetas SD. Las placas con las que estamos trabajando, sin embargo, nos proporcionan una manera de almacenar información de manera persistente sin necesidad de añadirles nada. Para aprovecharnos de esta funcionalidad, en este capítulo aprenderemos a utilizar la memoria *flash* del microcontrolador, donde generaremos un archivo con los datos medidos por el sensor que podrá ser posteriormente descargado para su análisis.

En cuanto al material, no se introduce ningún elemento de hardware nuevo; simplemente retomaremos el montaje del sensor de temperatura TMP36 del capítulo anterior.

10.1 Proyecto “*Data logger*”

«Un registrador de datos o *data logger* es un dispositivo electrónico que registra datos por medio de instrumentos y sensores propios o conectados externamente. Casi todos están basados en microcontroladores. Por lo general son pequeños, con pilas, portátiles, y equipados con un microprocesador, memoria interna para almacenamiento de datos y sensores.» (Fuente: Wikipedia).

El Sistema Mundial de Observación del Clima –o *Global Climate Observing System*, GCOS– se creó en 1992 para evaluar periódicamente el estado de las observaciones climáticas mundiales y elaborar directrices para su mejora. Aunque el trabajo del GCOS se apoya en observaciones y procedimientos muy sofisticados para evaluar el estado climático de la atmósfera, la tierra y el océano, el proyecto que proponemos en este capítulo es un punto de partida para adentrarnos en el estudio científico de las variables esenciales relacionadas con el cambio climático: construiremos un *data logger* con el que monitorizaremos la evolución de la temperatura a lo largo del tiempo. Los datos medidos por el sensor los guardaremos en un archivo para, posteriormente, analizarlos y poder sacar conclusiones.



En 2024 la temperatura global en la superficie de la Tierra fue 1,47 ºC superior que en el promedio preindustrial de finales del siglo XIX (1850-1900). Datos: Goddard Institute for Space Studies (GISS), NASA.

10.2 El módulo os

Cuando presentamos los microcontroladores hablamos del sistema de archivos de la placa –el *filesystem*–, que está alojado dentro de la memoria *flash* del dispositivo. Dado que en este capítulo vamos a utilizar esa memoria para guardar un archivo con los valores medidos por un sensor, necesitamos acceder al sistema de archivos de la placa desde el propio programa, y no desde el entorno de programación, como hemos hecho hasta ahora. Es el momento de presentar otro de los módulos integrados de MicroPython: el módulo os.

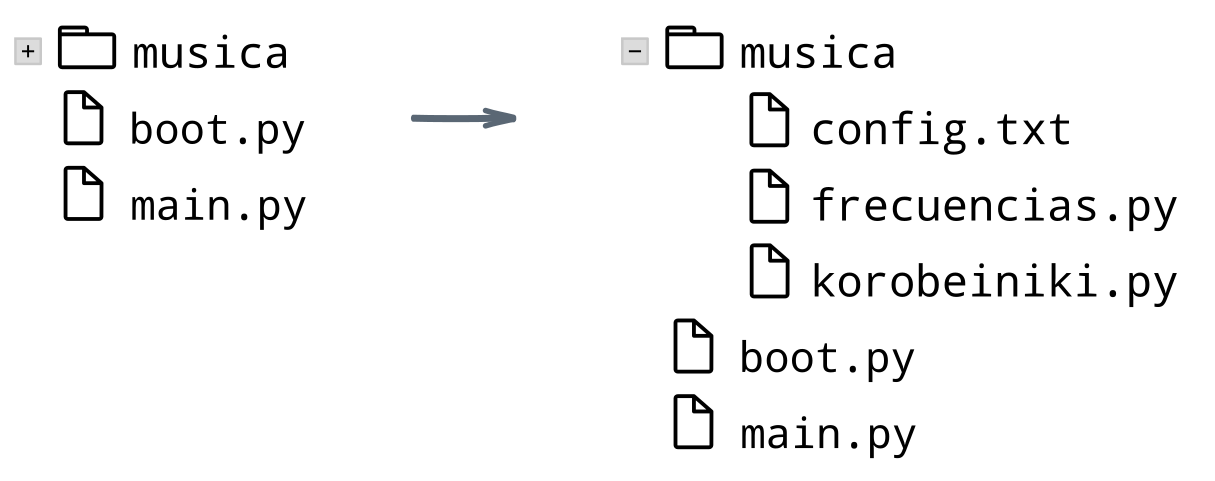
El módulo os proporciona servicios básicos relacionados con el sistema operativo. Entre otras, contiene funciones que permiten el acceso al sistema de archivos de la placa, que es justo lo que necesitamos. En la siguiente tabla recogemos algunas funciones de este módulo, que nos permitirán manejar los archivos –o ficheros– y las carpetas –o directorios– desde el código.

|  |  |
| --- | --- |
| listdir() | Devuelve una lista con el contenido –archivos y carpetas– del directorio actual (por defecto este directorio es el directorio raíz del *filesystem*). |
| listdir(carpeta) | Devuelve una lista con el contenido –archivos y carpetas– de una carpeta. |
| mkdir(carpeta) | Crea una carpeta. |
| rmdir(carpeta) | Borra una carpeta vacía. |
| remove(archivo) | Borra un archivo. |
| rename(antiguo, nuevo) | Cambia el nombre de un archivo o carpeta, de antiguo a nuevo. |

Los nombres de los archivos y las carpetas, así como las rutas, son cadenas de caracteres; por tanto, los argumentos de las funciones anteriores deben ir entrecomillados.

10.3 El contenido del *filesystem*

El entorno de programación nos muestra de manera esquemática qué archivos hay en el *filesystem* de la placa y cómo están organizados. Imaginemos que el contenido es el que se muestra en la imagen.



La carpeta musica, así como los archivos boot.py y main.py, están en el directorio raíz. A su vez el archivo frecuencias.py está en la carpeta musica, por tanto la ruta para acceder a él será "musica/frecuencias.py", y así sucesivamente.

***i***

Utilicemos el módulo os para analizar el *filesystem* de la imagen. El primer paso es la importación del módulo. Dado que usaremos varias funciones, importamos el módulo completo, por lo que cada vez que usemos una función debemos anteponer el nombre del módulo.

>>> import os

El directorio raíz

Para ver cuál es el contenido del directorio raíz usamos listdir().

>>> os.listdir()

['boot.py', 'musica', 'main.py']

La función listdir() devuelve una lista con los tres elementos que hay en el nivel más alto del *filesystem*.

Dentro de una carpeta

Para ver el contenido de una carpeta usamos la misma función, pero esta vez poniendo como argumento el nombre de la carpeta. Si la carpeta está en el directorio raíz no es necesario poner la ruta, pero si estuviese dentro de otra carpeta habría que indicarlo.

Veamos qué hay en la carpeta musica.

>>> os.listdir("musica")

['korobeiniki.py', 'frecuencias.py', 'config.txt']

Esta lista contiene los elementos (archivos o carpetas) que hay en la ubicación indicada. Dado que la función len de Python devuelve el número de elementos de una lista, podemos usarla para contar cuántos archivos hay dentro de esta carpeta. Aunque en este caso es fácil contarlos “a mano”, para otras aplicaciones puede resultar necesario hacerlo así.

>>> len(os.listdir("musica"))

3

Con listdir() también podemos comprobar si un elemento está o no en una determinada carpeta.

>>> "frecuencias.py" in os.listdir("musica")

True

>>> "notas.py" in os.listdir("musica")

False

El valor lógico devuelto por el operador in nos dice si el archivo existe –True– o no –False– en la carpeta indicada.

10.4 Trabajando con archivos

Una vez que sabemos manejarnos en el sistema de archivos de la placa ha llegado el momento de aprender cómo crear archivos de texto y cómo escribir información en ellos. Para llevar a cabo esta tarea hay que seguir los pasos que se resumen a continuación:

* En primer lugar se abre el archivo.
* Una vez abierto se escribe el contenido.
* Para finalizar se cierra el archivo.

Abrir el archivo

Cada vez que vayamos a escribir en un archivo debemos abrirlo primero. De esto se encarga la función open(). Esta función está definida en el propio lenguaje, por lo que no hay que importar ningún módulo para utilizarla.

|  |
| --- |
| open(archivo, modo) |
| Devuelve un objeto tipo archivo que se usará para trabajar con el archivo. Argumentos:   * archivo: indica el nombre del archivo. Si el archivo no existe, se crea en el momento de abrirlo. Si el archivo no está en el directorio raíz se debe indicar la ruta a su ubicación. * modo: indica si el archivo se abre en modo lectura o escritura. Puede tomar, entre otros, los siguientes valores:   + r: el archivo se abre en modo lectura (*read*). Es el modo por defecto.   + w: el archivo se abre en modo escritura (*write*). Si el archivo existe, al abrirlo se borra su contenido.   + a: el archivo se abre en modo escritura (*append*). Si el archivo existe, el nuevo contenido se añade al final del archivo.   Ambos argumentos son cadenas de caracteres, por lo que deben escribirse entre comillas. |

Escribir en el archivo

Después de abrir el archivo en modo w o a podemos escribir en él. La diferencia entre hacerlo de una manera u otra es importante cuando se abre un archivo ya existente: con w (*write*) el contenido que había en el archivo se sobreescribe y, por tanto, desaparece, mientras que con a (*append*) lo que se escriba se añade a lo que ya había.

Para escribir en un archivo se utiliza write(). Esta función actúa sobre un objeto tipo archivo, que es el objeto devuelto por open().

|  |
| --- |
| write(texto) |
| Escribe un texto en un archivo. Devuelve el número de bytes escritos. Tiene un único argumento:   * texto: es la cadena de caracteres que se quiere escribir en el archivo. |

Cerrar el archivo

Cuando se ha terminado de utilizar el archivo hay que cerrarlo usando la función close(). Al hacerlo se finalizan las acciones que pudieran estar pendientes y se liberan recursos.

Leer el contenido del archivo

Para leer el contenido de un archivo en MicroPython mostraremos simplemente el método read() que, aplicado a un objeto tipo archivo, muestra el contenido del archivo.

|  |
| --- |
| read() |
| Muestra el contenido de un archivo. Si no se indica el argumento lee todo el contenido del archivo. |

10.5 El primer archivo

Probemos en el REPL las instrucciones anteriores. Vamos a crear un archivo llamado prueba.txt en el que escribiremos una frase muy poco original: “Archivo de prueba”.

En primer lugar abrimos el archivo con open(). Dado que aún no existe, antes de abrirlo se va a crear. El primer argumento de open() es el nombre del archivo, que se creará en el directorio raíz si no indicamos ninguna carpeta donde guardarlo. El segundo argumento es el modo en que lo abrimos; pondremos w porque vamos a escribir en él (cuidado, porque si el archivo ya existía este modo sobreescribirá el contenido).

>>> archivo = open("prueba.txt", "w")

Al ejecutar esta instrucción se ha creado en el *filesystem* el archivo prueba.txt (si no se ve en el editor hay que refrescar el listado de archivos). La función open() devuelve un objeto tipo archivo, que será el que utilizaremos como abstracción del archivo. Por tanto, para poder trabajar con él lo hemos guardado en una variable que hemos llamado archivo.

Para escribir en el archivo aplicamos el método write() al objeto archivo que acabamos de crear. Como argumento ponemos la cadena de caracteres que deseamos escribir (entre comillas).

>>> archivo.write("Archivo de prueba")

17

El número devuelto por esta instrucción indica cuántos bytes se han escrito en el archivo.

Finalmente hacemos que el método close() actúe sobre archivo para cerrarlo; de esta manera se finalizan las operaciones necesarias para que el archivo quede correctamente procesado.

>>> archivo.close()

Lectura del archivo

Para comprobar que tenemos lo deseado podemos leer el archivo que acabamos de crear. Cada vez que se quiere acceder a un archivo hay que volverlo a abrir, pero esta vez lo haremos en modo lectura ("r").

>>> archivo = open("prueba.txt", "r")

>>> archivo.read()

'Archivo de prueba'

¡Perfecto!

Más líneas

¿Y si queremos añadir más líneas al archivo? Hay varios detalles que debemos tener en cuenta. El primero es muy obvio: ahora debemos abrir el archivo en modo *append* ("a"). El segundo está relacionado con el hecho de que write() no añade de manera automática los saltos de línea y, en consecuencia, hay que ponerlos a mano (en Python, la secuencia de escape correspondiente al salto de línea es "\n"). Por otro lado hay que tener cuidado con los caracteres especiales (acentos, eñes, etc.), ya que no son soportados en la codificación que, por motivos de espacio, usa MicroPython.

Dicho esto, escribamos alguna línea más en el archivo.

>>> archivo = open("prueba.txt", "a")

>>> archivo.write("\nPrueba 1")

>>> archivo.write("\nPrueba 2")

>>> archivo.write("\nPrueba 3")

>>> archivo.close()

Una vez que hemos terminado de escribir y hemos cerrado el archivo, lo volvemos a abrir para leerlo.

>>> archivo = open("prueba.txt", "r")

>>> archivo.read()

'Archivo de prueba\nPrueba 1\nPrueba 2\nPrueba 3'

>>> archivo.close()

Vemos que lo escrito se ha añadido a lo que ya había en el archivo. Si abrimos el archivo en el editor lo veremos como un documento de texto “normal” con cuatro líneas.

10.6 El bloque **with**

En Python existe una manera alternativa de trabajar con archivos: usando el bloque with. Vamos a añadir más líneas al archivo pero, esta vez, usando una sintaxis diferente.

>>> with open("prueba.txt", "a") as archivo:

... archivo.write("\nPrueba 4")

... archivo.write("\nPrueba 5")

... archivo.write("\nPrueba 6")

Comparemos estas instrucciones con las que usamos anteriormente. Donde antes hicimos archivo = open("prueba.txt", "a") ahora hemos escrito with open("prueba.txt", "a") as archivo. El significado de ambos comandos es el mismo, solo cambia la sintaxis.

Después de los dos puntos con que termina la orden anterior hemos añadido, dentro de un bloque, las instrucciones para escribir más líneas en el archivo. Las instrucciones son las mismas que antes, pero ahora van indentadas (es la manera que tiene Python de delimitar un bloque).

Y eso es todo. Puede parecer que nos hemos olvidado de cerrar el archivo, pero usando el bloque with no es necesario hacerlo explícitamente, ya que se cierra automáticamente una vez se han ejecutado todas las instrucciones del bloque.

Vamos a leer el archivo resultante usando el bloque with.

>>> with open("prueba.txt", "r") as archivo:

... archivo.read()

'Archivo de prueba\nPrueba 1\nPrueba 2\nPrueba 3\nPrueba 4\nPrueba 5\nPrueba 6'

Las dos maneras mostradas de trabajar con archivos son totalmente equivalentes.

10.7 Midiendo la temperatura

Ahora que sabemos trabajar con archivos en MicroPython ha llegado el momento de generar el archivo con las medidas de la temperatura. Antes de hacerlo volvamos un momento al sensor que programamos en el capítulo anterior. Vamos a usar el mismo montaje y el mismo código que entonces, salvo una pequeña modificación: juntaremos en una función todas las instrucciones relacionadas con la medida de la temperatura. La función medir\_temperatura() que mostramos a continuación devuelve el valor medido por el sensor, con una cifra decimal. Si se usa la Raspberry Pi Pico hay que sustituir la medida del voltaje por la instrucción voltaje = TMP36.read\_u16()\*3.3/65535; además se debe cambiar el GPIO14 por otro que tenga funcionalidad ADC (el GPIO 26, 27 o 28).

from machine import Pin, ADC

from time import sleep

TMP36 = ADC(Pin(14))

def medir\_temperatura():

i = 1

suma = 0

num = 20

while i <= num:

voltaje = TMP36.read\_uv()/1000000

suma += voltaje

i += 1

voltaje\_medio = suma/num

temperatura = round((voltaje\_medio - 0.5)\*100, 1)

return temperatura

Por ahora el programa no hace nada, ya que simplemente hemos definido la función medir\_temperatura() pero no la hemos llamado. Podemos comprobar que todo funciona correctamente imprimiendo el valor devuelto por la función

while True:

print(medir\_temperatura())

sleep(1)

El bucle while True que hemos añadido llama a la función medir\_temperatura() y muestra en el REPL la temperatura actualizada cada segundo. Lo que tenemos que conseguir ahora es que esa temperatura se guarde en un archivo.

10.8 El *data logger* en versión minimalista

Modificar el programa anterior para que el valor devuelto por medir\_temperatura() se guarde en un archivo no debería suponer ningún problema. Solo hay que cambiar el contenido del bucle para que, en lugar de imprimir la temperatura, escriba el valor en el archivo. Una posible solución es la que mostramos a continuación.

while True:

with open("datos.csv", "a") as archivo:

archivo.write(str(medir\_temperatura())

archivo.write("\n")

sleep(1)

Le hemos puesto al archivo el nombre datos.csv. Dado que los datos serán posteriormente analizados para extraer conclusiones es conveniente que los escribamos de forma que puedan ser fácilmente procesados. Un formato ampliamente utilizado para guardar tablas de datos es el csv (de *comma-separated values* o “valores separados por comas”). Además de ser compatible con hojas de cálculo y herramientas de software estadístico, un archivo csv se puede abrir con cualquier editor de texto, ya que no es más que un archivo de texto plano. Aunque este tipo de archivos suelen llevar la extensión .csv en su nombre, también se pueden guardar con la extensión .txt, si se prefiere.

La función medir\_temperatura() devuelve un valor de tipo float (un número real) con la temperatura medida por el sensor. Este es el valor que queremos guardar en el archivo, pero hay un problema: el método write() solo acepta como argumento una cadena de caracteres. Para solucionarlo hemos utilizado la función str(), que transforma el argumento en una cadena de caracteres.

Por otro lado, en el formato csv los datos medidos se organizan en filas que se separan entre sí introduciendo un salto de línea, "\n". Dado que write() no añade el salto de línea automáticamente, tenemos que escribirlo explícitamente para que cada medida quede en una línea diferente.

Actualizamos el programa de medida de temperatura con este bucle while y lo ejecutamos desde el editor. Dejamos que se guarden varios valores y, al cabo de unos segundos, paramos la ejecución pulsando Stop. Si todo ha ido bien se ha creado el archivo datos.csv en el directorio raíz del sistema de archivos de la placa.

El archivo de datos

Comprobemos que el archivo se ha creado correctamente escribiendo las siguientes instrucciones en el REPL.

>>> import os

>>> "datos.csv" in os.listdir()

True

Veamos cuál es su contenido.

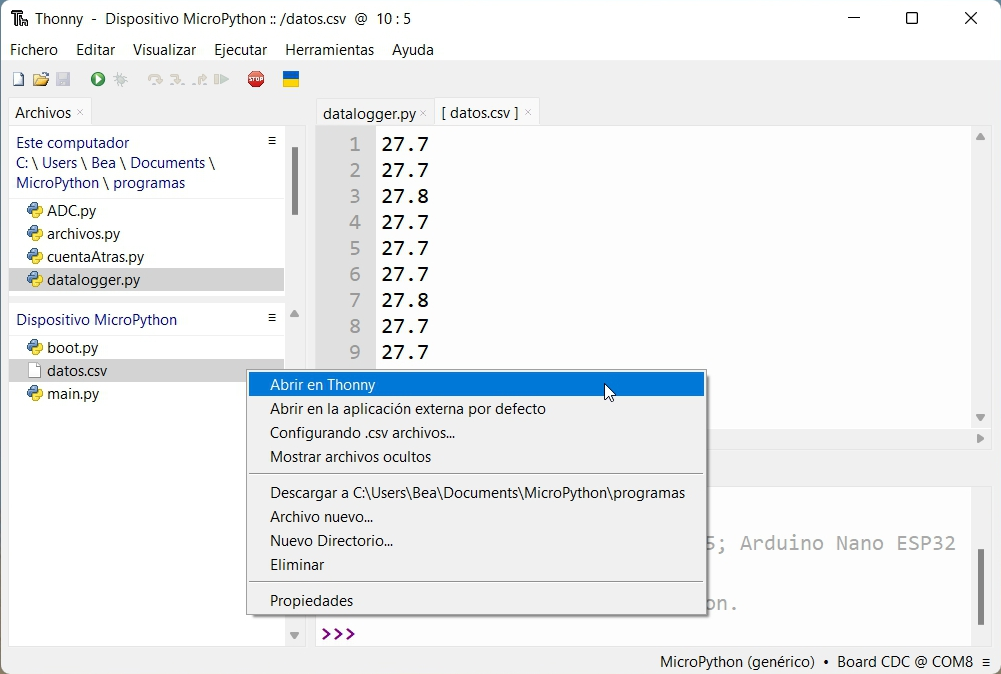
>>> with open("datos.csv", "r") as archivo:

... archivo.read()

'27.7\n27.7\n27.8\n27.7\n27.7\n27.7\n27.8\n27.7\n27.7\n27.6\n'

El archivo contiene lo esperado: varios valores de la temperatura, separados por un salto de línea, \n.

Para ver el contenido del archivo también podemos abrirlo desde el propio entorno de programación. En Thonny, en el panel donde se muestra el contenido del *filesystem*, hacemos clic con el botón derecho del ratón sobre el nombre del archivo (si no aparece el archivo hay que refrescar la visualización del panel). En el menú contextual que se despliega escogemos “Abrir en Thonny” y se abre el documento: vemos los datos escritos en un documento de texto “normal”, cada uno en una línea.

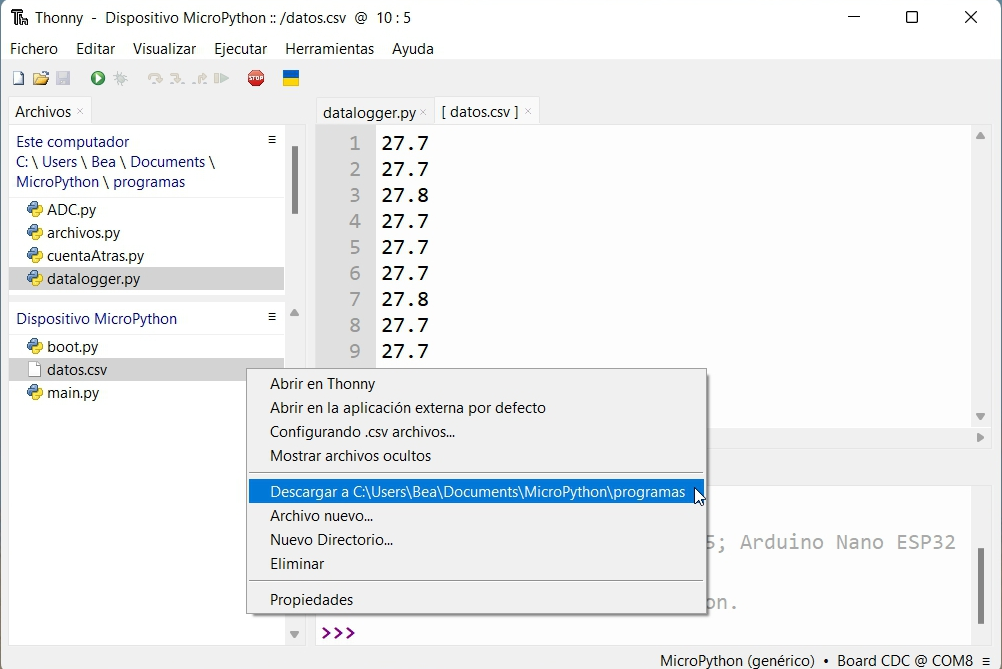


Descargar el archivo

Una funcionalidad imprescindible en un *data logger* es que nos permita descargar en el ordenador el archivo con los datos. Esto debemos hacerlo desde el entorno de programación. En Thonny disponemos de varias opciones.

La primera opción consiste en guardar el archivo de datos igual que hicimos con los programas. Primero tenemos que abrir el archivo en el editor. Una vez abierto, en el menú Archivo escogemos “Guardar como…”. Aparece el menú donde se nos pregunta si deseamos guardarlo en el ordenador o en el dispositivo. Escogemos el ordenador y, en el nuevo menú que aparece, buscamos la ubicación que deseemos. Hacemos clic en Guardar y el archivo se descarga en el disco duro.

Otra opción consiste en hacer clic con el botón derecho del ratón sobre el nombre del archivo en el panel del dispositivo, y en el menú contextual escoger la opción de descargarlo al disco duro (aparece por defecto la ubicación que esté seleccionada en el panel Archivos). Así de fácil.



Con esto ya hemos logrado cumplir el objetivo que nos habíamos marcado: guardar en un archivo los valores medidos por un sensor. ¡Buen trabajo! Sin embargo, a pesar de lo logrado aún queda mucho trabajo por hacer. Esta versión es lo mínimo que necesitamos para tener un *data logger* funcional, pero hay muchos aspectos que se pueden mejorar. Sigamos leyendo.

10.9 El nombre del archivo

Empecemos con las mejoras. Hemos abierto el archivo en modo *append* para que cada dato medido se añada a continuación de los anteriores. Esto está muy bien si usamos el *data logger* para hacer una única toma de datos. Pero si queremos tomar más medidas –en instantes diferentes o en otra localización– y no hemos borrado el archivo anterior, los nuevos datos se añadirán al mismo archivo, algo que en muchas ocasiones no es deseable. Veamos cómo podemos hacer para que, cada vez que se inicia una nueva toma de datos, las medidas se guarden en un archivo diferente.

La estrategia para conseguirlo se basa en una simple comprobación. Antes de crear el archivo datos.csv, comprobamos si ya existe. Si no existe, se crea. Pero si ya existe un archivo con ese nombre (porque ya se ejecutó el *data logger* con anterioridad y no se borró el archivo), entonces creamos un nuevo archivo llamado datos(1).csv para guardar los nuevos datos. Y así sucesivamente.

Pongamos, pues, las siguientes instrucciones al principio del programa.

import os

nombre = "datos"

extension = "csv"

ARCHIVO = nombre + "." + extension

i = 1

while ARCHIVO in os.listdir():

ARCHIVO = "{}({}).{}".format(nombre, i, extension)

i += 1

El nombre del archivo, que hemos llamado ARCHIVO, lo escribimos en tres partes: el nombre, el punto y la extensión. En el bucle while comprobamos si ya existe un archivo con ese nombre. Si no existe –es decir, si ARCHIVO in os.listdir() es False– las instrucciones del bucle no se ejecutan (porque la condición es falsa) y el programa sigue; el nombre del archivo no cambia y es por tanto datos.csv. En el caso de que la condición sea verdadera –porque ya existe un archivo con ese nombre– se ejecutan las instrucciones del bucle. La primera de ellas asigna a ARCHIVO el nombre con el valor de i entre paréntesis. Inicialmente i=1; entonces el nombre será datos(1).csv; en la siguiente instrucción se incrementa en 1 el valor de i (ahora i=2). Como estamos dentro de un bucle, antes de salir se vuelve a comprobar si la condición es cierta. Si es falsa, se sale del bucle y ya tenemos el nuevo nombre del archivo, datos(1).csv. Pero si datos(1).csv ya existía se vuelve a ejecutar el bucle: a ARCHIVO se le asigna un nuevo nombre, datos(2).csv, e i=3. De nuevo se vuelve a la comprobación, y vuelta a empezar hasta que se obtenga un nombre que no existe en el sistema de archivos de la placa.

***i***

Si añadimos las instrucciones anteriores al programa del *data logger*, y ponemos ARCHIVO como nombre del archivo en la instrucción open(), al ejecutarlo varias veces vemos cómo de cada vez se crea un nuevo archivo con numeración consecutiva.

10.10 Añadiendo *feedback*

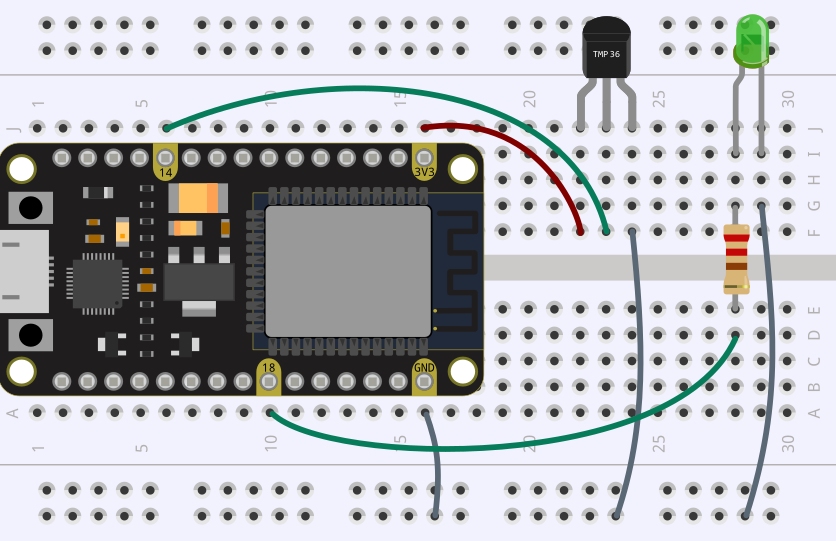
Al igual que en muchos de los proyectos anteriores, pretendemos conseguir un dispositivo que sea independiente del ordenador. Procedamos entonces. Guardamos el programa en la placa con el nombre main.py y, con la placa conectada a una fuente de alimentación –como un *power bank–*, el *data logger* se ejecuta cada vez que se resetea la placa (al pulsar el botón de Reset, o al desconectarla de la corriente y volverla a conectar). Inmediatamente, el dispositivo se pone a tomar medidas, guardándolas en un archivo. Pero, un momento. ¿Cómo podemos estar seguros de que el *data logger* está funcionando? Estaría bien recibir algún tipo de confirmación de que el programa está efectivamente tomando medidas.

Cuando se diseña algún aparato es conveniente añadir un *feedback* que informe al usuario de que las cosas están funcionando como era esperado. Esto se puede llevar a cabo de distintas maneras. En nuestro caso, lo más sencillo y efectivo es añadir un led que parpadee cada vez que se toma una medida. Esta señal no nos garantiza que las medidas se estén tomando correctamente, pero por lo menos nos informa de que el programa funciona.

Como ejercicio, proponemos a los lectores que añadan este elemento al *data logger* que, sin duda, le dará un aspecto mucho más profesional. La solución, en el próximo apartado.

10.11 El *data logger* final

Reunamos en un programa los “trozos” del *data logger* que hemos ido desarrollando a lo largo del capítulo. Para darle una estructura más modular al programa, además de la función para medir la temperatura hemos creado otras dos, una para guardar los datos en el archivo y otra para hacer parpadear el led. Veamos cómo queda.



La configuración

En primer lugar se recogen las instrucciones que configuran los diferentes elementos: se importan los módulos necesarios; se crea el objeto TMP36 como abstracción del sensor de temperatura (que está conectado en el GPIO14), y se crea el objeto LED asociado al led (se ha conectado en el GPIO18). Así mismo se establece la variable INTERVALO con el tiempo de espera entre medidas, en segundos.

import os

from machine import Pin, ADC

from time import sleep, sleep\_ms

TMP36 = ADC(Pin(14))

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

INTERVALO = 1

El archivo de datos

Se añaden las instrucciones para numerar de manera correlativa los archivos de datos. El nombre del archivo se guarda en la variable ARCHIVO.

nombre = "datos"

extension = "csv"

ARCHIVO = nombre + "." + extension

i = 1

while ARCHIVO in os.listdir():

ARCHIVO = "{}({}).{}".format(nombre, i, extension)

i += 1

Medida de la temperatura

Se define la función medir\_temperatura(), que devuelve la temperatura medida por el sensor. Si se usa la Raspberry Pi Pico, la función de cálculo del voltaje debe cambiarse por voltaje = TMP36.read\_u16()\*3.3/65535.

def medir\_temperatura():

i = 1

suma = 0

num = 20

while i <= num:

voltaje = TMP36.read\_uv()/1000000

suma += voltaje

i += 1

voltaje\_medio = suma/num

temperatura = round((voltaje\_medio - 0.5)\*100, 1)

return temperatura

Escritura en el archivo

La función escribir\_temperatura() se encarga de guardar los datos en el archivo. Primero se abre el archivo con el nombre creado al principio del programa. A continuación se llama a la función medir\_temperatura() para obtener el valor de la temperatura y se escribe este valor en el archivo. Finalmente se añade un salto de línea.

def escribir\_temperatura():

with open(ARCHIVO, "a") as archivo:

archivo.write(str(medir\_temperatura()))

archivo.write("\n")

El led

Para hacer que el led parpadee se crea la función parpadeo(), que enciende el led durante 50 milisegundos y lo vuelve a apagar.

def parpadeo():

LED.value(1)

sleep\_ms(50)

LED.value(0)

El bucle

Para finalizar, en el bucle while True se llama a las funciones anteriores con la periodicidad indicada en la variable INTERVALO.

while True:

parpadeo()

escribir\_temperatura()

sleep(INTERVALO)

Los datos

Se escriben las instrucciones anteriores en el orden indicado y se sube el programa a la placa con el nombre main.py . En el momento en que la placa recibe corriente comienza la toma de datos. Una vez finalizada la recogida de datos se conecta la placa al ordenador y, usando el entorno de programación, se descarga el archivo creado. Ya tenemos un *data logger* perfectamente funcional para hacer un seguimiento de la temperatura.

¿Algo más?

Aunque esta versión del *data logger* sirve perfectamente para aplicaciones sencillas –y, sobre todo, nos ha servido para introducir una funcionalidad interesantísima de los microcontroladores–, sigue siendo un prototipo bastante limitado. Proponemos posibles mejoras y ampliaciones que se pueden afrontar con los conocimientos adquiridos a lo largo de este manual.

* El tiempo de espera entre medidas se ha establecido en la función sleep(). Sin embargo, ya hemos comentado que el tiempo así medido es aproximado. Para aplicaciones en las que sea necesaria una medida más precisa del tiempo se puede llevar cuenta del tiempo con la función ticks\_ms() o usando un *timer*.
* En el caso en que la medida del tiempo sea relevante, se puede añadir en el archivo de datos el instante en que se ha medido cada dato.
* La toma de datos comienza en el momento en que se conecta el *data logger* a una fuente de alimentación. Se puede añadir un pulsador para que se empiecen a guardar los datos en el momento en que se pulse el botón.
* Investigando un poco en el módulo os, se puede comprobar el tamaño del archivo generado; de este modo puede encenderse un led de aviso cuando el archivo supere un determinado tamaño.
* Y cualquier otra modificación que sirva para adaptar el proyecto a una aplicación específica.

En resumen

Este capítulo lo hemos dedicado a una funcionalidad muy especial de los microcontroladores con los que hemos trabajado: la posibilidad de utilizar una memoria de almacenamiento persistente –la memoria *flash*– para guardar datos en el dispositivo. Hemos aprendido a usar el módulo os para analizar el contenido del *filesystem* de la placa. Así mismo hemos creado archivos en el sistema de archivos de la placa y hemos visto cómo escribir en ellos y cómo leer su contenido. Finalmente hemos utilizado un sensor de temperatura para fabricar un *data logger* que recoge periódicamente en un archivo de datos la temperatura medida por el sensor.